Centrum Mistrzostwa Informatycznego

Zbiór zadań algorytmicznych CMI

Pod redakcją Bartłomieja Stasiaka

Partner Wiodący Projektu

Politechnika Łódzka

Partnerzy Projektu

AGH

Politechnika Gdańska

Politechnika Warszawska

Politechnika Wrocławska

I love math

Cyfrowy Dialog

Fundusze Europejskie Polska Cyfrowa

Rzeczpospolita Polska

Centrum Projektów Polska Cyfrowa

Unia Europejska Europejski Fundusz Rozwoju Regionalnego

Projekt "Centrum Mistrzostwa Informatycznego" współfinansowany jest ze środków Unii Europejskiej z Europejskiego Funduszu Rozwoju Regionalnego w ramach Programu Operacyjnego Polska Cyfrowa na lata 2014 - 2020

Redaktor zbioru: *Bartłomiej Stasiak* (Politechnika Łódzka)
Redaktor techniczny: *Krzysztof Smółka* (Politechnika Łódzka)

Recenzent: *dr hab. inż. Andrzej Romanowski, prof. uczelni* (Prorektor ds. kształcenia Politechniki Łódzkiej)

Autorzy poszczególnych rozdziałów i zadań:

- *Andrzej Dyrek* (Akademia Górniczo-Hutnicza w Krakowie)
- *Jan Filipowicz* (Politechnika Łódzka)
- *Przemysław Gordinowicz* (Politechnika Łódzka)
- *Andrzej Jastrzębski* (Politechnika Gdańska)
- *Paweł Koszałka* (Akademia Górniczo-Hutnicza w Krakowie)
- *Marcin Markowski* (Politechnika Wrocławska)
- *Łukasz Skonieczny* (Politechnika Warszawska)
- *Jacek Starzyński* (Politechnika Warszawska)
- *Bartłomiej Stasiak* (Politechnika Łódzka)
- *Paweł Tarasiuk* (Politechnika Łódzka)
- *Filip Turoboś* (Politechnika Łódzka)
- *Magdalena Wachulec* (Akademia Górniczo-Hutnicza w Krakowie)

Współpraca merytoryczna:

- *Piotr Broda* (Akademia Górniczo-Hutnicza w Krakowie)
- *Jacek Kaczor* (Akademia Górniczo-Hutnicza w Krakowie)
- *Marzena Krzysztoń* (Akademia Górniczo-Hutnicza w Krakowie)
- *Karol Puchała* (Politechnika Wrocławska)
- *Radosław Roszczyk* (Politechnika Warszawska)
- *Danuta Smołucha* (Akademia Górniczo-Hutnicza w Krakowie)

Zespół techniczny do obsługi konkursów:

- *Bartłomiej Jencz* (Politechnika Łódzka)
- *Katarzyna Jóźwiak* (Politechnika Łódzka)
- *Anna Ptak* (Politechnika Łódzka)
- *Arkadiusz Tomczyk* (Politechnika Łódzka)
- *Tomasz Wiechno* (Politechnika Łódzka)

Partner wiodący projektu:

Politechnika Łódzka

Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

ul. B. Stefanowskiego 22, 90-537 Łódź

Koordynatorzy projektu:

- *Hubert Gęsiarz* (Politechnika Łódzka)
- *Anna Firych-Nowacka* (Politechnika Łódzka)

Koordynator merytoryczny:

- *Bartłomiej Stasiak* (Politechnika Łódzka)

Koordynatorzy / Opiekunowie projektu na uczelniach partnerskich:

- *Beata Chodacka* (Akademia Górniczo-Hutnicza w Krakowie)
- *Anna Domagalska* (Politechnika Gdańska)
- *Michał Gajda* (Politechnika Warszawska)
- *Marcin Markowski* (Politechnika Wrocławska)

ISBN: 978-83-927875-9-4

Instytut Mechatroniki i Systemów Informatycznych
Wydział Elektrotechniki, Elektroniki Informatyki i Automatyki
Politechnika Łódzka
ul. B. Stefanowskiego 22, 90-537 Łódź

Egzemplarz bezpłatny

PARTNER WIODĄCY PROJEKTU

Politechnika Łódzka

PARTNERZY PROJEKTU

AGH
Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Politechnika Gdańska

Politechnika Warszawska

Politechnika Wrocławska

I ♥ math
Stowarzyszenie I love math

Cyfrowy Dialog

PATRONAT

Ministerstwo Edukacji i Nauki

NASK

Ministerstwo Cyfryzacji

Spis treści

1 Wprowadzenie

W roku 2019 wystartował ogólnopolski projekt grantowy pod nazwą *Centrum Mistrzostwa Informatycznego*, w skrócie CMI, nastawiony na podnoszenie kompetencji cyfrowych nauczycieli oraz odkrywanie i rozwijanie talentów informatycznych uczniów szkół podstawowych i ponadpodstawowych. Projekt CMI realizowały następujące uczelnie:

- Politechnika Łódzka (lider projektu);
- Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie;
- Politechnika Wrocławska;
- Politechnika Warszawska;
- Politechnika Gdańska;

oraz stowarzyszenia:

- Cyfrowy Dialog;
- I love math.

1.1 Cel projektu

Głównym celem projektu było podniesienie kompetencji kadry dydaktycznej, tj. osób prowadzących zajęcia pozalekcyjne rozwijające zainteresowania informatyczne, a także aktywizacja młodzieży uzdolnionej informatycznie, pobudzanie kreatywności oraz promowanie współpracy zespołowej w ramach kół informatycznych.

Projekt CMI stanowił kompleksową koncepcję wzmocnienia polskiej edukacji informatycznej ukierunkowanej na kształcenie uzdolnionych uczniów przy zaangażowaniu najlepszych uczelni technicznych w kraju. Dzięki realizacji wskazanych celów projekt wzmocnił u uczniów chęć rozwoju zainteresowań z zakresu algorytmiki i programowania, posłużył także upowszechnieniu idei konkursów informatycznych oraz wyłonił zespoły zdolne do podjęcia rywalizacji w zawodach informatycznych na poziomie krajowym oraz międzynarodowym. CMI przyczyniło się do podniesienia kompetencji dydaktycznych ponad 1500 osób prowadzących koła informatyczne oraz ponad 12 000 uczniów.

Projekt skierowany był do nauczycieli szkół podstawowych, ponadpodstawowych, nauczycieli akademickich oraz innych osób dorosłych wykazujących odpowiednie predyspozycje oraz zainteresowanie pracą z uzdolnioną młodzieżą – uczniami z klas IV-VI szkół podstawowych, uczniami klas VII-VIII szkół podstawowych, uczniami szkół średnich w tym liceów ogólnokształcących i techników oraz szkół branżowych I i II stopnia.

Centrum Mistrzostwa Informatycznego zapewniło uczestnikom udział w prestiżowym projekcie o zasięgu ogólnopolskim, w tym możliwość stałego kontaktu ze społecznością renomowanych uczelni technicznych, a także kooperację środowisk dydaktycznych, kadry akademickiej i nauczycieli szkół podstawowych i średnich.

- Całkowita wartość projektu: 50 239 096,15 zł.
- Kwota dofinansowania: 49 885 951,15 zł.
- Dofinansowanie projektu z UE: 42 218 480,45 zł.
- Dofinansowanie projektu z budżetu państwa: 7 667 470,70 zł.
- Termin realizacji: 11.12.2018 – 31.12.2023.

1.2 Konkursy i zawody

Uczniowie rozwijając zainteresowania algorytmiczne, programistyczne oraz robotyczne, mieli szansę wykorzystać swoją wiedzę w praktyce, biorąc udział w konkursach i zawodach CMI. Uczniowie mogli sprawdzić się między innymi w *Zawodach Algorytmicznych* (dla uczestników kółek algorytmicznych) oraz *Lidze Algorytmicznej* (dla uczestników wszystkich typów kółek). Udział w zawodach był doskonałą okazją nie tylko do tego, żeby sprawdzić swoją wiedzę w praktyce, ale również aby wykazać się we współpracy z kolegami i koleżankami z kółka, ponieważ w zaproponowanych formach współzawodnictwa uczniowie mogli wziąć udział tylko zespołowo.

Zawody Algorytmiczne – były cyklicznie organizowanym wydarzeniem (odbyły się cztery edycje w kolejnych latach), w którym poprzez współzawodnictwo, ale przede wszystkim poprzez pracę w grupie i wspólne rozwiązywanie zadań, uczniowie mogli sprawdzić w praktyce wiedzę zdobytą na kółkach CMI.

Zawody były trzyetapowe i obejmowały kolejno:

- etap lokalny – w systemie on-line;
- etap regionalny – odbywający się jednocześnie na wszystkich uczelniach partnerskich;
- etap ogólnopolski – czyli finał zawodów, który odbywał się na Politechnice Łódzkiej.

Uczniowie współzawodniczyli ze sobą w 3 kategoriach, w zależności od poziomu edukacyjnego, zaawansowania oraz języka programowania:

- **Skrzaty** – poziom podstawowy, dla uczniów szkół podstawowych z klas IV-VIII, którzy programowali w środowisku Scratch;
- **Gnomy** – poziom średnio-zaawansowany, dla uczniów z klas IV-VIII szkół podstawowych oraz I klasy szkół ponadpodstawowych, którzy swoje programy tworzyli w językach Python oraz C++;
- **Gremliny** – poziom zaawansowany, dla uczniów z klas IV-VIII szkół podstawowych oraz szkół ponadpodstawowych, którzy również programowali w Pythonie oraz w C++.

Liga Algorytmiczna była doskonałym praktycznym przygotowaniem do Zawodów Algorytmicznych CMI, a także innych konkursów algorytmicznych i informatycznych, takich jak Olimpiada Informatyczna i Olimpiada Informatyczna Juniorów. Uczestnicy kół brali w niej udział zespołowo (jedno koło = jeden zespół). Liga składała się z kolejnych rund, w których uczestnicy za rozwiązane zadania otrzymywali punkty. Łączna liczba zdobytych przez zespół punktów decydowała o jego pozycji w prowadzonym na bieżąco rankingu.

Niniejszy zbiór obejmuje wybór zadań powstałych w projekcie CMI na potrzeby przedstawionych wyżej form współzawodnictwa. Trzeba tu wyraźnie zaznaczyć, że nie jest to *podręcznik* algorytmiki ani programowania. Oferta wydawnicza na polskim rynku jest w tym zakresie bardzo bogata i zainteresowany czytelnik z pewnością znajdzie wiele wartościowych pozycji wprowadzających w tajniki pisania kodu w popularnych językach programowania [20][10][3][13][9][19][12][11] oraz tworzenia i analizy algorytmów [4][22][2][8][7][23]. Z drugiej strony, prezentujemy coś więcej niż zbiór zadań [5][6] – ta książka jest swojego rodzaju podsumowaniem „algorytmicznej" części projektu CMI i odzwierciedla w pierwszym rzędzie niezwykle szeroki przekrój jego uczestników i beneficjentów. Platforma CMI była miejscem spotkań uczniów piszących swoje pierwsze programy w Scratchu i doświadczonych uczestników Olimpiady Informatycznej, nauczycieli szkół podstawowych i wykładowców akademickich. Ta różnorodność była nie lada wyzwaniem również dla naszych autorów, którzy musieli opracowywać zadania konkursowe na każdym możliwym poziomie trudności. Dzięki temu jednak,

możemy teraz zaprezentować pozycję dość unikatową na tle innych publikacji dydaktycznych z zakresu algorytmiki, obejmującą swoim materiałem szerokie spektrum problemów, prowadzącą niejako „za rękę” młodego czytelnika w jego rozwoju i dostarczającą mu wyzwań na miarę jego rosnących umiejętności programistycznych. Żywimy głęboką nadzieję, że zaproponowane tu zadania, wykorzystane w ramach kół informatycznych lub indywidualnie, będą dobrze spełniać swoją rolę, pomagając odkrywać i rozwijać talenty uczniowskie w zakresie programowania, algorytmiki i umiejętności rozwiązywania problemów informatycznych.

1.3 Zawartość zbioru

Zadania prezentowane w niniejszym zbiorze pojawiały się na przestrzeni lat 2020 – 2023 podczas poszczególnych etapów Zawodów Algorytmicznych oraz rund Ligi Algorytmicznej CMI. Znajdziemy tu zarówno zadania dla najmłodszych uczestników (z kategorii Skrzaty) – do rozwiązania w Scratchu, jak i poważniejsze wyzwania algorytmiczne (dla Gnomów i Gremlinów) implementowane w C++ i Pythonie. Cały zbiór został podzielony na 9 działów:

I Proste zadania z duszkami – animacje;
II Podstawowe operacje na liczbach i napisach;
III Wczytywanie, wypisywanie i przechowywanie danych;
IV Sortowanie;
V Dwuwymiarowe struktury danych i elementy geometrii obliczeniowej;
VI Zadania obliczeniowe, elementy teorii liczb, podzielność, algorytm Euklidesa;
VII Algorytmy tekstowe;
VIII Grafy;
IX Programowanie dynamiczne.

Układ materiału ma w założeniu odzwierciedlać rosnący poziom trudności, ale dotyczy to głównie trzech pierwszych działów (i – do pewnego stopnia – kolejności zadań w każdym z działów)[1]. Rozpoczynamy od zadań z animacją duszków w Scratchu, aby stopniowo wprowadzić elementy obliczeniowe oraz przetwarzanie liczb i tekstów zapisanych z użyciem list, a później także bardziej złożonych struktur danych. Już w drugim dziale pojawiają się pierwsze implementacje w C++ i Pythonie, a w kolejnych działach normą staje się prezentacja rozwiązań tych samych zadań równolegle we wszystkich trzech językach programowania. W intencji autorów, stanowi to czynnik wspierający naturalny proces ewolucji młodych programistów w stronę języków „tekstowych” i rozwijający umiejętności w zakresie analizy algorytmów i myślenia komputacyjnego. W dalszych działach, rozwiązania prezentowane są już zwykle tylko w C++ i Pythonie, z uwagi na brak w Scratchu bibliotek standardowych z gotowymi implementacjami bardziej złożonych algorytmów i struktur danych (co niejako w naturalny sposób stanowi element motywujący do zapoznania się z tymi językami).

Każdy dział rozpoczyna się wprowadzeniem do danej tematyki, wyjaśniającym podstawowe pojęcia i metody, które następnie wykorzystywane są w opisach rozwiązań zadań zawartych w tym dziale. Same zadania mają określoną strukturę, obejmującą część fabularną, opis zadania, specyfikację wejścia/wyjścia oraz omówione przykłady konkretnych danych wejściowych i wyjściowych[2].

[1] Oczywiście w każdym dziale znajdziemy zarówno zadania łatwiejsze (dla Skrzatów lub Gnomów), jak i trudniejsze (dla Gremlinów).

[2] Aby ułatwić czytelnikowi nawigację po treści zbioru, tytuł każdego zadania (oraz każdego wprowadzenia) poprzedzony został numerem działu w nawiasie.

Po każdym zadaniu znajduje się sekcja prezentująca jego rozwiązanie. W zależności od tematyki zadania, możemy znaleźć tam omówienie kluczowych elementów rozwiązania, przykładowe implementacje w postaci fragmentów lub całości kodu, rysunki pomocnicze, wyprowadzenia odpowiednich wzorów matematycznych lub dowody formalne. Zaprezentowane mogą być niekiedy różne warianty rozwiązania, różniące się np. złożonością obliczeniową. Gotowe implementacje są dostępne w formie linku do pliku z rozwiązaniem. Może to być link do pliku projektu Scratcha (rozszerzenie .sb3) i/lub do plików z kodem w języku Python (rozszerzenie .py) i C++ (rozszerzenie .cpp). Z uwagi na specyfikę środowiska Scratch, przyjęto założenie, że jeśli zadanie występuje w dwóch wersjach: dla Skrzatów i dla Gnomów, wówczas zarówno jego treść, jak i opis rozwiązania prezentowane są niezależnie dla obu tych kategorii. Warto zauważyć, że niektóre zadania prezentowane są w języku angielskim – podobnie jak miało to miejsce podczas zawodów CMI. W takich przypadkach, ich rozwiązania omawiane są jednak zawsze po polsku.

Pliki z implementacjami umieszczono w katalogu `zadania`, który znajduje się w tym samym katalogu co niniejszy zbiór zadań w wersji PDF. Dzięki temu, wykorzystując przeglądarkę do otwarcia niniejszego dokumentu, można bez problemu przechodzić do właściwych rozwiązań klikając jedynie na zamieszczone tu linki. W linkach tych widoczne są ścieżki dostępu do odpowiednich podkatalogów i plików, co pozwoli na znalezienie i pobranie rozwiązań również czytelnikom wersji drukowanej.

Oprócz kodów źródłowych, w większości katalogów można także znaleźć plik `test.zip`, zawierający pliki tekstowe z przykładowymi danymi wejściowymi i oczekiwanymi wyjściowymi. Zawarte tam przypadki testowe obejmują również takie, w których występują duże rozmiary wejścia/wyjścia (a także różnego rodzaju przypadki szczególne), co ma w założeniu stanowić istotną pomoc w samodzielnej konstrukcji i implementacji efektywnych algorytmów o niskiej *złożoności obliczeniowej* [4][7][2].

O tym, czym właściwie jest złożoność obliczeniowa i dlaczego jest dla nas ważna, a także – krótko – o wspomnianych wyżej trzech językach programowania opowiemy za chwilę, ale już teraz zapraszamy Cię, drogi Czytelniku do... sięgnięcia po wygodną klawiaturę i przygotowania się na spotkanie z czekającymi na Ciebie za parę stron wyzwaniami algorytmicznymi...

Powodzenia!
Autorzy

2 Zmagania konkursowe – o co i czym...

Mistrzostwa, to walka... oczywiście o zwycięstwo. Ale jak to wygląda w kontekście *Mistrzostwa Informatycznego*? Co jest naszym zwycięstwem i jakie mamy środki, żeby je osiągnąć?

Sprawa nie jest taka prosta, jak mogłoby się wydawać. Napisanie działającego programu jest oczywiście pewnym sukcesem (programiści C++ powiedzieliby: napisanie programu, który najpierw da się w ogóle skompilować...). Jednak chcielibyśmy, żeby nasz program nie tylko działał, ale jeszcze działał dobrze – i to w każdym, może również w mniej typowym lub nawet całkiem nietypowym przypadku.

Sprawdzanie wszystkich przypadków – tych typowych i nietypowych – może być dość żmudne, szczególnie, że powinniśmy powtarzać je właściwie po każdej większej zmianie kodu czy próbie udoskonalenia naszego programu. Z tego powodu wymyślono tzw. *sprawdzarki*, czyli specjalne programy do testowania innych programów, które wyręczają nas w sprawdzaniu poprawności naszego kodu. Sprawdzarki, określane czasem także jako programy sędziujące, działające zwykle w trybie on-line jako tzw. systemy OJ (ang. *On-line Judge*), są powszechnie wykorzystywane w różnego rodzaju zawodach i konkursach algorytmicznych. Oczywiście również w projekcie CMI, zadania uczestników poszczególnych etapów Zawodów Algorytmicznych i Ligi Algorytmicznej były testowane przez specjalne sprawdzarki (dla C++, Pythona i – co już nie było takie do końca oczywiste – dla Scratcha).

Użycie sprawdzarki niesie ze sobą kilka korzyści, takich jak możliwość automatycznego (i szybkiego!) sprawdzania poprawności programów nadsyłanych przez wielu użytkowników jednocześnie, czy duża powtarzalność i obiektywizm uzyskiwanych ocen. Sprawdzarka przydaje się jednak do weryfikacji jeszcze jednej ważnej rzeczy, która – oprócz tego, czy program zwraca poprawne wyniki, czy nie – często ostatecznie decyduje o sukcesie lub porażce. Chodzi o *czas działania*.

Tworząc swoje programy powinniśmy zawsze robić to tak, aby działały jak najszybciej, wykonując minimalną liczbę operacji potrzebnych do uzyskania poprawnych rezultatów, a także, aby zużywały jak najmniej pamięci operacyjnej. To jest w gruncie rzeczy nasz cel ostateczny i o to walczymy podczas zawodów, a sprawdzarka – w oparciu o odpowiednio przygotowane zbiory danych testowych – pomaga nam stwierdzić, czy cel ten udało się nam osiągnąć.

To odpowiedź na pierwszą część tytułowego pytania, którą za chwilę jeszcze sobie uzupełnimy, omawiając zagadnienie złożoności obliczeniowej [4][7][2]. Odpowiedzmy tu jeszcze krótko na drugą, czyli – czym dysponujemy w naszych zmaganiach, z jakich narzędzi możemy korzystać w kontekście implementacji, uruchamiania i testowania projektowanych przez nas algorytmów.

2.1 Scratch

Scratch to popularne narzędzie do nauki programowania, pozwalające na tworzenie programu w sposób wizualny – poprzez dokładanie kolejnych instrukcji do wykonania w postaci kolorowych bloków pasujących do siebie jak puzzle [20][10]. Ten sposób programowania jest dość uciążliwy na dłuższą metę (dla doświadczonych programistów pisanie kodu w postaci tekstu jest dużo szybsze niż przeciąganie bloków myszką), jednak niepodważalną zaletą Scratcha jest przejrzysta, graficzna reprezentacja struktury kodu programu w sposób bez-

pośrednio równoważny diagramowi blokowemu. Znacznie redukuje się dzięki temu konieczność „tłumaczenia" formalnego zapisu kodu programu na postać abstrakcyjną, odpowiadającą semantyce algorytmu. Z tego względu Scratch stanowi nieocenioną pomoc przy nauce podstaw programowania dla najmłodszych, wspomagając rozwój umiejętności tworzenia algorytmów i myślenia komputacyjnego.

Scratch to nie tylko wizualny język programowania, ale całe środowisko programistyczne (dostępne m.in. dla systemu Windows, macOS, Linux, a także on-line) wspierane przez społeczność zajmującą się jego rozwojem i publikującą projekty zrealizowane na jego podstawie. Został opracowany na początku XXIw. w jednym z laboratoriów badawczych (MIT Media Lab) na uczelni Massachusetts Institute of Technology przez Mitchela Resnicka i jego współpracowników. Obecnie dostępne są trzy wersje Scratcha:

- 1.4
- 2.0 (2013)
- 3.0 (2019)

Scratch może być pobrany https://scratch.mit.edu/download i zainstalowany lokalnie, na komputerze użytkownika, ale można również korzystać z wersji on-line za pomocą przeglądarki internetowej. Ta druga opcja wymaga założenia konta na stronie https://scratch.mit.edu/, co daje nam automatycznie większe możliwości w zakresie udostępniania i popularyzacji naszych projektów, a także korzystania z projektów udostępnionych przez innych użytkowników.

Znakomitym źródłem wiedzy na temat Scratcha są materiały dostępne na stronie stowarzyszenia Mistrzowie Kodowania. Znajdziemy tam między innymi filmy instruktażowe na temat instalacji Scratcha i zakładania konta on-line.

Oprócz zapoznania się z powyższymi materiałami, najlepszym sposobem nauki programowania w Scratchu jest własnoręczne eksperymentowanie, pisanie i uruchamianie swoich programów. Scratch, to środowisko edukacyjne, zapewniające wyjątkowo przejrzysty i intuicyjny interfejs graficzny, dostępny nawet dla użytkowników bez wcześniejszego przygotowania, czy doświadczenia programistycznego.

2.2 C++

Język C++ został zaprojektowany przez Bjarne Stroustrupa w 1979 r. jako rozszerzenie języka C o mechanizmy obiektowe abstrakcji danych [19][9][12][11]. Zyskał dużą popularność ze względu na następujące cechy:

- możliwość stosowania (nawet w ramach pojedynczej aplikacji) kilku paradygmatów programowania: proceduralnego, obiektowego i generycznego;
- wysoka wydajność kodu wynikowego;
- bezpośredni dostęp do sprzętu i funkcji systemu operacyjnego, na którym uruchomiona jest aplikacja napisana w tym języku;
- łatwość tworzenia i korzystania z bibliotek;
- bogactwo bibliotek standardowych i publicznych, tworzonych przez użytkowników języka;
- przenośność kodu źródłowego programu pomiędzy różnymi platformami sprzętowymi i systemami operacyjnymi.

Język C++ jest językiem ogólnego przeznaczenia służącym do implementacji wszelkiego rodzaju aplikacji, gier komputerowych, systemów operacyjnych, często też stosowany jest jako główny język programowania

systemów wbudowanych (małych systemów komputerowych opartych o tzw. mikrokontrolery, umożliwiające sterowanie prostymi urządzeniami użytku codziennego, takimi jak automaty do kawy, lodówki, wyświetlacze, czujniki temperatury czy zadymienia, itp.). Gama zastosowań języka jest bardzo szeroka, często jest on także określany „językiem najbardziej niskiego poziomu spośród języków wysokiego poziomu", ponieważ programista ma możliwość zarówno bezpośredniego programowania sprzętowego (poziom niski) jak i programowania w sposób bardzo abstrakcyjny (poziom wysoki), nawet w ramach pojedynczej aplikacji czy projektu informatycznego. Wszystkie te cechy sprawiają, że język ten, pomimo upływu lat, cieszy się niesłabnącą popularnością w środowisku programistycznym.

2.3 Python

Python jest językiem młodszym niż C i C++, bo powstał dopiero w latach 90 XX wieku (jego twórcą jest Guido van Rossum) [3][13]. Jest często wykorzystywany jako język skryptowy, a do jego charakterystycznych cech należy zastosowanie wcięć (tabulacji/spacji) do wyznaczania bloków kodu. Ciekawą i użyteczną częścią tego języka jest wbudowany system liczb nieskończonej precyzji. Python posiada bardzo czytelną składnię, pozwalającą na użycie wyrażeń o dużej sile ekspresji. Język ten oferuje programistom pokaźną bibliotekę standardową, zawierającą między innymi parsery HTML'a czy serwer HTTP. Co więcej, korzysta z systemu zarządzania pakietami, który pozwala na proste dołączanie dodatkowych bibliotek, przy czym stosunkowo łatwo jest również dodawać biblioteki napisane w językach kompilowanych, takich jak C++.

Cechy te sprawiają, że – choć Python nie jest zwykle pierwszym wyborem przy nauce algorytmiki (m.in. z uwagi na wydajność oraz ograniczony wpływ programisty na zarządzanie pamięcią) – jest jednak niezwykle wysoko ceniony w wielu zastosowaniach, takich jak np. praca z narzędziami sztucznej inteligencji, analiza danych i uczenie maszynowe.

XXI wiek jest czasem nieustannie rozwijanych technologii oraz gigantycznych korporacji sektora usług IT. Konkurencja w oprogramowaniu nigdy wcześniej nie była tak wielka, wcześniej nie istniała też presja ze strony projektów *open source*, wymuszająca jakość oraz rozumienie potrzeb użytkowników aplikacji. Obfitość wyboru w zakresie narzędzi programistycznych prowadzi do sytuacji, w której użytkownicy pracujący z danymi językami programowania wyrażają szczegółowo swoje preferencje i oczekiwania, a badanie ich opinii stanowi ważny czynnik prognostyczny i decyzyjny w rozwoju technologii.

Ciężko jest jednak opracować odpowiednio miarodajne wskaźniki opisujące zainteresowanie poszczególnymi językami programowania. Obecnie najwyżej oceniane są dwa takie wskaźniki:

- TIOBE - mierzy wyszukania zagadnień związanych z danym językiem,
- PYPL - mierzy wyszukania, ale ogranicza się do materiałów służących do nauki danej technologii.

Zgodnie z tymi wskaźnikami (aktualizacja – sierpień 2020):

- wg. TIOBE - Python jest trzecim najczęściej wyszukiwanym językiem programowania z udziałem 9.69%,
- wg. PYPL - Python jest najczęściej poszukiwanym językiem programowania do nauki z udziałem 31,59% oraz przyrostem w ostatnim roku 3,3%. Jest jednym z najczęściej wybieranych języków programowania na całym świecie do nauki i rozwoju umiejętności.

3 Złożoność obliczeniowa

Wiemy już z poprzedniego rozdziału, że nasze algorytmy powinniśmy pisać tak, aby działały możliwie jak najszybciej. Zastanawiając się nad tym nieco dokładniej zauważymy, że szybkość działania programów zależy od wielu różnych czynników, takich jak:

- rozmiar danych;
- rodzaj komputera na którym jest wykonywany program;
- język programowania, który został użyty do napisania programu;
- rodzaj i parametry kompilatora/interpretera;
- sposób napisania programu (czyli implementacji algorytmu);
- sposób konstrukcji samego algorytmu.

Na rozmiar danych nie mamy zwykle wpływu, gdyż jest to integralny składnik rozwiązywanego problemu, wyznaczający jego poziom trudności. Nasz wpływ na kolejne trzy czynniki też jest ograniczony, szczególnie w kontekście zadań konkursowych uruchamianych przez sprawdzarkę w konkretnym środowisku testowym. Oczywiście jest prawdą, że programy napisane w C++ działają zwykle szybciej niż analogiczne programy w Pythonie, że różne kompilatory generują mniej lub bardziej zoptymalizowany kod wynikowy (również w zależności od użytych parametrów kompilatora[1]), a komputer najnowszej generacji zapewne wykona nasz program szybciej niż maszyna sprzed dekady. To wszystko ma jednak małe znaczenie w zestawieniu z dwoma ostatnimi czynnikami z powyższej listy, czyli sposobem konstrukcji i implementacji algorytmu.

Rozróżnienie między samym algorytmem, a jego implementacją (czyli sposobem zapisu w konkretnym języku programowania) nie jest tak do końca jednoznaczne, bo zależy poniekąd od stopnia szczegółowości opisu algorytmu. Przykładowo – dana operacja, taka jak posortowanie ciągu wartości lub pomnożenie dwóch liczb, może być potraktowana jako pojedynczy krok algorytmu (i zrealizowana za pomocą gotowej funkcji bibliotecznej `sort` czy standardowego operatora mnożenia) lub też „rozpisana" szczegółowo w kolejnych krokach algorytmu, stając się tym samym jego istotną częścią, być może nawet decydującą o ostatecznym czasie wykonania[2].

Tak czy inaczej, trzeba podkreślić, że to przede wszystkim od nas – twórców algorytmów i programów – zależy jak szybko będą działać, bo nieoptymalnie zaprojektowany i zaimplementowany algorytm *zawsze* ostatecznie okaże się zbyt wolny, nawet na najszybszym komputerze, jeśli tylko rozmiar przetwarzanych danych będzie odpowiednio duży. To ostatnie stwierdzenie spróbujemy teraz przeanalizować w sposób nieco bardziej formalny.

Przede wszystkim przyjmiemy założenie, że nie będziemy na razie mierzyć faktycznego czasu działania programu za pomocą stopera (ani – co jest zwykle nieco lepszym pomysłem – za pomocą funkcji do pomiaru czasu dostępnych w różnych bibliotekach, czy w samym systemie operacyjnym). Taki pomiar jest dość mocno zależny od różnych nieistotnych czynników – tych wymienionych wyżej i wielu innych, np. od liczby zadań wykonywanych akurat w tle przez system operacyjny, czy nawet od tego czy nasz laptop pracuje aktualnie na baterii (tryb „oszczędzania energii" oznacza zwykle znacznie wolniejsze działanie procesora).

Zamiast tego, spróbujemy zacząć od policzenia, ile operacji wykona nasz algorytm podczas swojego działania.

---

[1] Przykładowo, program w C++ skompilowany w trybie „Release" (czyli z używaną również przez sprawdzarki OJ opcją kompilatora -O2 lub -O3) może okazać się nawet kilkukrotnie szybszy niż skompilowany w domyślnym trybie „Debug".

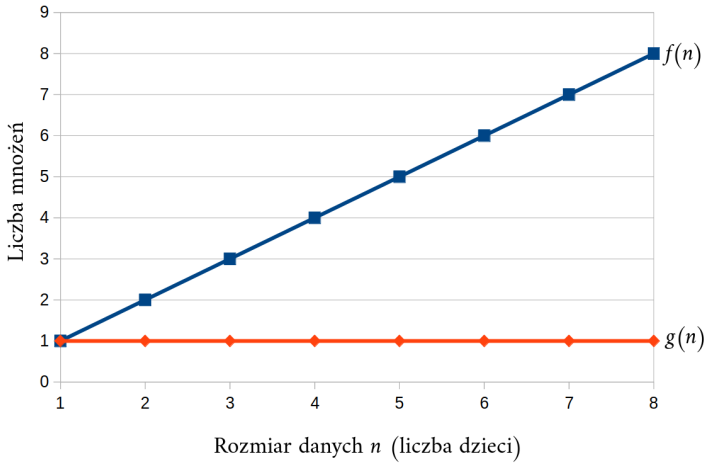
[2] Zauważmy, że np. mnożenie liczb całkowitych o liczbie cyfr przekraczającej rozmiar typów wbudowanych w język programowania, faktycznie wymaga napisania odpowiedniego algorytmu przez programistę.

Oczywiście czas trwania każdej operacji może być inny, ale uprościmy sobie nieco problem, wybierając jeden, konkretny rodzaj operacji (np. operacje arytmetyczne, operacje porównywania liczb, zamiany miejscami pary elementów, etc.), który dominuje w naszym algorytmie i ma największy wpływ na jego czas działania.

Przykładowo, możemy zastanowić się, ile operacji arytmetycznych wymaga policzenie sumy oszczędności dwójki dzieci wracających z wycieczki (tzn. kieszonkowego, którego udało się nie wydać na lody i pamiątki). Ali zostało A złotych, Bartkowi B złotych, a do obliczenia wyniku $A+B$ wystarczy oczywiście jedno dodawanie. Jeśli wycieczka była zagraniczna, oszczędności dzieci trzeba przeliczyć np. z euro na złotówki po aktualnym kursie wynoszącym k PLN za EUR. Ala ma więc A euro, czyli $k \cdot A$ zł, Bartek ma B euro, czyli $k \cdot B$ zł, a ich wspólne oszczędności, to łącznie $(kA + kB)$ zł. Potrzebujemy więc dodatkowo dwóch operacji mnożenia.

Jak łatwo zauważyć, ten sam wynik możemy uzyskać sprytniej (wykorzystując tylko jedno mnożenie), jeśli najpierw dodamy oszczędności w euro, a przeliczymy na złotówki dopiero na koniec, czyli tak: $k(A + B)$. Ta redukcja liczby mnożeń wydaje się niewielka, ale jeśli dzieci byłoby więcej, np. 10, 100, 1000,... to przeliczanie osobno oszczędności każdego z nich wymagałoby 10, 100, 1000 mnożeń: $(kA + kB + kC + kD + ...)$, podczas gdy nasz sprytniejszy sposób obliczania wymagałby wciąż tylko jednego mnożenia: $k(A + B + C + D + ...)$.

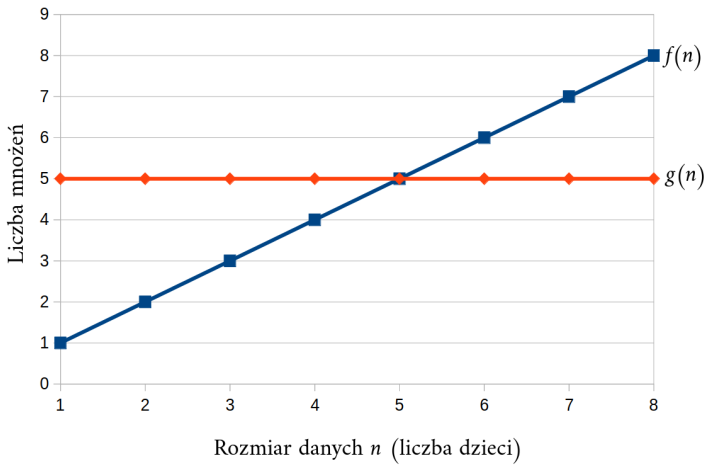
Jeśli potraktujemy mnożenie jako operację dominującą w naszym algorytmie, rozmiar danych (czyli liczbę dzieci) oznaczymy przez n, a liczbę operacji dominujących, jakie należy wykonać dla danej liczby dzieci – przez $f(n)$ w pierwszym podejściu, a przez $g(n)$ w drugim (tym „sprytniejszym"), wówczas otrzymamy następujący wykres:



Zauważmy tu kilka rzeczy. Po pierwsze, wyrażenie $f(n)$ lub $g(n)$ nazywamy w matematyce *funkcją* (w tym wypadku funkcja ta przypisuje każdemu możliwemu rozmiarowi danych liczbę operacji mnożenia, jakie należy wykonać). Formalnie rzecz biorąc, jest to funkcja ze zbioru liczb naturalnych w zbiór liczb rzeczywistych dodatnich $f : \mathbb{N} \to \mathbb{R}_+$. Ponieważ liczba dzieci jest zawsze całkowita, moglibyśmy więc – zamiast o wartościach funkcji: $f(1), f(2), f(3), ...$ – mówić równoważnie o wyrazach *ciągu* $f_1, f_2, f_3, ...$. Podobnie, liczba mnożeń też jest całkowita, więc moglibyśmy mówić o funkcji w zbiór liczb naturalnych, zamiast rzeczywistych dodatnich, jednak wygodniej będzie nam przyjąć nieco ogólniejsze założenia i dopuścić również wartości rzeczywiste. Dzięki temu łatwo przejdziemy w dalszej części ze zliczania operacji na pomiar faktycznego czasu obliczeń w sekundach.

Wracając do naszego wykresu – zauważmy, że obie funkcje różnią się diametralnie między sobą. Funkcja $g(n)$ jest funkcją stałą, a więc jej wartość nie zależy od rozmiaru danych wejściowych n, w przeciwieństwie do funkcji $f(n)$. Mówiąc bardziej formalnie, obie funkcje mają inną *klasę złożoności obliczeniowej* (funkcja $g(n)$

– stałą, a funkcja $f(n)$ – liniową). Zauważmy też, że wobec stałości funkcji $g(n)$, konkretna wartość, jaką przyjmuje (tutaj: jeden), nie ma tak naprawdę znaczenia. Gdyby przyjmowała np. wartości 5 razy większe (czyli wszędzie 5):

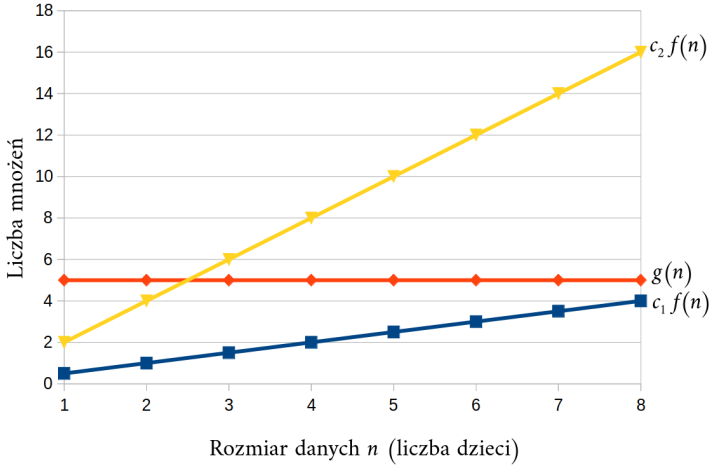


to i tak dla dostatecznie dużego rozmiaru danych (tu: dla liczby dzieci n większej niż $n_0 = 5$), będzie „wygrywać" z funkcją $f(n)$ pod względem liczby mnożeń. Podobnie, pomnożenie wartości funkcji $f(n)$ przez dowolną stałą (2, 5, a nawet $\frac{1}{2}$, czy $\frac{1}{5}$) również niewiele zmieni, bo zawsze znajdziemy taką graniczną liczbę dzieci n_0, że funkcja $g(n)$ zacznie wygrywać (i będzie już wygrywać dla wszystkich $n > n_0$).

Nasze spostrzeżenia prowadzą do wniosku, że najbardziej interesującą częścią naszego wykresu jest jego prawa strona. Pytanie „co będzie dla dostatecznie dużego n" możemy rozszerzyć, przechodząc z n do nieskończoności, a więc rozważając tzw. *oszacowanie asymptotyczne* przebiegu naszych funkcji.

Jako symbol oszacowania asymptotycznego przyjmuje się grecką literę Θ (*theta*). Jeśli funkcja f należy do tej samej kategorii pod względem tempa wzrostu, co funkcja g (czyli obie funkcje „rosną tak samo szybko"), to piszemy, że $f = \Theta(g)$. Formalnie rzecz biorąc, oznacza to, że istnieją liczby rzeczywiste dodatnie c_1, c_2, n_0 takie, że dla każdego $n > n_0$ spełnione są nierówności $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$. Innymi słowy, odpowiednio skalując funkcję f możemy nią ograniczyć funkcję g zarówno z dołu, jak i z góry.

Wracając do naszego przykładu, widzimy od razu, że takie ograniczenie jest niemożliwe, bo – jak zauważyliśmy wcześniej – funkcje f i g należą do różnych klas. Przykładowo, przyjmując $c_1 = 0.5$ oraz $c_2 = 2$ otrzymamy następujący wykres:



Na pierwszy rzut oka, wygląda na to, że funkcja g została tu skutecznie ograniczona z dołu i z góry dla $n > 2$, ale jeśli wyobrazimy sobie co będzie dla większych n (powyżej 10), to zauważymy, że warunek $c_1 \cdot f(n) \leq g(n)$ nie będzie już spełniony. Nawet jeśli wybierzemy mniejsze c_1, to ten warunek kiedyś w końcu (dla dostatecznie dużych n) przestanie być prawdziwy, przekonując nas o tym, że funkcja f rośnie istotnie szybciej od funkcji g.

W praktyce, zwykle wystarcza nam tylko jedno, górne ograniczenie, które zapisujemy za pomocą notacji „wielkiego O", tzn. piszemy, że $f = O(g)$, co oznacza, że funkcja f rośnie „nie szybciej" niż funkcja g, czyli – ściśle mówiąc – że istnieją liczby rzeczywiste dodatnie c, n_0, takie że dla każdego $n > n_0$ spełniona jest nierówność $c \cdot f(n) \leq g(n)$. Oczywiście, jeżeli zachodzi zarówno $f = O(g)$ oraz $g = O(f)$, oznacza to, że funkcje f i g spełniają tym samym warunek $f = \Theta(g)$.

Dla ilustracji, napiszmy program w języku Python rozwiązujący poniższe zadanie:

Zadanie: Wyznacz sumę S_n wszystkich liczb naturalnych od 1 do n.

Rozwiązanie 1:

```
def liczSume1(n):
        suma = 0
        for i in range(1,n+1):
                suma += i
        return suma
```

Rozwiązanie 2:

```
def liczSume2(n):
        return sum(range(1,n+1))
```

Spójrzmy na czasy wykonania tych funkcji:

wartości n w milionach	100	200	500	1000
czas wykonania `liczSume1` w sekundach	8,5	16,8	43,0	87,5
czas wykonania `liczSume2` w sekundach	5,4	11,8	26,9	54,1

Można zauważyć, że czas działania pierwszego programu rośnie liniowo ze względu na n (dla dwa razy większego n czas zwiększa się w przybliżeniu dwa razy), co sugeruje złożoność $O(n)$, czyli liniową. Do tego samego wniosku dojdziemy analizując sam kod programu (w którym liczba przebiegów pętli `for i...`, a więc i liczba dodawań, jest po prostu równa n).

W drugim rozwiązaniu bezwzględne czasy wykonania są wyraźnie mniejsze niż w pierwszym, jednak jego złożoność obliczeniowa jest taka sama, tj. liniowa (tu też dwukrotne zwiększenie n wydłuża dwukrotnie czas działania). Rozwiązanie 2 jest zatem przykładem tylko tzw. *mikrooptymalizacji*, czyli bardziej efektywnej implementacji algorytmu o tej samej klasie złożoności obliczeniowej[3].

Zadanie to można jednak rozwiązać za pomocą znacznie szybszego algorytmu:

```
def liczSume3(n):
        return n*(n+1)//2
```

[3] Ten przykład jest dość typowy dla programów w Pythonie – użycie gotowych funkcji bibliotecznych, takich jak „sum" zapewnia zwykle lepszą wydajność.

dla którego czas działania jest stały[4], czyli $O(1)$. W rozwiązaniu tym wykorzystujemy znany wzór na sumę pierwszych n wyrazów ciągu arytmetycznego.

Czy oprócz złożoności stałej i liniowej istnieją jeszcze inne? Oczywiście – aby się o tym przekonać, rozważmy inne zadanie:

Zadanie: Niech S_k będzie sumą wszystkich liczb naturalnych od 1 do k. Wyznacz sumę n kolejnych takich sum, czyli: $S_1 + S_2 + \ldots + S_n$.

Rozwiązanie 1:

```
def liczPodwojnaSume1(n):
        suma = 0
        for i in range(1,n+1):
                suma += liczSume1(i)
        return suma
```

Rozwiązanie 2:

```
def liczPodwojnaSume2(n):
        suma = 0
        s_k = 0
        for i in range(1,n+1):
                s_k += i
                suma += s_k
        return suma
```

Rozwiązanie 1 wykorzystuje funkcję `liczSume1` z pierwszego rozwiązania poprzedniego zadania. Funkcja ta jest wywoływana w pętli, która wykonuje się n razy. Ponieważ sama funkcja `liczSume1` również posiada pętlę, która także wykonuje się n razy (czyli mamy tu *de facto* dwie pętle zagnieżdżone jedna w drugiej), możemy oszacować całkowitą liczbę operacji dodawania na $n \cdot n$ (w przybliżeniu). Rozwiązanie 2 natomiast zawiera tylko jedną pętlę, w której jednocześnie liczymy zarówno bieżącą wartość sumy wynikowej, jak i potrzebną nam aktualnie wartość S_k (to ostatnie obliczenie robimy „sprytnie", zauważając, że aktualną wartość S_k możemy uzyskać po prostu dodając i do poprzedniej – a więc jednym dodawaniem, bez potrzeby użycia dodatkowej pętli).

Czas wykonania obu rozwiązań dla różnych n przedstawiono w poniższych tabelach:

wartości n w tysiącach	5	10	20	40
czas wykonania `liczPodwojnaSume1` w sekundach	0,9	4,2	16,8	68,4

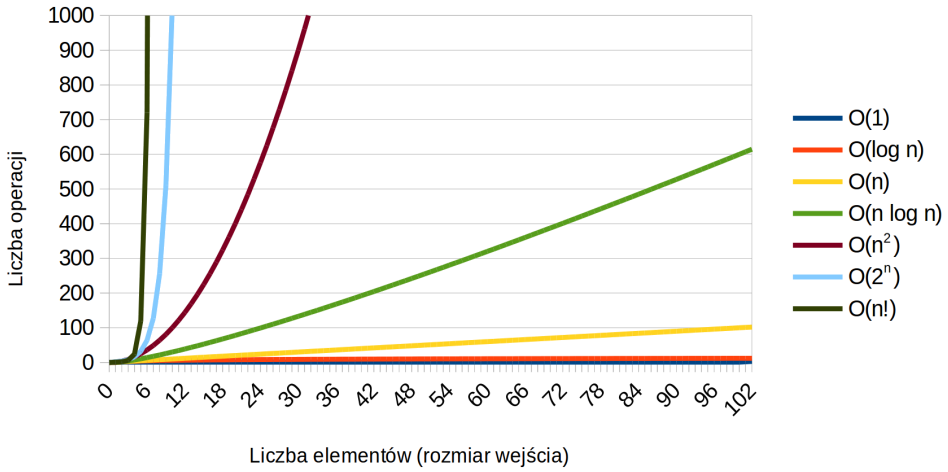
wartości n w milionach	5	10	20	40
czas wykonania `liczPodwojnaSume2` w sekundach	0,9	1,7	3,5	7,2

Można zauważyć, że czas wykonania implementacji `liczPodwojnaSume1` wynosi $O(n^2)$, czyli rośnie kwadratowo ze względu na n, a czas wykonania implementacji `liczPodwojnaSume2` wynosi $O(n)$, czyli rośnie

[4]Nie jest to tak do końca prawdą, co można zauważyć dla bardzo dużych n – lepszym oszacowaniem jest tutaj $O(\log^2 n)$.

liniowo ze względu na n. Z powyższych obserwacji wynika, że algorytmy o czasie $O(n^2)$ są istotnie „gorsze” niż algorytmy o czasie $O(n)$ (które z kolei są „gorsze” od algorytmów o czasie $O(1)$). Jest to szczególnie widoczne dla dużych n, podczas gdy dla małych może się zdarzyć, że będzie odwrotnie. Jednak, jak pamiętamy z wcześniejszej dyskusji, najbardziej interesuje nas „prawa część wykresu”, ponieważ to, jak szybko wykona się program dla n większych od pewnego granicznego n_0 decyduje ostatecznie o skuteczności danego rozwiązania.

Na poniższym rysunku pokazano wykresy zależności liczby operacji od rozmiaru danych wejściowych dla kilku różnych klas złożoności obliczeniowej:



Nie są to oczywiście wszystkie możliwe klasy złożoności, z jakimi możemy się spotkać. Algorytmy o złożoności $O(n^2), O(n^3), O(n^4)$, itd. określamy ogólnie jako algorytmy o złożoności wielomianowej. Znacząco wolniejsze są od nich algorytmy o złożoności wykładniczej $O(2^n)$, czy – jeszcze mniej wydajne – algorytmy o złożoności $O(n!)$[5]. Algorytmy z tych dwóch ostatnich klas są zwykle tak wolne, że ich wykonanie dla rozmiaru danych większego niż kilka-kilkanaście elementów jest niemożliwe w jakimkolwiek sensownym limicie czasowym.

Mówiąc o limitach, warto również pamiętać, że nasz program ma nie tylko ograniczony czas działania, ale również inne zasoby – w szczególności pamięć operacyjną. Moglibyśmy zatem mówić nie tylko o złożoności czasowej, ale również pamięciowej, używając analogicznych oszacowań asymptotycznych (notacja Θ, notacja wielkiego „O”). W niektórych zadaniach algorytmicznych największym wyzwaniem jest właśnie napisanie programu w tak „sprytny” sposób, aby nie przekraczał limitu dostępnej pamięci.

Na koniec, wspomnijmy jeszcze o tym, że sam rozmiar danych wejściowych nie zawsze decyduje jednoznacznie o czasie działania programu (liczbie wykonanych operacji, zużytej pamięci, itd.). Dwa zbiory danych o tym samym rozmiarze mogą nie być tak samo trudne dla naszego algorytmu. Prostym przykładem jest zadanie ustawienia w kolejności rosnącej ciągu liczb podanego na wejście (czyli *posortowanie* go). Liczby te mogą być podane w losowej kolejności, ale może się też tak zdarzyć, że będą już od razu prawidłowo posortowane. Nasz algorytm będzie wówczas miał dużo łatwiejsze zadanie (a w zasadzie – nie będzie miał w ogóle nic do roboty). Czy faktycznie będzie w stanie skorzystać z tego i podać wynik szybciej – to już zależy od konkretnego algorytmu. Podobnie, możemy napotkać takie „złośliwe” ułożenie danych, które może spowodować znaczące wydłużenie czasu działania programu. Oszacowanie klasy złożoności obliczeniowej algorytmu powinno się

[5]Wartość wyrażenia $n!$ (czyli „n silnia”) to $1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$ – jest to liczba wszystkich możliwych permutacji [23] (czyli ustawień w każdej możliwej kolejności) n-elementowego ciągu liczb.

zatem formułować osobno dla przypadku najtrudniejszego („pesymistycznego”), średniego i najłatwiejszego („optymistycznego”). Oczywiście, poszczególne algorytmy mogą być tutaj bardziej lub mniej wrażliwe na układ danych wejściowych (a zatem „wrażliwość” algorytmu jest jego kolejną właściwością, którą moglibyśmy rozważać).

Reasumując, temat złożoności obliczeniowej, który tu zarysowaliśmy jest w gruncie rzeczy bardzo obszerny i zdecydowanie wykracza poza ramy naszej książki. Omawiając rozwiązania poszczególnych zadań będziemy jednak często wracać do tych zagadnień, ponieważ umiejętność określania i porównywania klas złożoności algorytmów stanowi ważną umiejętność każdego programisty. Czytelników zainteresowanych pogłębieniem tego tematu odsyłamy do literatury [4][7][2], a my przechodzimy już do praktyki. Przed nami pierwszy dział naszego zbioru zadań, w którym rozpoczniemy od prostych problemów dla Skrzatów, do rozwiązania w środowisku Scratch.

Centrum
Mistrzostwa
Informatycznego

Dział I

Proste zadania z duszkami – animacje

4 (I) – Wprowadzenie

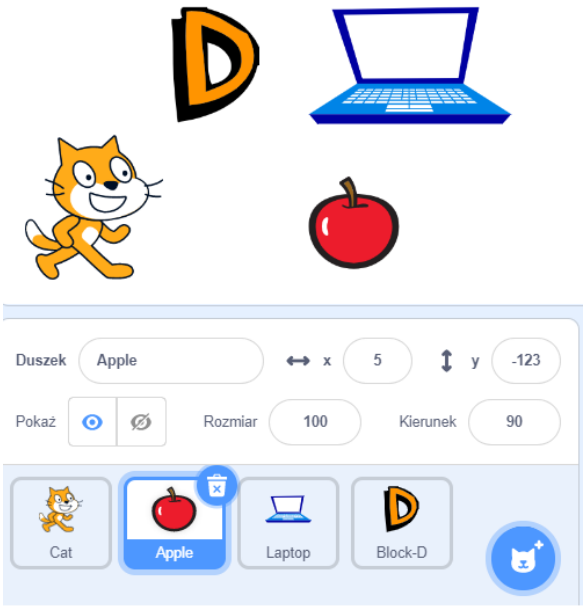
Autor: Marcin Markowski, Politechnika Wrocławska

Dział ten zawiera zadania z *duszkami*, do rozwiązywania w środowisku Scratch. Zadania są wprawdzie przeznaczone dla najmłodszych programistów, ale i ci bardziej doświadczeni znajdą w nich interesujące wyzwania. Przede wszystkim, zadania zapewnią Wam doskonałą zabawę i pokażą jak łatwo w środowisku Scratch sterować duszkami tworząc animacje, a nawet jak napisać prostą grę. A zatem – do dzieła!

4.1 Duszki i plansza

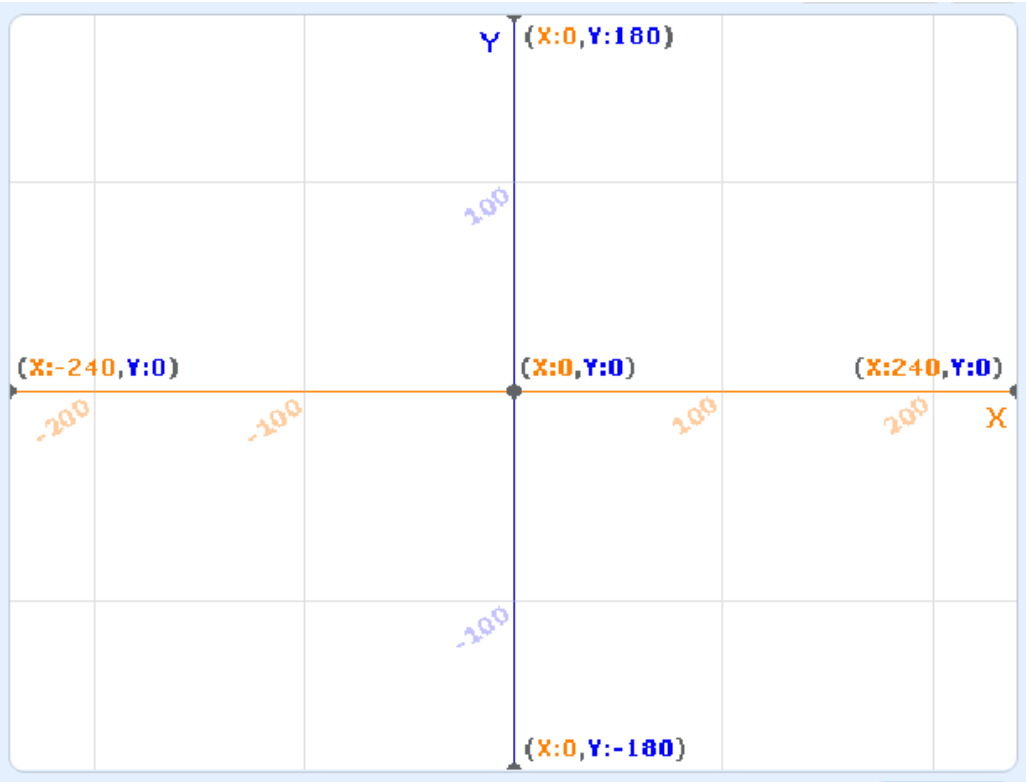
Jednym z podstawowych elementów w środowisku Scratch jest *duszek* (ang. *sprite*). Duszek to pewien kształt, postać lub przedmiot, który może poruszać się po ekranie, spotykać z innymi duszkami i wykonywać różne czynności, a także obliczenia. Za wygląd duszka odpowiada jego *kostium*, a jego zachowaniem steruje odpowiedni *program* (inaczej: *skrypt*). Programowanie w Scratchu jest najczęściej właśnie programowaniem duszków (pisaniem skryptów sterujących duszkami).

W swoich programach możesz używać wielu różnych gotowych kształtów duszków, możesz nawet tworzyć swoje własne, korzystając z wbudowanego edytora graficznego (ale to jest poza tematyką tej książki). Na rysunku poniżej możesz zobaczyć cztery różne duszki – kota, jabłko, laptopa i literę D:



Duszki można wybierać po kliknięciu przycisku z symbolem duszka (biało-niebieskie kółko w prawym dolnym rogu na rysunku). Duszki można przestawiać po ekranie „ręcznie”, tj. za pomocą myszy lub wpisując ich współrzędne x i y. Dla każdego duszka możesz również określić nazwę, rozmiar (domyślnie 100%), kierunek w którym będzie się poruszał (0 – do góry, 90 – w prawo, 180 – w dół, -90 – w lewo) oraz to, czy jest widoczny na ekranie.

Obszar ekranu, po którym będą poruszały się duszki (czyli *scena* albo *plansza*) ma ograniczone rozmiary – a dokładnie wymiary 480 (punktów, pikseli) w poziomie oraz 360 w pionie. Punkt na środku planszy ma współrzędne (0, 0) – tak jak pokazano na rysunku poniżej.



Jeżeli umieścisz duszka w miejscu o współrzędnych wykraczających poza współrzędne planszy, to nie będzie on widoczny (albo będzie widoczna tylko jego część). Rozwiązując nasze zadania należy starać się, aby duszek nie oddalał się poza widoczną część planszy (chyba, że treść zadania mówi co innego).

4.2 Rodzaje bloków w środowisku Scratch

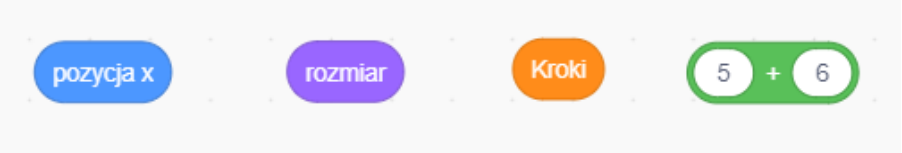
W środowisku Scratch, z lewej strony ekranu umieszczone są *bloki*, reprezentujące instrukcje i wyrażenia, z których będziesz budować swoje skrypty. Bloki są pogrupowane tematycznie w sekcje `Ruch`, `Wygląd`, `Dźwięk`, `Zdarzenia`, itp.

Omówimy teraz bloki szczególnie przydatne do rozwiązywania zawartych w tym dziale zadań z duszkami. Przede wszystkim, zwróć uwagę na różne kształty bloków. Możemy wśród nich wyróżnić:

- bloki przypominające puzzle – z „wcięciami” i „wypustkami” umożliwiającymi ich łączenie:

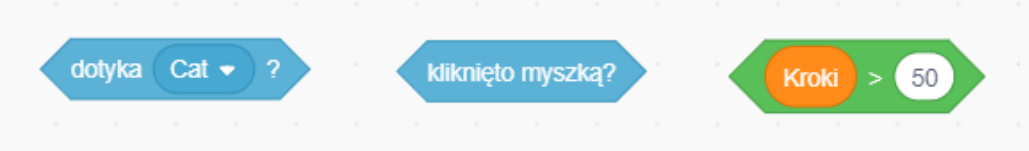
Takie bloki reprezentują różnego rodzaju *instrukcje* i polecenia – na przykład *pętlę* (`powtórz 10 razy`), polecenie zmiany ustawienia duszka (`obrót o 15 stopni`) albo uruchomienie programu po spełnieniu warunku: „`kiedy kliknięto` ikonkę *zielonej flagi*”;

- bloki owalne („prostokąty z zaokrąglonymi rogami”):



Bloki tego typu reprezentują *wartości* (liczby lub napisy) i *zmienne*, na przykład pozycję `x` (duszka), rozmiar (duszka), wartość zmiennej `Kroki` albo wartość wyrażenia 5+6;

- bloki sześciokątne:



Takie bloki reprezentują *warunki* – na przykład czy duszek dotyka innego duszka (o nazwie `Cat`), czy kliknięto myszką albo czy wartość zmiennej `Kroki` jest większa od 50.

4.3 Zmienne

W programowaniu (czy to w Scratchu, czy w innym języku) bardzo ważnym elementem są *zmienne*. Zmienna to miejsce w pamięci komputera, w którym możemy przechowywać dane – na przykład liczby (5) albo napisy, czyli łańcuchy znaków („Ala ma kota”). Możemy sobie wyobrazić zmienną jako pojemnik przechowujący wartość (daną), podczas gdy nazwa zmiennej jest nazwą tego pojemnika. W Twoim programie możesz mieć wiele zmiennych (pojemników z danymi) do przechowywania i wykorzystywania różnych danych.

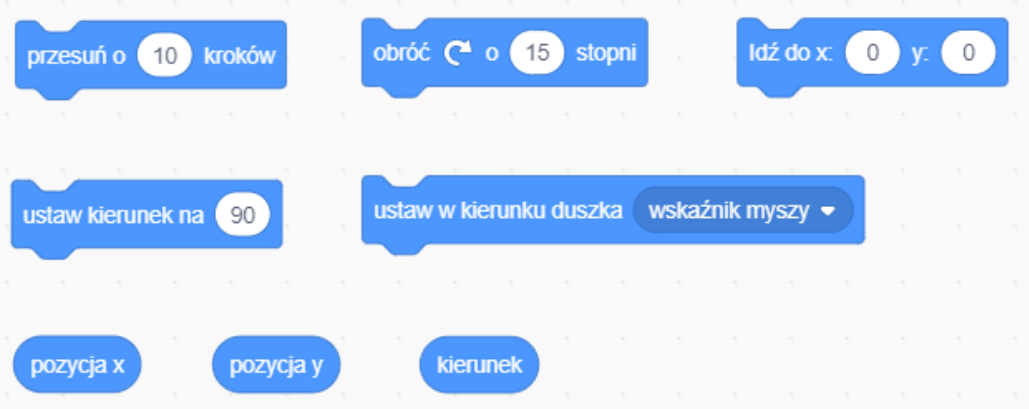
W Scratchu możesz tworzyć i używać zmiennych korzystając z bloków zgrupowanych w sekcji `Zmienne` – bloki w kolorze pomarańczowym. Aby utworzyć zmienną, kliknij przycisk `Utwórz zmienną`, a następnie podaj jej nazwę. W Scratchu zmienna może być dostępna tylko dla jednego duszka, albo dla wszystkich duszków. Aby użyć w programie danych przechowywanych przez zmienną, użyj owalnego bloku z jej nazwą (na przykład, na wcześniejszym rysunku pokazaliśmy jak sprawdzić czy wartość zmiennej `Kroki` jest większa od 50). Pozostałe pomarańczowe bloki pozwalają na zmianę wartości przechowywanej przez zmienną oraz na pokazanie jej na ekranie lub ukrycie. Blok do zmiany wartości zmiennej pozwala wybrać z rozwijanej listy tę zmienną, której wartość chcesz zmodyfikować.

4.4 Przegląd bloków

W zadaniach w tym dziale będziemy korzystali z różnych bloków pozwalających sterować duszkami. Jak wspomniano wyżej, bloki te są pogrupowane w sekcje oznaczone różnymi kolorami. Każda sekcja zawiera bloki służące do różnego rodzaju czynności – na przykład umożliwiające poruszanie duszkiem (sekcja `Ruch`), obsługę różnych zdarzeń, takich jak naciśnięcie przycisku na klawiaturze (`Zdarzenia`) czy wykonywanie

obliczeń (sekcja `Wyrażenia`). Sekcje są widoczne po lewej stronie ekranu i oznaczone kolorowymi kółkami. Przyjrzyjmy się teraz sekcjom i blokom przydatnym do rozwiązywania zadań z tego działu.

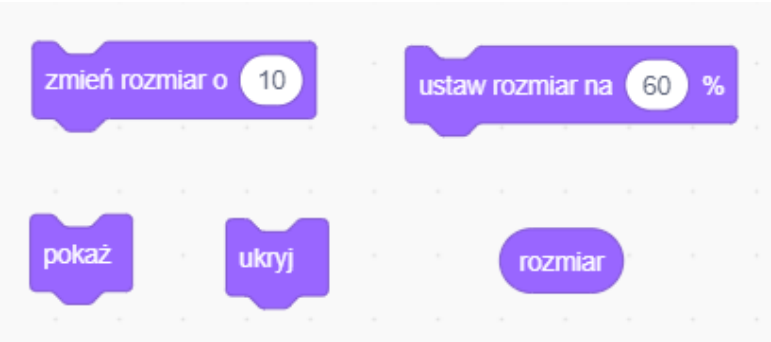
Bloki w sekcji `Ruch` pozwalają poruszać duszkiem (przesuwać go po ekranie, obracać, zmieniać jego pozycję) oraz odczytywać jego pozycję. Kilka bloków tej sekcji pokazano poniżej:



Pozycję duszka (współrzędne `x` i `y`) oraz kierunek, w którym jest ustawiony, możemy odczytać za pomocą owalnych bloków widocznych na dole rysunku. Pamiętaj, że te bloki nie są tak właściwie zmiennymi – nie możesz bezpośrednio zmieniać ich wartości. Natomiast wartości te zmienią się, gdy zmienisz ustawienie duszka za pomocą któregoś z bloków poleceń (np. `idź do x: ... y: ...` albo `obróć o 15 stopni`).

Tu jedna ważna uwaga – współrzędne duszka (`x` i `y`) oznaczają współrzędne jego *środka*! Na przykład kot widoczny po uruchomieniu Scratcha ma rozmiary około 100 x 100 (pikseli/jednostek), a jego współrzędne określają środek postaci.

Sekcja `Wygląd` pozwala zmienić wygląd duszka, ale też wygląd tła. W naszych zadaniach nie będziemy jednak za bardzo wykorzystywać tych możliwości środowiska Scratch. Z tej sekcji przydatne nam będą jedynie bloki umożliwiające zmianę rozmiaru duszka, ukrycie i pokazanie duszka na ekranie oraz odczytanie aktualnego rozmiaru duszka.



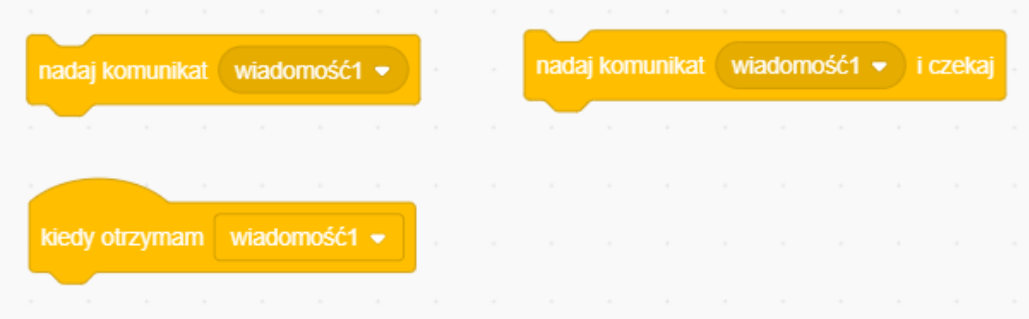
Sekcja `Dźwięk` umożliwia duszkowi wydawanie dźwięków. Nie będziemy jej omawiać, ponieważ nie wykorzystujemy tych bloków w naszych zadaniach.

Bloki z kolejnej sekcji (`Zdarzenia`) będą natomiast bardzo istotne w naszych skryptach. Dwa z tych bloków będą wykorzystywane do rozpoczynania skryptów sterujących różnymi duszkami w Twoim programie albo skryptów wykonujących obliczenia.

Wszystkie nasze programy będą się rozpoczynały po kliknięciu ikonki zielonej flagi, znajdującej się nad

planszą po lewej stronie – dlatego blokiem, od którego będzie się zaczynał każdy Twój skrypt będzie blok `kiedy kliknięto zieloną flagę`. Zwróć uwagę, że blok ma charakterystyczny kształt – zaokrąglony na górze – co oznacza, że zawsze będzie pierwszym blokiem w skrypcie (nie da się go „podczepić" do żadnego innego bloku).

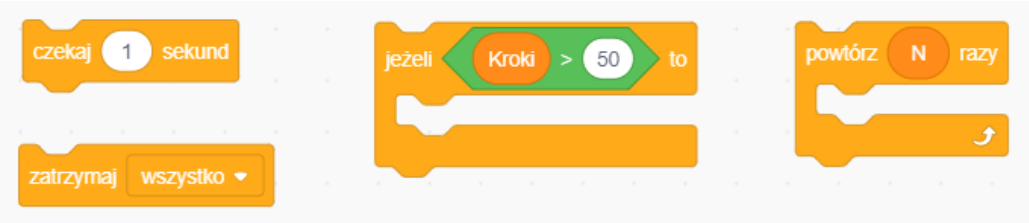
Innym sposobem rozpoczęcia wykonywania skryptu jest wykorzystanie bloku `kiedy otrzymam wiadomość` (zwróć uwagę, że blok ma identyczny kształt jak blok z zieloną flagą, czyli musi być pierwszy w sekwencji poleceń).



Blok ten służy do rozpoczynania wykonania skryptów po otrzymaniu komunikatu. Czasami, przed uruchomieniem wszystkich duszków będziemy chcieli wykonać jakieś wstępne czynności (np. ustalić wartości zmiennych albo rozstawić duszki na ekranie) – wówczas dobrą praktyką jest uruchomienie za pomocą zielonej flagi skryptu wykonującego te czynności, a dopiero potem uruchomienie wszystkich duszków poprzez rozesłanie im odpowiedniego komunikatu. Nasze programy będą się zatem rozpoczynały po kliknięciu zielonej flagi, co uaktywni skrypt jednego lub kilku duszków, a następnie aktywne duszki będą mogły wysyłać komunikaty uruchamiające skrypty kolejnych duszków. Do wysyłania komunikatów służą bloki `nadaj komunikat` (przy czym nazwę komunikatu do wysłania, np. `wiadomość 1`, wybieramy z listy). Nadanie komunikatu, np. `wiadomość 1`, aktywuje wszystkie skrypty rozpoczynające się od bloku `kiedy otrzymam wiadomość 1`. Po wysłaniu komunikatu, skrypt który go wysłał wykonuje się dalej (wykonują się kolejne instrukcje), a aktywowany wiadomością inny skrypt zaczyna się wykonywać równolegle („żyć swoim życiem"). Trochę inaczej działa blok `nadaj komunikat i czekaj` – po wysłaniu komunikatu skrypt wysyłający zaczeka, aż aktywowany skrypt skończy się wykonywać i dopiero wtedy będzie kontynuował swoje działanie.

Warto w tym miejscu podkreślić, że komunikaty to bardzo użyteczny mechanizm, m.in. dlatego że pozwalają nam wymusić odpowiednią kolejność wykonania skryptów. Oczywiście, jest rzeczą bardzo wygodną, że możemy mieć wiele duszków i każdy z nich może „żyć swoim życiem" zaczynając wykonywać swój skrypt po kliknięciu zielonej flagi, jednak czasami prowadzi to do irytujących (i trudnych do znalezienia!) błędów. Przykładowo, jeśli jeden duszek ustawia np. wartość zmiennej, a drugi w tym samym momencie próbuje ją odczytać, to nie wiemy tak do końca, która z tych operacji wystąpi pierwsza. Jeśli pierwszy duszek wygra ten „wyścig", to zmienna odczytana przez drugiego duszka będzie już mieć wartość prawidłowo ustawioną. Jeśli jednak drugi duszek będzie szybszy i odczyta wartość zmiennej przed jej ustawieniem, to... mamy problem. Odpowiednie użycie mechanizmu komunikatów pozwala nam na uniknięcie takich kłopotów.

Sekcja `Kontrola` zawiera bardzo ważne bloki pozwalające na kontrolowanie przebiegu programu – instrukcje pętli (`powtórz`, `powtarzaj...aż`, `zawsze`), instrukcje warunkowe (`jeżeli...to`, `jeżeli...to... – w przeciwnym razie`) oraz instrukcje pozwalające wstrzymać na pewien czas wykonywanie programu (`czekaj...sekund`, `czekaj aż`). W tej sekcji zawarty jest również bardzo przydatny blok, który powoduje zakończenie działania skryptu duszka albo całego programu – blok `zatrzymaj`. Ponadto sekcja ta zawiera bloki do sterowania *klonami* (tak, tak, duszki można klonować!), ale nie będziemy wykorzystywali tej funkcjonalności w naszych zadaniach.



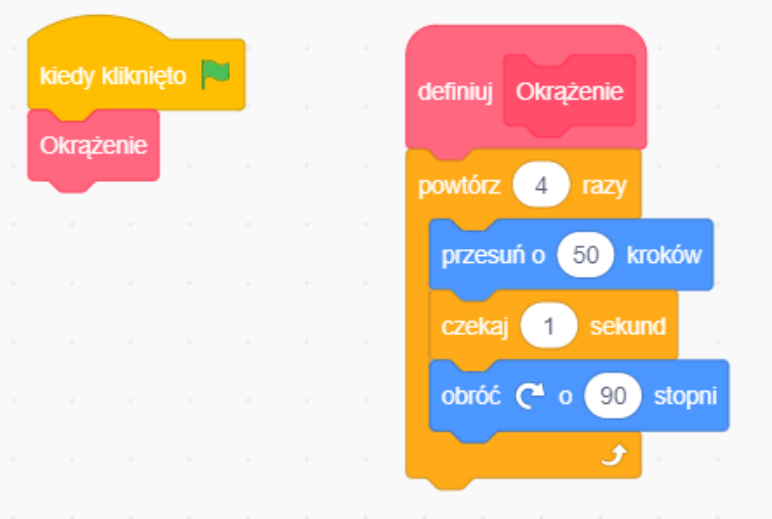
Bloki z sekcji `Czujniki` pozwalają na wykrywanie pewnych sytuacji, na przykład kolizji duszków (gdy duszki się dotykają), czy kolizji duszka z krawędzią ekranu. Z tej sekcji używać będziemy dwóch bloków – `dotyka` (innego duszka lub krawędzi ekranu) oraz `dotyka koloru`.

Inne bloki tej sekcji (nie używane w zadaniach w tym zbiorze) pozwalają m.in. na odczytywanie pozycji myszy, dialog z użytkownikiem (zadawanie pytań i wczytywanie odpowiedzi), czy ustalanie daty i godziny.

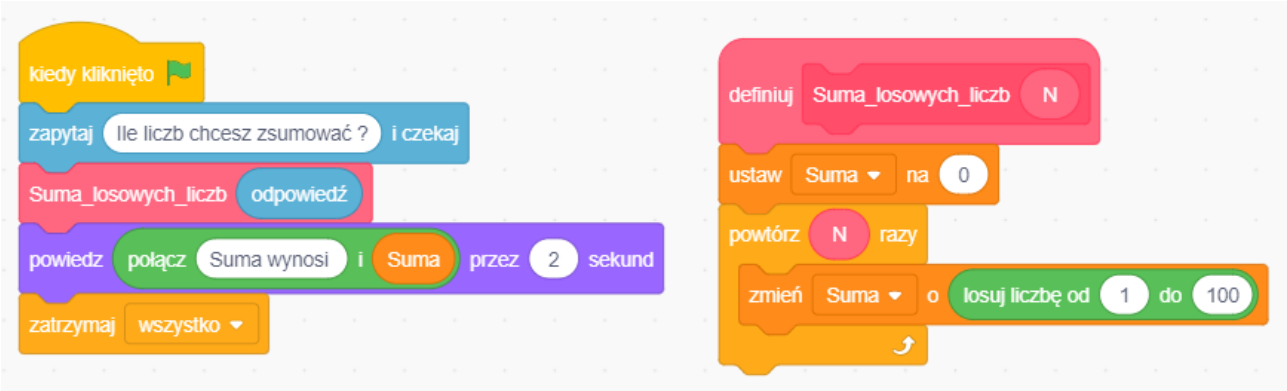
Sekcja `Wyrażenia` zawiera bloki pozwalające wykonywać obliczenia, sprawdzać warunki (większy, mniejszy, równy) i operować na łańcuchach. Dokładniejszy opis bloków tej sekcji znajdziesz w kolejnym dziale tej książki.

Ostatnia sekcja `Moje bloki` pozwala na tworzenie własnych dodatkowych bloków. Własne bloki są odpowiednikiem *funkcji* (procedur, podprogramów) obecnych w innych językach programowania. Jeżeli Twój program staje się bardzo długi, warto podzielić go na podprogramy – czyli własne bloki. Dzięki ich użyciu, główny program staje się czytelniejszy i – co ważne w Scratchu – łatwiej go zmieścić na pojedynczym ekranie. Dopóki nie utworzysz własnych bloków, sekcja ta będzie zawierała jedynie przycisk `Utwórz blok`. Tworząc blok pamiętaj, że warto nadać mu intuicyjną nazwę!

Na kolejnym rysunku pokazano sposób tworzenia i wykorzystywania bloków. Blok `Okrążenie` (po prawej) spowoduje pokonanie przez duszka trasy po kwadracie – czterokrotne przesunięcie o 50 kroków i obrót o 90 stopni. Blok ten składa się ze standardowych bloków, omówionych wcześniej w tym rozdziale. Zdefiniowanego bloku możesz używać w skryptach tak, jak każdego innego, co pokazano z lewej strony rysunku.



Tworząc nasz własny blok, mamy możliwość zdefiniowania *parametrów*, które będą do niego przekazywane w momencie jego użycia (podobnie jak w innych językach programowania możemy przekazywać *argumenty* do funkcji). Na poniższym przykładzie widzimy (po prawej) blok o nazwie `Suma_losowych_liczb` ze zdefiniowanym jednym parametrem `N`. Celem tego bloku jest obliczenie sumy wylosowanych liczb. Wartością przekazywaną do bloku jest `N` – informacja ile liczb ma być wylosowanych i zsumowanych. Po lewej stronie rysunku widzimy krótki program wykorzystujący zdefiniowany blok. Program otrzymuje liczbę od użytkownika (w zmiennej `odpowiedź`) i uruchamia blok `Suma_losowych_liczb` podając tę wartość jako parametr.



Zauważ, że dzięki użyciu własnego bloku, główny program jest krótszy, bardziej czytelny i łatwiej go analizować. Poza tym, nowego bloku możesz użyć w wielu miejscach Twojego programu, dzięki czemu unikniesz powtarzania takich samych sekwencji bloków.

4.5 Uwagi końcowe

Zanim przejdziesz do rozwiązywania zadań zawartych w tym dziale, pamiętaj o jeszcze jednej ważnej rzeczy – w naszych zadaniach, początkowe położenia duszków (i początkowe wartości ich parametrów) nie mogą być ustawiane przez Twoje skrypty! Testując swoje rozwiązania ustawiaj je na ekranie za pomocą myszy lub wpisując współrzędne w okienkach.

Wymóg ten związany jest z możliwością automatycznego sprawdzania poprawności programu przez spraw-

dzarkę. Podczas testów, sprawdzarka sama ustawi duszki w odpowiednich miejscach (może także ustawić im inne parametry – kierunek, widoczność, czy rozmiar), „kliknie” zieloną flagę, zaczeka określoną ilość czasu i sprawdzi, czy po wykonaniu programu duszki znalazły się w prawidłowych miejscach (i – ewentualnie – czy ich pozostałe parametry są zgodne z oczekiwanymi). Inicjalizacja parametrów i położenia duszków przez skrypt, „popsuje szyki” sprawdzarce, a wynik testowania będzie zapewne negatywny. Pamiętaj również, że sprawdzarka podczas ustawiania/odczytywania parametrów posługuje się *nazwami duszków*, które muszą w związku z tym ściśle odpowiadać wymogom treści zadania.

5 (I) – Zbijak – Skrzaty

Autor: Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (II edycja – liga algorytmiczna)
Język programowania: Scratch

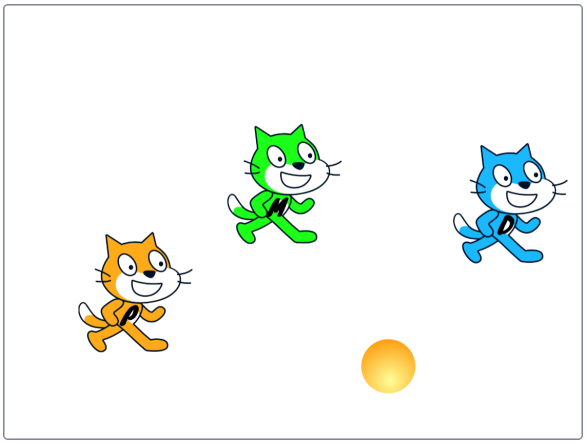
5.1 Treść zadania

Nasze trzy znajome koty: Procek, Mruczek i Drapek postanowiły dziś zagrać razem w zbijaka. Procek przyniósł w tym celu swoją specjalną, magiczną piłkę, którą wystarczy tylko raz rzucić w stronę ściany, żeby zaczęła odbijać się od kolejnych ścian bez końca.

Piłka została rzucona i zabawa się rozpoczęła, ale żeby uniknąć nieporozumień, zostałeś poproszony o pełnienie roli sędziego. Twoim zadaniem jest sprawdzenie w jakiej kolejności koty zostały zbite. Zbicie następuje oczywiście w momencie dotknięcia kota przez piłkę, która – co ciekawe – nie odbija się przy tym, tylko przelatuje przez niego na wylot nie zmieniając kierunku (piłka jest w końcu magiczna!).

Zadanie

Stwórz nowy projekt Scratch. Zmień nazwę duszka na `Procek` i utwórz jeszcze dwa takie same duszki, nadając im nazwy `Mruczek` i `Drapek`. Ostatni duszek, którego jeszcze potrzebujesz, to magiczna piłka Procka (piłka, to po angielsku *Ball*, więc wybierz duszka `Ball` i – uwaga – nie zmieniaj jego nazwy). Duszki możesz ustawić ręcznie, za pomocą myszy, w dowolnych miejscach planszy – na przykład tak, jak na rysunku 5.1.



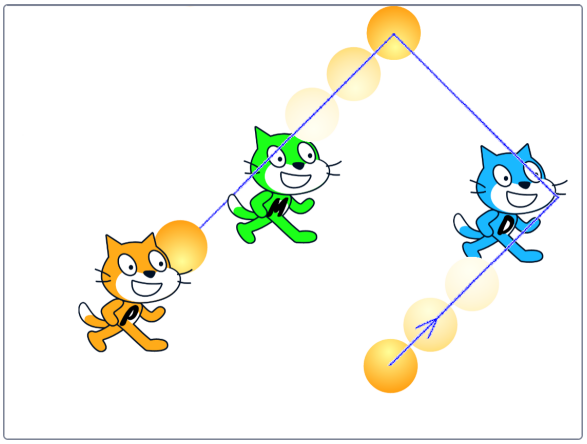
Rysunek 5.1: Przykładowe początkowe rozmieszczenie duszków na planszy

Zadanie polega na napisaniu programu, który po naciśnięciu zielonej flagi wprawi w ruch piłkę i będzie sprawdzał kto został przez nią zbity. Pamiętaj, że:

- Na początku programu piłka zawsze powinna wystartować w kierunku 45° (czyli w prawo i do góry), a następnie odbijać się od kolejnych ścian pod tym samym kątem (Rys. 5.2);
- Należy zadbać, aby odbicie nastąpiło w momencie dotknięcia krawędzi planszy (piłka nie powinna np.

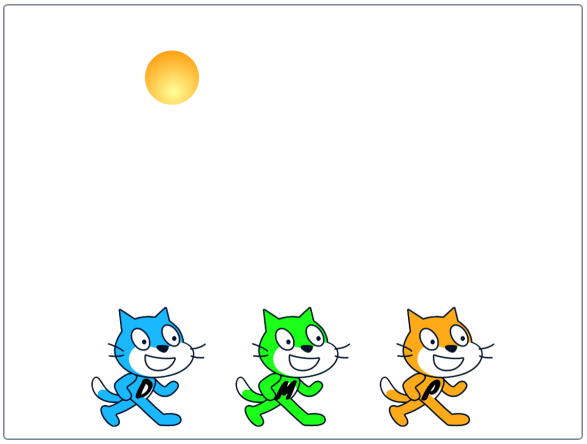
„zagłębiać się w ścianie”);

- Piłka powinna poruszać się nie wolniej niż 5 kroków i nie szybciej niż 10 kroków w każdym ruchu.



Rysunek 5.2: Tor lotu piłki dla przykładu z rysunku 5.1. Uwaga: Twój program nie musi go rysować – tu został pokazany tylko dla ilustracji

Po zbiciu ostatniego kota, piłka powinna się zatrzymać. Twój program powinien umieścić ją w dowolnym miejscu w górnej połowie planszy, a koty ustawić w dolnej połowie planszy w kolejności, w jakiej zostały zbite (od lewej do prawej). Następnie program powinien się zakończyć. Przykładowy wygląd planszy po zakończeniu programu pokazano na rysunku 5.3.



Rysunek 5.3: Przykładowy wygląd planszy po zakończeniu programu

Uwagi

Kolory duszków nie mają znaczenia. Ważne jest jednak, aby nie zmieniać ich parametrów (w szczególności kształtu i rozmiaru). Należy wykorzystać wyłącznie duszki `Cat` (dla Procka, Mruczka i Drapka) oraz `Ball` (dla piłki) w domyślnym rozmiarze 100, nadając im odpowiednie nazwy, zgodnie z treścią zadania.

5.2 Rozwiązanie

Rozwiązując zadanie należy uwzględnić kilka elementów:

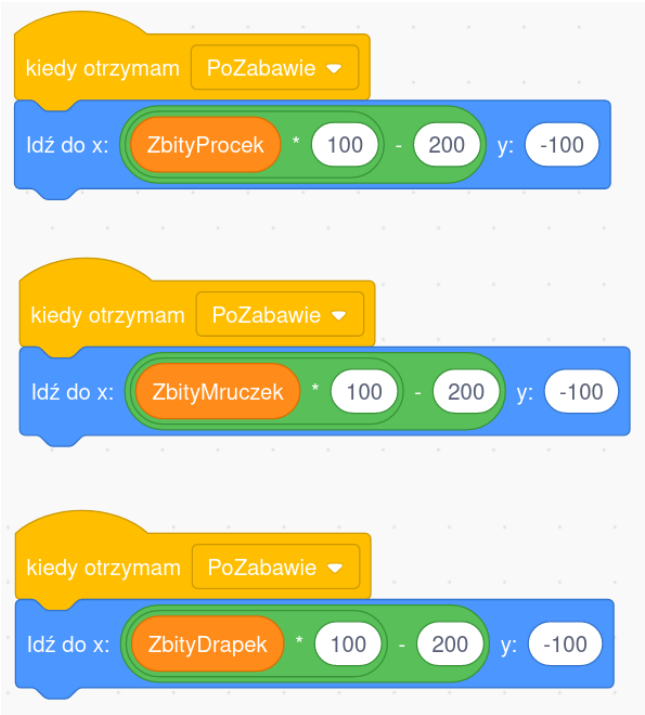
- Zapamiętać kolejność zbić i przerwać program, gdy wszystkie kocury zostaną już zbite;
- Odbijać piłkę od ściany (krawędzi ekranu), gdy do niej doleci;
- Na koniec ustawić koty we właściwej kolejności – bardzo ważne aby zrobić to dopiero po zakończeniu ruchu piłki!

W przykładowym rozwiązaniu użyliśmy kilku zmiennych. Zmienna `IleZbitych` (ustawiana na zero na początku programu) oznacza liczbę zbitych kotów, a zmienne `Zbity...` – określą w jakiej kolejności zbito kocury (zmienne te również zostają wyzerowane na początku programu). Program będzie się wykonywał, dopóki wartość zmiennej `IleZbitych` nie będzie równa 3. Gdy piłka dotknie któregoś nie dotkniętego wcześniej kota (ważne żeby nie był już wcześniej zbity, bo ignorujemy zbicie drugi, trzeci i kolejne razy), wówczas zwiększamy wartość tej zmiennej, następnie ustawiamy wartość odpowiedniej zmiennej zbitego kota.

W głównej pętli programu powinniśmy również poruszać piłką (blokiem `przesuń o ... kroków`) oraz odbijać piłkę od ściany – do tego najlepiej posłuży nam blok `jeżeli na brzegu, odbij się`.

Co ważne, wszystkie powyższe działania wykonują się w skrypcie duszka-piłki (`Ball`). Natomiast na koniec programu trzeba poinformować koty, aby ustawiły się w odpowiedniej kolejności. Po zatrzymaniu piłki,

wyślemy wszystkim kotom komunikat – ustawcie się w odpowiedniej kolejności! Używamy do tego bloku `nadaj komunikat`. W naszym przykładowym rozwiązaniu nadaliśmy temu komunikatowi nazwę: `PoZabawie`. Skrypty duszków-kotów, które uruchamiają się po jego otrzymaniu wyglądają następująco:



Ustawienie kotów w odpowiedniej kolejności można oczywiście oprogramować na różne sposoby, na przykład korzystając z instrukcji warunkowych (`jeżeli`). Ale chyba najszybszym i najbardziej eleganckim sposobem będzie wyliczenie ich pozycji x na podstawie kolejności zbicia – tak jak to pokazaliśmy na rysunku powyżej.

Pełne przykładowe rozwiązanie znajdziesz w pliku:

./zadania/01_Proste_zadania_z_duszkami/Zbijak/solution.sb3.

6 (I) – Zebra – Skrzaty

Autor: Marcin Markowski, Politechnika Wrocławska
Kategoria: Skrzaty (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

6.1 Treść zadania

Przechodzenie przez jezdnię zawsze jest wyzwaniem. Po pierwsze, trzeba znaleźć przejście dla pieszych, czyli zebrę. Następnie, uważnie się rozejrzeć i upewnić czy nie zbliża się jakiś pojazd. Dopiero gdy jest bezpiecznie można przeprawić się na druga stronę. Procek postanowił podjąć wyzwanie dodatkowe (oczywiście nie zapominając o bezpieczeństwie) i policzyć przez ile pasów przejdzie. Pasy zostały namalowane na jezdni czerwoną farbą. Farba miejscami już się starła, przez co niektóre pasy są częściowo niewidoczne. Procek postanowił, że jeśli pas na jego trasie jest niewidoczny, to nie będzie go liczył. Po zakończeniu przeprawy przez jezdnię Procek rośnie (w końcu umie już sam bezpiecznie przejść przez jezdnię) – zwiększa swój rozmiar o tyle jednostek, przez ile pasów przeszedł.

Zadanie

Stwórz nowy projekt Scratch i zmień nazwę duszka na `Procek`. Utwórz dodatkowo 9 duszków, które będą reprezentowały pasy na jezdni. Należy wybrać duszki typu `Line` (*Line* po angielsku oznacza „linia”). Kolor pasów powinien być czerwony (*Kolor: 0, Nasycenie: 100, Jasność: 100*) – taki jest domyślny kolor duszka `Line`. Duszki-pasy należy nazwać: `Pas1`, `Pas2`, `Pas3`, `Pas4`, `Pas5`, `Pas6`, `Pas7`, `Pas8`, `Pas9`. Rozmiar pasów należy pozostawić bez zmian (100), rozmiar `Procka` ustawić na 20, a kierunek na 90 (głową do góry).

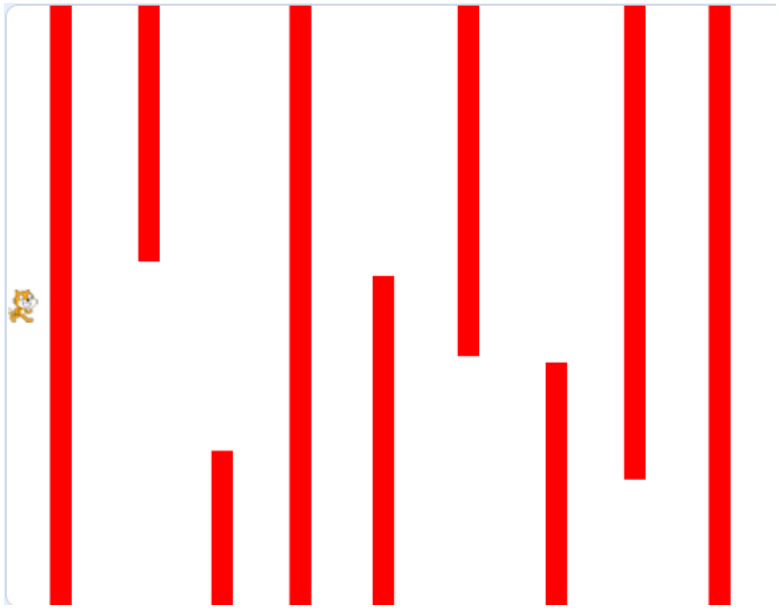
Pasy należy ustawić ręcznie za pomocą myszy. Mogą być ustawione wyłącznie pionowo (kierunek 0), odległości w poziomie (współrzędna x) pomiędzy nimi muszą wynosić co najmniej 50 jednostek, a położenie w pionie może być dowolne (część pasów jest „starta”, więc nie będą przecinały trasy `Procka`). `Procek` wchodzi na pasy z punktu przy lewym brzegu planszy (nie dotykając brzegu). Przykład rozmieszczenia pasów i `Procka` pokazano na rysunku 6.1.

Zadanie polega na napisaniu programu, który po naciśnięciu zielonej flagi przeprowadzi `Procka` przez jezdnię i policzy przez ile pasów przeszedł `Procek`.

Pamiętaj przy tym, że:

- `Procek` porusza się tylko poziomo – od lewej do prawej strony planszy. Gdy dotrze do prawej strony planszy, program powinien się zakończyć;
- `Procek` „przechodzi” przez pas, wtedy gdy dotknie go jakąkolwiek częścią kociego ciała. Jeżeli pas jest „starty” w miejscu przejścia `Procka`, to oczywiście nie jest uwzględniany w wyniku końcowym;
- Po przejściu po zebrze `Procek` zwiększa swój rozmiar o tyle jednostek, przez ile pasów przeszedł.

Dla przykładu, w przypadku pokazanym na rysunku 6.1, `Procek` przejdzie po 6 pasach i na koniec zwiększy swój rozmiar o 6 (osiągając końcowy rozmiar 26).



Rysunek 6.1: Przykładowe rozmieszczenie początkowe Procka i pasów

Uwagi

`Procek` jest małym kotkiem, więc porusza się małymi krokami – na pewno nie da rady przejść „nad pasem” w jednym kroku.

W Twoim skrypcie `Procek` może poruszać się wyłącznie poziomo (kierunek 90).

Pamiętaj, że ani początkowe położenie `Procka` ani położenie i kierunek pasów nie mogą być ustawiane przez Twoje skrypty. Podczas testowania musisz robić to ręcznie, np. za pomocą myszy.

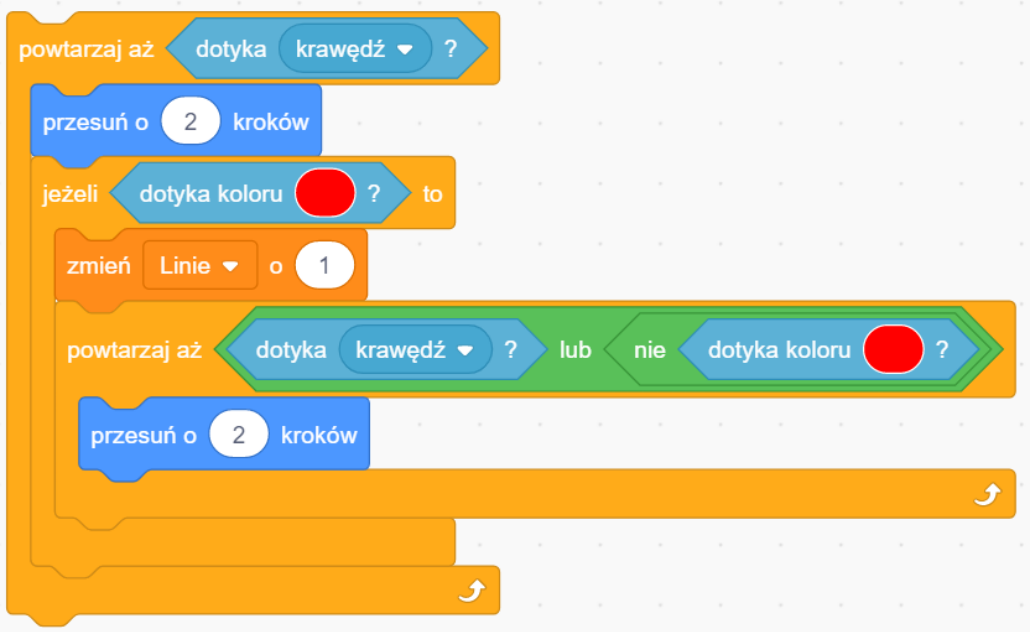
6.2 Rozwiązanie

Rozwiązując zadanie należy zwrócić uwagę na kilka elementów:

- `Procek` musi przejść poziomo od lewej do prawej strony ekranu, a gdy dotrze do prawej krawędzi, wówczas program powinien się zakończyć;
- `Procek` nie może się poruszać zbyt szybko, a dokładnie – zbyt dużymi krokami. W przeciwnym wypadku mógłby „przeskoczyć" pas i nie wykryć, że nad nim przeszedł;
- Należy zliczyć pasy, przez które przeszedł `Procek`, przy czym każdy dotknięty pas musi być policzony dokładnie jeden raz. Potrzebna będzie zmienna do zliczania dotkniętych pasów oraz sposób ukończenia przejścia nad dotkniętym pasem.

Pozycja początkowa `Procka` to lewa strona ekranu, na dowolnej wysokości. Ustaw go tam za pomocą myszy – pozycji początkowej nie powinien ustawiać program. Pamiętaj też, aby `Procek` nie dotykał lewej krawędzi ekranu. Dlaczego? Ponieważ program musi wykryć dotknięcie prawej krawędzi, a będziemy to robić za pomocą bloku sprawdzającego dotykanie do dowolnej krawędzi.

Zakładając, że `Procek` stoi z lewej strony ekranu, jego przejście do prawej strony i zatrzymanie przy prawej krawędzi możemy zaprogramować za pomocą bloku `powtarzaj aż ... dotyka krawędź`, który będzie główną pętlą programu.



W każdej iteracji `Procek` wykona dwa kroki – to wystarczająco mało, aby nie przeskoczył nad pasem (pasy mają szerokość około 15 kroków). Możesz poeksperymentować z większą liczbą kroków w każdej iteracji – `Procek` będzie poruszał się szybciej.

Pętla skończy się, gdy `Procek` dotknie krawędzi. Zwróć uwagę, że chodzi o którąkolwiek krawędź (również lewą, dolną i górną), dlatego `Procek` nie może dotykać żadnej z nich przed rozpoczęciem programu!

Drugi problem do rozwiązania to prawidłowe zliczanie pasów. Gdy `Procek` stanie na pasie (dotknie duszka `Pas`), należy zwiększyć wartość zmiennej zliczającej dotknięte pasy (zmienna `Linie`) na rysunku. Należy również „przeprowadzić" `Procka` do końca pasa i to tak, aby już więcej go nie policzyć. Dotknięcie pasa wykryjemy

czujnikiem `dotyka koloru`. W takiej sytuacji zwiększamy liczbę dotkniętych pasów o 1. Następnie przesuwamy `Procka` w prawo, tak długo jak długo dotyka pasa (koloru). Musimy również pamiętać o zatrzymaniu go, gdy dotrze do krawędzi ekranu – stąd drugi warunek związany z dotykaniem krawędzi.

Na koniec, gdy `Procek` dotrze do prawej krawędzi ekranu, pozostaje zwiększyć jego rozmiar za pomocą odpowiedniego bloku z sekcji `Wygląd`.

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/01_Proste_zadania_z_duszkami/Zebra/solution.sb3.

7 (I) – Do brzegu – Skrzaty

Autor: Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (II edycja – zawody algorytmiczne – etap regionalny)
Język programowania: Scratch

7.1 Treść zadania

Stało się – Procek został rzucony na głęboką wodę! I to dosłownie...
A było to tak: w ciepły majowy weekend Procek, Mruczek i Drapek pojechali na działkę do kolegi, który miał w ogrodzie spory basen. Po dobrym obiedzie i poobiedniej drzemce, postanowili pobawić się trochę na świeżym powietrzu. Ich zabawa, niestety, nie była chyba dobrze przemyślana, bo w jej efekcie Procek – nie wiedząc jak i kiedy – znalazł się nagle w samym środku basenu!

Koty na szczęście potrafią pływać, choć zwykle bardzo nie lubią wody. Procek chciał więc jak najszybciej dopłynąć do brzegu, a koledzy próbowali mu w tym pomóc. Mieli pod ręką pięć dużych piankowych „makaronów", które szybko wrzucili do wody, a Procek płynąc przed siebie trafił w końcu na jeden z nich i mógł wreszcie chwilę odetchnąć. Stwierdził, że będzie się czuł pewniej poruszając się wzdłuż makaronu, a w zasadzie – wzdłuż kilku kolejnych makaronów, bo pianki wrzucone w pośpiechu do wody krzyżowały się ze sobą w paru miejscach. Jego droga do brzegu – choć względnie bezpieczna – nie była więc najkrótsza, a do tego dość kręta. Aby nie pogubić się zupełnie, Procek postanowił liczyć kolejne zakręty w prawo i w lewo. Kiedy w końcu dotarł na brzeg, koledzy szczerze mu gratulowali pływackich umiejętności, a on sam – widząc, jak wyraźnie urósł w ich oczach – szybko zapomniał o strachu, a nawet zaproponował, aby nazajutrz wspólnie urządzić zawody pływackie.

Zadanie

Stwórz nowy projekt Scratch i zmień nazwę duszka na `Procek`. Utwórz jeszcze 5 nowych duszków, które będą reprezentować makarony. W tym celu należy wybrać duszka o nazwie `Line` (co po angielsku oznacza „linia" – i faktycznie – ten duszek to po prostu czerwona pozioma linia). Nazwij te duszki: `Makaron1`, `Makaron2`, `Makaron3`, `Makaron4` i `Makaron5`. Makarony mogą być ustawione tylko prostopadle do brzegów basenu (czyli planszy), więc mogą mieć kierunek tylko $0°$ lub $90°$.

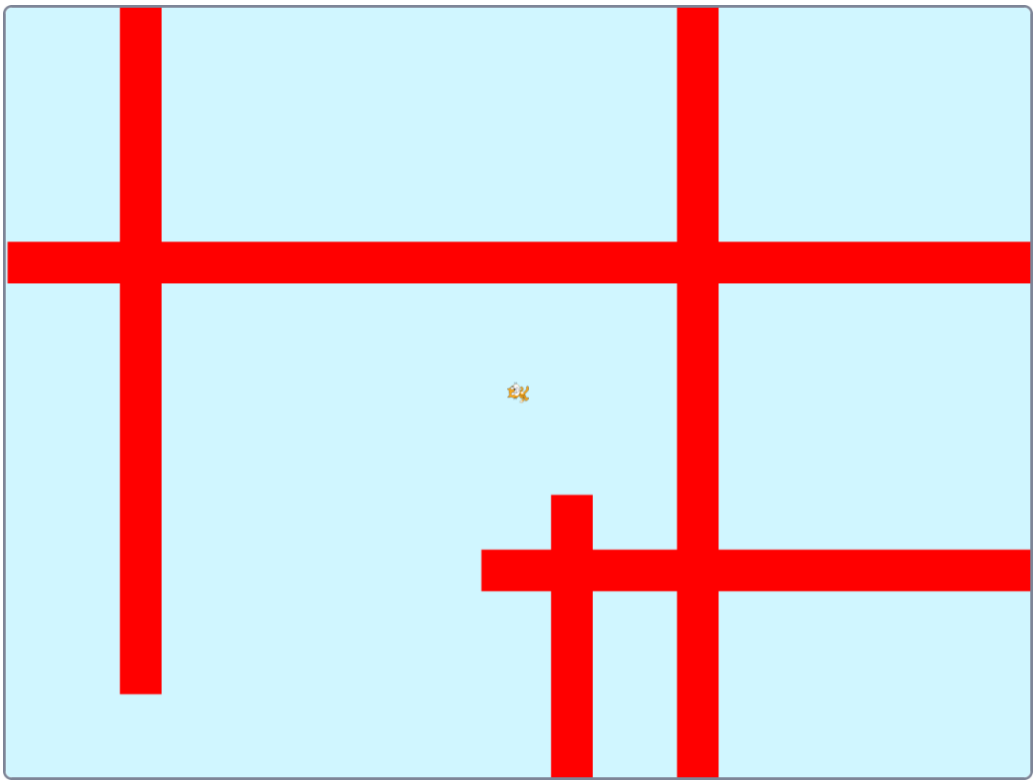
Rozmiar każdego makaronu należy zmienić na 150, a rozmiar `Procka` – na 10.

`Procka` i makarony należy ustawić ręcznie, za pomocą myszy, w dowolnych miejscach planszy – na przykład tak, jak na rysunku 7.1.

Zadanie polega na napisaniu programu, który po naciśnięciu zielonej flagi przeprowadzi `Procka` bezpiecznie do brzegu. Pamiętaj przy tym, że:

- Na początku programu `Procek` powinien zawsze płynąć do góry ekranu, a po napotkaniu pierwszego makaronu – skręcić w prawo.

- Po napotkaniu kolejnego makaronu (a także po dotarciu do końca bieżącego makaronu) `Procek` skręca ponownie i płynie dalej tak długo, aż nie dotrze do brzegu.
- Uwaga: może się też tak zdarzyć, że `Procek` od razu dopłynie do brzegu, nie napotykając żadnego makaronu po drodze.



Rysunek 7.1: Przykładowe początkowe rozmieszczenie duszków na planszy

Możesz założyć, że odległość między dowolnymi dwoma równoległymi makaronami, a także odległość między końcem makaronu, a najbliższym makaronem prostopadłym do niego, jest większa niż 20 kroków. Możesz także założyć, że dotarcie do brzegu jest zawsze możliwe (makarony nie tworzą zamkniętej pętli wokół `Procka`).

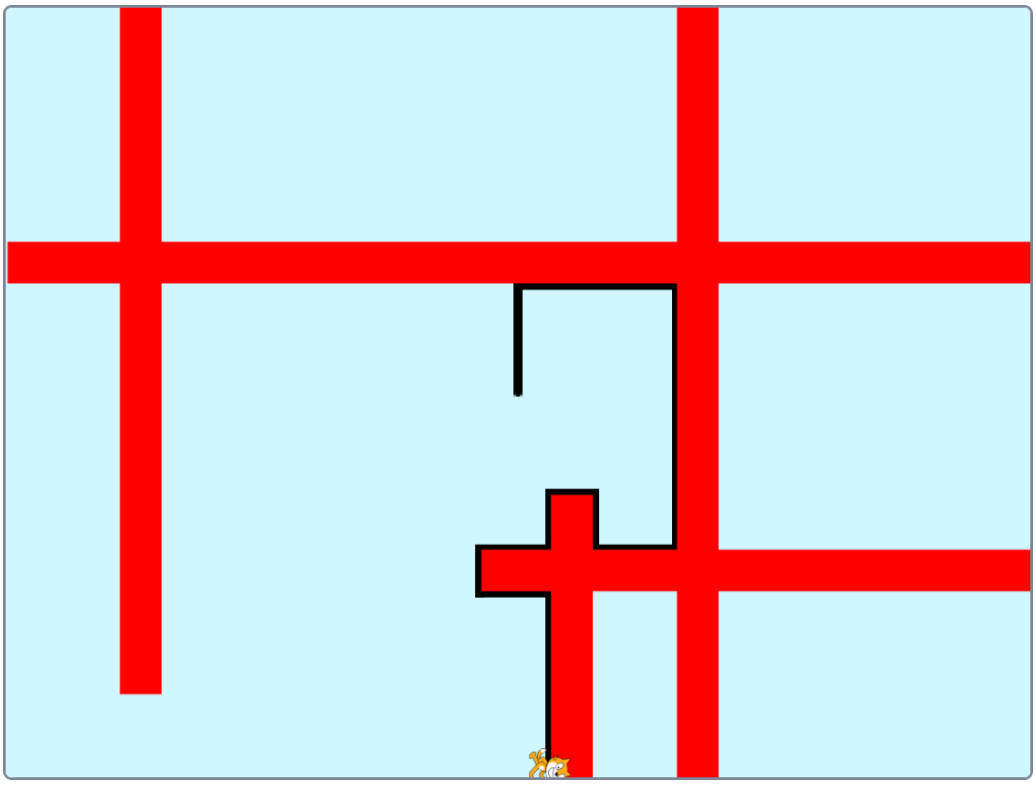
Po dotarciu do brzegu, `Procek` powinien „urosnąć w oczach kolegów”, czyli zmienić swój rozmiar o tyle, ile zakrętów wykonał po drodze, a następnie Twój program powinien się zakończyć. Rysunek 7.2 przedstawia stan końcowy planszy dla przykładu z rysunku 7.1. Pokazano tu także (na czarno) trasę pokonaną przez `Procka`. Twój program nie musi jej rysować (choć może), ale ważne, żeby po zakończeniu programu `Procek` znalazł się we właściwym miejscu przy brzegu. Powinien przy tym dotykać jednocześnie wody, makaronu i brzegu planszy. Ważne też, aby miał właściwy rozmiar. Tu `Procek` wykonał po kolei: 4 zakręty w prawo, 2 w lewo, 1 w prawo, 2 w lewo i 1 w prawo, więc jego rozmiar wzrósł o 10 i wynosi teraz 20.

Uwagi

Kolor makaronów, ani wody w basenie nie ma znaczenia. Nieistotne jest także, w jakim kierunku jest ustawiony `Procek` – powinien on oczywiście poruszać się po właściwej trasie, ale nie musi (choć może) obracać się przy każdym zakręcie.

Ważne jest natomiast, aby nie zmieniać wyglądu i kształtu duszków oraz by ustawić właściwie ich rozmiar oraz nazwy, zgodnie z treścią zadania. Pamiętaj też, że ani początkowe położenie `Procka` ani położenie i kierunek

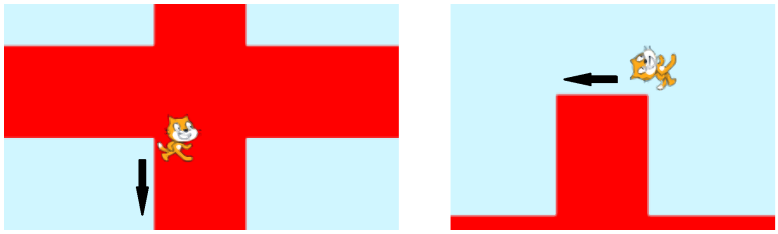
makaronów nie mogą być ustawiane przez Twoje skrypty. Podczas testowania musisz robić to ręcznie, np. za pomocą myszy.



Rysunek 7.2: Widok planszy po zakończeniu programu. Uwaga: trasę Procka pokazano tylko dla ilustracji – Twój program nie musi jej rysować

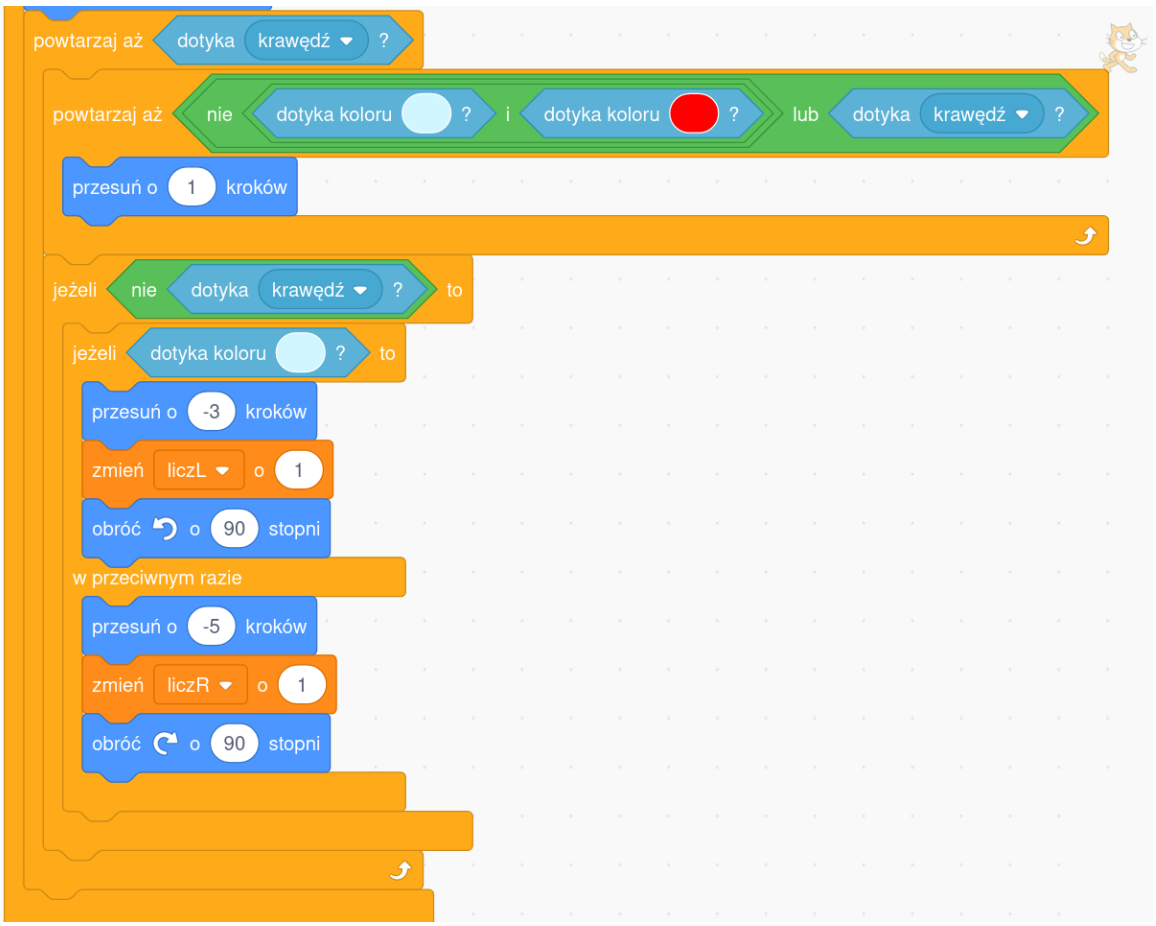
7.2 Rozwiązanie

Podstawowym wyzwaniem w zadaniu jest określenie, kiedy `Procek` powinien skręcić w prawo, a kiedy w lewo. Możemy zauważyć (w czym powinna pomóc analiza przykładu z treści zadania), że `Procek` będzie skręcał w prawo gdy napotka skrzyżowanie makaronów, a w lewo – gdy dotrze do końca makaronu. Jedną z możliwości rozwiązania (czyli wykrywania skrzyżowań i końców makaronów) jest prowadzenie `Procka` wzdłuż makaronu tak, aby stale dotykał zarówno makaronu, jak i wody. Jeżeli poruszając się w taki sposób `Procek` przestanie dotykać wody, to znaczy że trafił na skrzyżowanie makaronów (wspiął się na makaron), zatem czeka go skręt w prawo. Jeżeli przestanie dotykać makaronu, to znaczy się dotarł do jego końca i musi skręcić w lewo. Obie sytuacje przedstawiono na rysunku:



W przypadku wykrycia jednej z powyższych sytuacji, powinniśmy cofnąć `Procka` o kilka kroków (tak, aby ponownie dotykał i wody i makaronu), obrócić go o 90 stopni w odpowiednią stronę i (oczywiście!) zwiększyć licznik skrętów.

Fragment proponowanego rozwiązania widoczny jest na rysunku poniżej:



Główna pętla wykonuje się, dopóki `Procek` nie dotrze do brzegu (krawędzi ekranu). Wewnątrz tej pętli `Procek` porusza się (po jednym kroku) tak długo, jak długo jednocześnie: dotyka koloru wody i koloru makaronu (oczywiście wciąż nie dotykając krawędzi – aby kot nie uciekł poza ekran). Pętla poruszająca kotem kończy się gdy:

- `Procek` dotarł do krawędzi – zatem nie będzie się już więcej poruszał,
  lub
- `Procek` dotyka koloru wody. To znaczy, że nie dotarł do krawędzi (wykluczyliśmy to wcześniejszą instrukcją warunkową – `jeżeli nie dotyka krawędzi`), ale przestał dotykać koloru makaronu (bo gdyby dotykał, to nie skończyłaby się wcześniejsza pętla poruszająca `Prockiem`) – `Procek` skręca w lewo,
  lub
- `Procek` nie dotyka koloru wody. To znaczy, że dotyka tylko koloru makaronu i skręca w prawo.

W naszym przykładowym rozwiązaniu zliczamy osobno zakręty w prawo i w lewo, chociaż nie jest to konieczne (moglibyśmy sumować je wspólnie w jednej zmiennej).

Ale zanim `Procek` zacznie pokonywać zakręty, musi dopłynąć do najbliższego makaronu. Od tego powinien zaczynać się Twój skrypt. Ale na pewno już wiesz jak to zrobić – płynąc w górę ekranu `Procek` albo dotrze do krawędzi (i koniec) albo... wpłynie na makaron, czyli przestanie dotykać koloru wody. A wtedy – skręca w prawo, tak jak na skrzyżowaniu makaronów.

Na koniec, gdy `Procek` dotrze do krawędzi ekranu, należy zwiększyć jego rozmiar, za pomocą odpowiedniej instrukcji.

Pełne przykładowe rozwiązanie znajdziesz w pliku:

./zadania/01_Proste_zadania_z_duszkami/Do_brzegu/solution.sb3.

8 (I) – Inwazja rekinów – Skrzaty

Autor: Marcin Markowski, Politechnika Wrocławska
Kategoria: Skrzaty (III edycja – zawody algorytmiczne – etap finałowy)
Język programowania: Scratch

8.1 Treść zadania

Co 10 lat armia rekinów (zza siedmiu mórz i siedmiu oceanów) próbuje dokonać inwazji na plażę Bitlandii. Kolejna inwazja spodziewana była za rok, jednak, nie wiedzieć czemu, rekiny pojawiły się już dziś i Strażnik Plaż Bitlandii nie jest w pełni przygotowany do obrony. Sprawdź, czy rekinom uda się dotrzeć do plaży!

Już dawno odkryto, że najlepszy sposób na zniechęcenie rekina do ataku polega na trafieniu go gumową piłką (głośne PLAF! odbiera rekinowi chęć do dalszego ataku). Trafiony rekin znika pod powierzchnią morza i przestaje być groźny. Strażnik posiada 3 armaty do wystrzeliwania piłek (wycelowane w różne obszary morza, niestety sam nie jest w stanie ich obracać) i 15 piłek (więcej planowano dostarczyć gdzieś za 11 miesięcy...).

Zadanie

Stwórz nowy projekt Scratch, usuń domyślnego kota i utwórz duszki:

- duszka-plażę: duszek `Line` (nazwij go `Plaza`), rozmiar 100, kierunek 0, położenie x=240, y=5. Plaża znajduje się po prawej stronie planszy.
- 9 duszków-rekinów: duszek `Shark 2`, koniecznie z domyślnym kostiumem (`Shark2-a`), rozmiar 30, kierunek 90. Rekiny nazwij `Rekin1`, `Rekin2`, ..., `Rekin9`. Rekiny należy rozmieścić (ręcznie – za pomocą myszy) na lewej połowie planszy. Pamiętaj – pozycje początkowe rekinów nie mogą być ustawiane przez Twój skrypt!
- duszka-piłkę: duszek `Ball`, pozostaw nazwę `Ball`, rozmiar 30. Duszek zostanie wykorzystany do oddania 15 strzałów.

Wieża Strażnika i wszystkie armaty znajdują się w punkcie o współrzędnych (x=230, y=0). Wieży, Strażnika ani armat nie należy rysować ani umieszczać na planszy – po prostu z tego punktu zawsze będzie startowała wystrzelona piłka (a my wyobrazimy sobie, że jest tam Strażnik z armatami). Piłka będzie wystrzeliwana kolejno w kierunkach -60, -90, -120 (tak są wycelowane armaty). W inwazji (i Twoim skrypcie) na zmianę następują tury Strażnika i rekinów, przy czym tura Strażnika jest pierwsza:

- **Tura Strażnika.** Strażnik strzela po kolei z każdej armaty, w takiej kolejności (w takich kierunkach) jak podano powyżej. Piłka leci do czasu napotkania rekina lub krawędzi planszy. Piłka powinna przemieszczać się po 10 kroków. Jeżeli piłka trafi rekina, to rekin znika z powierzchni morza i przestaje brać udział w inwazji (tylko znika – ale pozostaje w tym samym miejscu i nie zmienia już swojego położenia ani rozmiaru). Piłka oczywiście nie leci już wtedy dalej – należy ją ustawić w pozycji do kolejnego strzału. Jeżeli piłka nie trafi żadnego rekina i dotknie krawędzi – również kończy swój lot i należy ją ustawić w pozycji do kolejnego strzału.

- **Tura rekinów** Po oddaniu trzech strzałów (po jednym z każdej armaty), Strażnik musi ponownie załadować armaty. W tym czasie wszystkie wciąż biorące udział w inwazji rekiny rosną o 10 i przesuwają się poziomo (w prawo) o 60 jednostek.

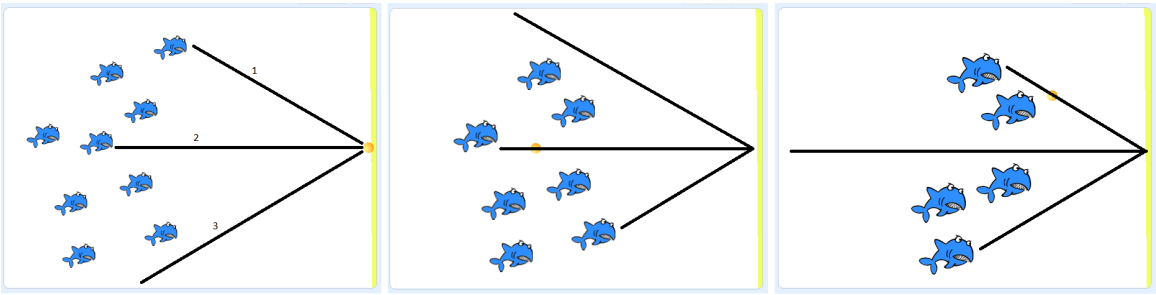
Jeśli Strażnik nie zdoła trafić wszystkich rekinów, pozostaje mu tylko bezsilnie patrzeć i czekać aż pozostałe rekiny dotrą do plaży. Po zakończeniu ostatniego (15-tego) strzału piłka znika (zostaje na miejscu, ale przestaje być widoczna), a w kolejnych turach Strażnika nic już się nie dzieje.

Inwazja kończy się w jednym z dwóch przypadków:

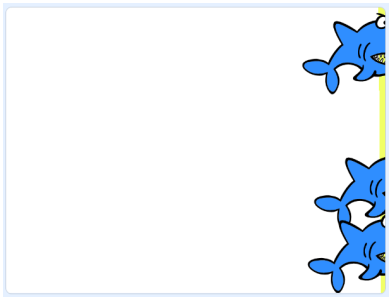
- po zakończeniu tury rekinów, jeśli którykolwiek z rekinów dotarł do plaży (niezależnie od tego, czy Strażnikowi pozostały jeszcze jakiekolwiek piłki);
- podczas tury Strażnika, w momencie gdy ostatni rekin zostanie trafiony piłką (i zniknie).

W obu przypadkach program powinien natychmiast się zakończyć.

Przykładowy przebieg inwazji przedstawiono na rysunku 8.1. Po prawej stronie widać kawałek plaży (żółta linia), wieża Strażnika znajduje się na środku plaży (tam gdzie piłka na obrazku pierwszym od lewej). Strażnik strzela z trzech armat – tory lotu piłek zaznaczono czarną linią, a kolejność strzałów (tylko na obrazku po lewej) kolejnymi liczbami. Trafione zostaną 2 rekiny (piłkami 1 i 2), które znikają (ale do końca skryptu nie zmieniają pozycji, kierunku ani rozmiaru). Następuje tura rekinów: 7 pozostałych rekinów rośnie i przesuwa się w prawo: ich nowe rozmiary i pozycje widać na środkowym obrazku. W kolejnej turze Strażnika (środkowy obrazek) trafione zostają 2 rekiny. W kolejnej turze rekinów pozostałe 5 rekinów rośnie i się przesuwa – obrazek po prawej. Następnie Strażnik trafia kolejnych dwóch napastników. Kolejne dwa etapy będą wyglądały podobnie. Patrząc na rozmieszczenie rekinów można przypuszczać, że Strażnik trafi pozostałe 3 (w końcu zostało mu jeszcze 6 piłek) i inwazja się nie uda. Z kolei przykład udanej inwazji przedstawiono na rysunku 8.2.



Rysunek 8.1: Przykładowy przebieg trzech pierwszych etapów inwazji (tory lotu piłki pokazano tylko dla ilustracji – nie powinieneś ich rysować w swoim programie)



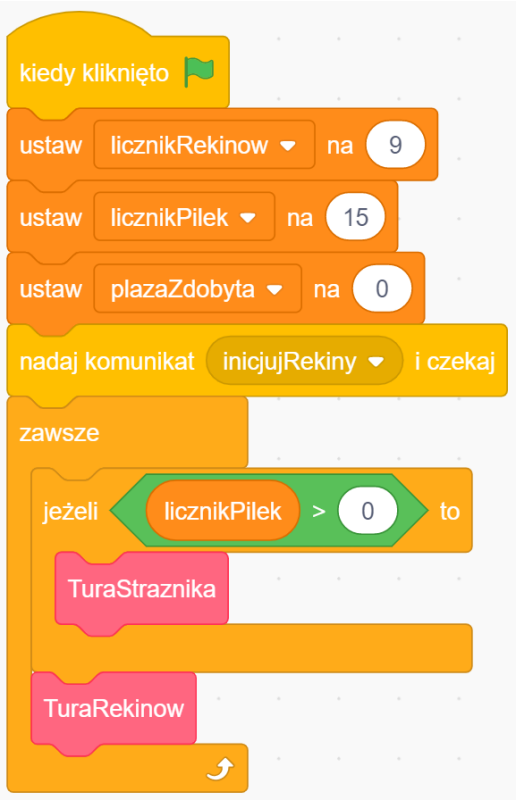
Rysunek 8.2: Przykład udanej inwazji

Uwagi

Pamiętaj, że początkowe położenia duszków-rekinów nie mogą być ustawiane przez Twoje skrypty. Podczas testowania musisz robić to ręcznie, np. za pomocą myszy. Rekiny nie mogą też – po trafieniu piłką – zmieniać swojego położenia, kierunku ani rozmiaru.

8.2 Rozwiązanie

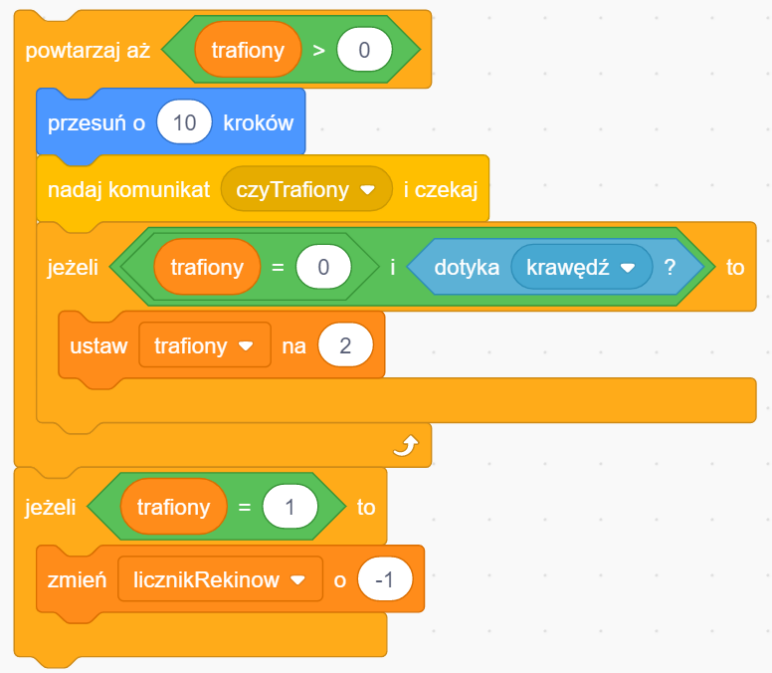
Zadanie wygląda na skomplikowane, no i do tego rekiny... ale nie będzie tak strasznie! Na co szczególnie należy zwrócić uwagę, to fakt, że wiele rzeczy będzie się działo równolegle, na przykład równolegle będą się poruszały wszystkie rekiny. Pojawi się zatem konieczność synchronizacji skryptów różnych duszków. Dobrym sposobem na jej realizację jest wykorzystanie systemu komunikatów przesyłanych pomiędzy duszkami. W rozwiązaniu należy oprogramować zarówno duszka-piłkę, jak i rekiny. Ponieważ program będzie dość długi (zwłaszcza w porównaniu z poprzednimi zadaniami w tym dziale), warto też go podzielić na moduły – dlatego w przykładowym rozwiązaniu zdefiniowaliśmy własne bloki. Dzięki temu główny program (dla duszka-piłki) jest krótki i czytelny:



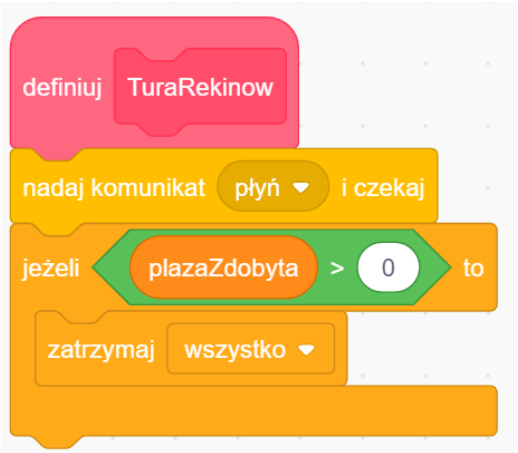
Rozpoczynamy od zainicjowania potrzebnych zmiennych (czyli nadania im początkowych wartości). Zauważ, że odpowiednie nazwy zmiennych pozwalają intuicyjnie rozumieć, do czego będą wykorzystane (np. zmienna `licznikPilek` przechowuje wartość oznaczającą liczbę piłek jakimi dysponuje strażnik). Następnie wysyłamy komunikat do rekinów – główny program poczeka, aż wszystkie duszki-rekiny wykonają odpowiednią procedurę i dopiero wtedy będzie kontynuował działanie (czyli wykonywał na przemian tury strażnika i rekinów).

Na następnym rysunku pokazano najważniejszą część bloku `TuraStraznika` (całość można obejrzeć w załączonym projekcie). Pokazany fragment opisuje ruch piłki i reakcje na jej kontakt z rekinami. W każdej iteracji piłka pokonuje 10 kroków (może ich być mniej – wówczas będzie wolniej się poruszała, może też być trochę więcej, ale nie za dużo – aby nie przeskoczyć rekina). Następnie sprawdzamy, czy któryś z rekinów nie został trafiony – a dokładniej, sprawdzają to same rekiny po otrzymaniu odpowiedniego komunikatu (za chwilę przeanalizujemy skrypt rekinów). Program duszka-piłki czeka na reakcję wszystkich rekinów i dopiero po tym jak zakończą one weryfikację, kontynuuje działanie, to jest sprawdza dwa warunki. Pierwszy warunek jest spełniony, gdy piłka nie trafiła żadnego rekina i doleciała do końca ekranu (dotyka krawędzi), drugi – gdy

któryś z rekinów został trafiony. Zmienna `trafiony` ma wartość równą 0 dopóki żaden z rekinów nie zostanie trafiony w bieżącym strzale, a wartość 1 – gdy piłka trafiła rekina.



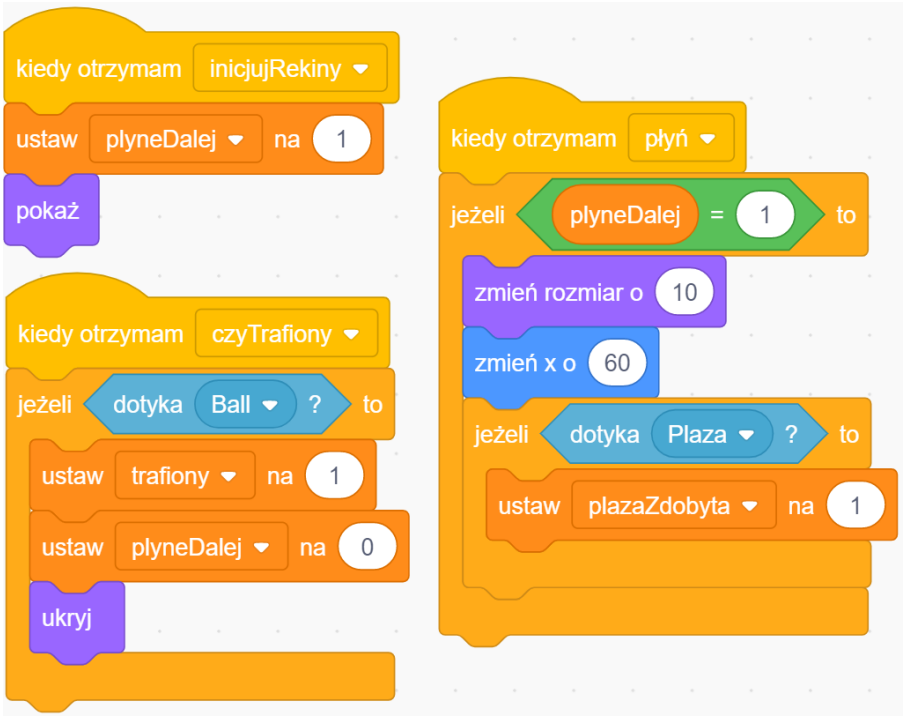
Tura rekinów jest nieco prostsza – wysyłany jest komunikat nakazujący rekinom poruszyć się i urosnąć, a gdy wszystkie to już zrobią, skrypt sprawdza warunek końca programu (dotarcie któregoś z rekinów do plaży):



Przyjrzyjmy się teraz skryptom rekinów (na następnej stronie). Rekiny muszą umiejętnie zareagować na trzy komunikaty przesyłane od duszka-piłki: `inicjujRekiny`, `czyTrafiony` oraz `płyń`.

Zmienna `plyneDalej` ma wartość równą 1 gdy rekin nie został jeszcze trafiony i równą 0, gdy po trafieniu przestaje brać udział w inwazji. Zauważ, że jest to prywatna zmienna rekina – to znaczy każdy rekin ma swoją własną zmienną `plyneDalej`, niezależną od zmiennych innych rekinów.

Inicjacja rekina polega na ustawieniu wartości tej zmiennej na 1 i pokazaniu rekina na ekranie. Komunikat `płyń` nakazuje rekinowi (nie trafionemu piłką) przesunąć się i zwiększyć rozmiar, następnie sprawdzić czy nie dotarł do plaży. Po otrzymaniu komunikatu `czyTrafiony`, rekin weryfikuje trafienie (kontakt z duszkiem-piłką), a w przypadku trafienia odpowiednio modyfikuje wartości zmiennych i chowa się („pod wodę”).



Pełne przykładowe rozwiązanie znajdziesz w pliku:

./zadania/01_Proste_zadania_z_duszkami/Inwazja_rekinow/solution.sb3.

9 (I) – Zabawy w labiryncie – Skrzaty

Autor: Magdalena Wachulec, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Skrzaty (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

9.1 Treść zadania

Duszek Bajtuś uwielbia spacery po labiryncie, a największą przyjemność sprawia mu odnajdywanie czerwonych drzwi, prowadzących do komnat pełnych łakoci. Zazwyczaj Bajtuś potrafi przenikać przez ściany, ale ten labirynt zrobiono z bardzo kłującego ostrokrzewu, więc duszek musi chodzić wyłącznie po wyznaczonych ścieżkach.

Duszek zapisuje wszystkie swoje kroki, nie tylko po to, by się nie zgubić, ale też, by wyznaczyć drogi dojścia do czerwonych drzwi. Pomóż Bajtusiowi sprawdzić, czy zapisane przez niego kroki doprowadzą go do wymarzonych słodyczy.

Zadanie

Pobierz projekt „template.sb3”:

./zadania/01_Proste_zadania_z_duszkami/Zabawy_w_labiryncie/template.sb3,
zawierający przykładowy labirynt oraz duszka Bajtusia. Znajdziesz w nim również przykładową listę poleceń sterujących ruchem Bajtusia (lista `DANE`) oraz pustą listę `WYNIKI`.

Celem zadania jest napisanie skryptu, który przeprowadzi Bajtusia przez labirynt zgodnie z kolejnymi poleceniami z listy `DANE` i sprawdzi czy dotrze on do czerwonego pola. Jeżeli tak – należy wpisać słowo TAK na listę `WYNIKI`, w przeciwnym przypadku należy tam wpisać słowo NIE. Obowiązują przy tym następujące zasady:

- Na liście `DANE` mogą wystąpić wyłącznie polecenia: LEWO, PRAWO, GÓRA, DÓŁ;
- Po zakończeniu skryptu na liście `WYNIKI` musi znaleźć się dokładnie jedno słowo (TAK lub NIE);
- Bajtuś może poruszać się wyłącznie po polach koloru szarego, żółtego lub czerwonego, jednorazowo pokonując 30 kroków w lewo, w prawo, w górę lub w dół;
- Jeżeli jakiś ruch jest niemożliwy (próba wejścia na zielone pole) – Bajtuś pozostaje na swoim miejscu i wykonuje kolejny ruch z listy;
- Jeżeli Bajtuś dotrze do czerwonego pola przed końcem listy poleceń, należy go tam pozostawić, wpisać na listę `WYNIKI` słowo TAK i zakończyć działanie programu;
- Bajtuś wyrusza zawsze z lewego górnego pola labiryntu (koloru żółtego).

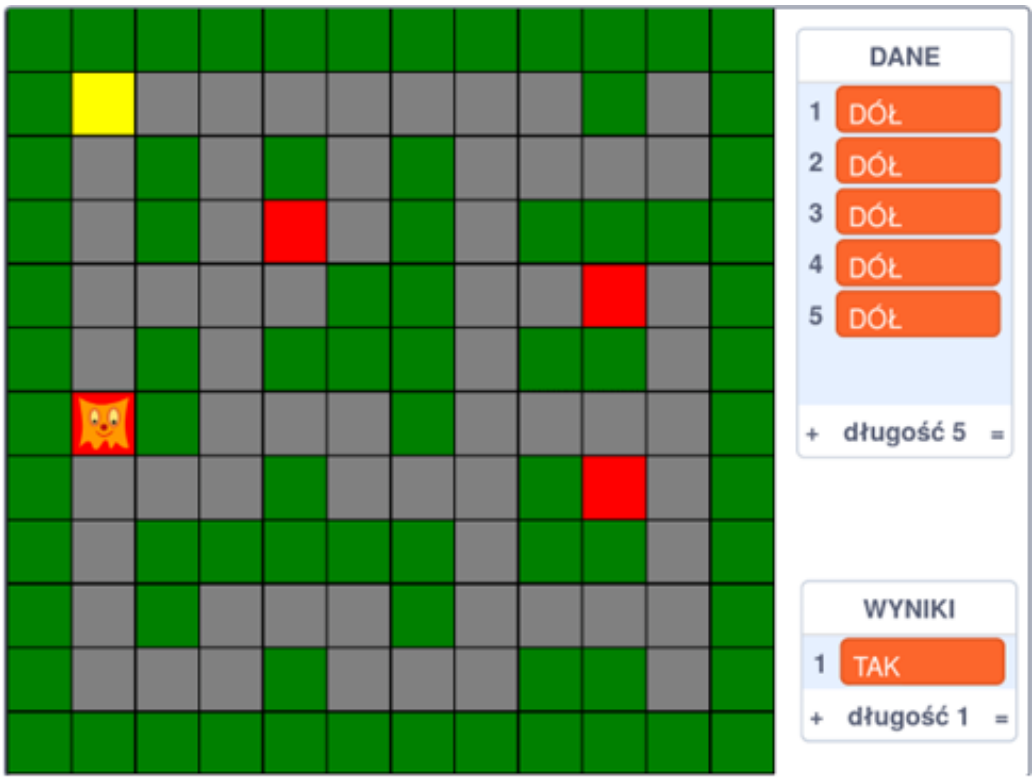
Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Program warto przetestować także na innych ciągach poleceń oraz na innych planszach labiryntu, ale położenie początkowe duszka jest zawsze takie samo (lewe górne pole labiryntu).

Jakiekolwiek modyfikacje duszka Bajtusia (zmiana nazwy, kostiumu, rozmiaru, widoczności, itd.) lub labiryntu są niedozwolone.

Przykład

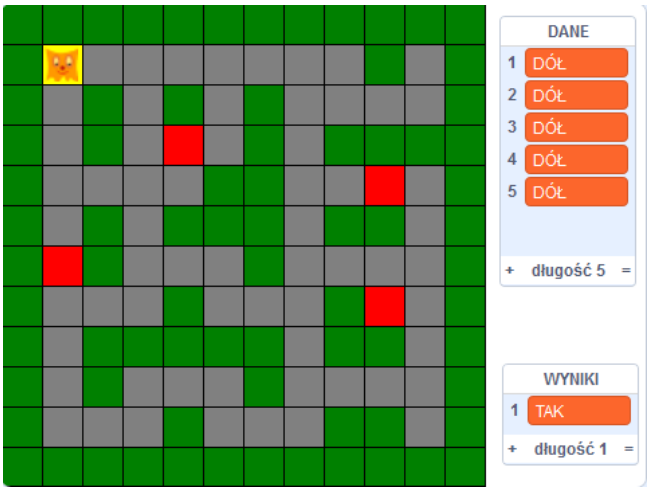
Na rysunku poniżej pokazano przykładowy wynik działania programu. Bajtuś wyruszył z żółtego pola w lewym górnym rogu labiryntu i po przejściu pięciu pól w dół znalazł się na czerwonym polu, więc na listę `WYNIKI` skrypt wpisał słowo TAK.



9.2 Rozwiązanie

W zadaniu należy zasymulować ruchy duszka na załączonej planszy labiryntu. Zadanie to różni się nieco od wcześniejszych – i to z dwóch powodów. Po pierwsze, mamy tu gotową planszę wyznaczającą sposób poruszania się duszka (co jest sporym ułatwieniem, bo nie trzeba jej samodzielnie tworzyć). Po drugie, występuje tu pewien bardzo ważny element środowiska Scratch, o którym nie mówiliśmy dotąd, a mianowicie *lista*. W kolejnym dziale omówimy dokładniej to pojęcie, zobaczymy też jak bardzo jest ono istotne i przydatne dla programisty, tu natomiast poprzestaniemy na stwierdzeniu, że ruchy duszka mamy po prostu zapisane w „liście wejściowej" o nazwie `DANE`, odsyłając zainteresowanego czytelnika do rozwiązania przykładowego (i – oczywiście – do kolejnych działów, w których listy i inne *struktury danych* będą już pojawiać się regularnie...).

Wracając do naszego problemu: jak pamiętamy, Bajtuś mógł poruszać się wyłącznie po szarych polach, jednorazowo pokonując 30 kroków w wybranym kierunku. Ruchy niemożliwe do wykonania (próby wejścia na zielone pole) należało ignorować.



W praktyce oznaczało to więc:

- ustawienie duszka na polu początkowym (oznaczonym kolorem żółtym),
  a następnie:
- wykonywanie (w pętli) kolejnych ruchów zapisanych w liście poleceń (przesunięcie duszka o 30 kroków w odpowiednim kierunku).

Po wykonaniu pojedynczego ruchu (zgodnie z poleceniem z listy) należało sprawdzać kolor pola, na którym wylądował duszek (np. poleceniem `jeżeli dotyka koloru...`):

- dotknięcie koloru zielonego (niedozwolona pozycja) powinno spowodować anulowanie ruchu, czyli cofnięcie się na poprzednie pole (`przesuń o -30 kroków`);
- dotknięcie koloru czerwonego – zapisanie słowa TAK do listy `WYNIKI` i zakończenie działania programu.

Jeżeli nie nastąpiło wcześniejsze zakończenie programu (na skutek znalezienia czerwonego pola) to duszek na koniec znajdował się na polu szarym. Zatem, po wyjściu z pętli (czyli po zrealizowaniu całej sekwencji ruchów z listy `DANE`), należało zapisać słowo NIE do listy `WYNIKI`.

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/01_Proste_zadania_z_duszkami/Zabawy_w_labiryncie/solution.sb3.

Centrum Mistrzostwa Informatycznego

Dział II

Podstawowe operacje na liczbach i napisach

10 (II) – Wprowadzenie

Autor: Magdalena Wachulec, Akademia Górniczo-Hutnicza w Krakowie

Dział ten – podobnie jak poprzedni – zawiera zadania dla początkujących programistów, którzy dopiero zaczynają swoją przygodę z algorytmiką. Tym razem jednak będziemy zajmować się bardziej operacjami na liczbach i napisach, niż animowaniem duszków. Wybrane tu problemy są zaimplementowane w Scratchu, aby nawet młodsi uczniowie, nie znający jeszcze języków programowania Python/C++, mogli je swobodnie rozwiązać (ale pomału przygotowujemy się do rozwiązywania zadań również i w tych językach). Tak naprawdę, rozwiązanie każdego z tych zadań można zapisać w Pythonie lub C++, pamiętając o odmiennej specyfice operacji wejścia/wyjścia (o czym nieco więcej w dalszej części zbioru).

Zadania, choć stosunkowo proste, nie są jednak trywialne i wymagają – oprócz chęci i zapału do rozwiązywania „zagadek" – znajomości podstawowych instrukcji (instrukcja warunkowa, pętla), które już wcześniej poznaliśmy, a także sprawnego wykorzystania wyrażeń arytmetycznych, co poćwiczymy w tym dziale. Będzie nam również potrzebna umiejętność posługiwania się listami, które już pojawiły się w dziale poprzednim, a które tu teraz omówimy dokładniej, ponieważ będą odtąd stanowić dla naszego programu podstawowy sposób komunikacji ze „światem zewnętrznym".

10.1 Interakcja z użytkownikiem

Scratch posiada kilka przydatnych bloków do wyświetlania komunikatów użytkownikowi i wczytywania jego odpowiedzi (`powiedz...przez...sekund` albo `zapytaj...i czekaj`). Dzięki nim można łatwo stworzyć program, który „rozmawia" z użytkownikiem. Jednak ta forma komunikacji jest z natury interaktywna i wiąże się ze wstrzymywaniem wykonania skryptu na pewien czas. Ponieważ w zadaniach algorytmicznych często występuje potrzeba automatycznego wczytania i/lub wypisania dużej ilości danych, dlatego też na potrzeby naszych zadań zaproponowaliśmy pewne alternatywne rozwiązanie oparte na listach.

Otóż, tworząc swój program, użytkownik powinien zadeklarować dwie listy, pełniące odpowiednio rolę wejścia (na które program otrzyma dane wejściowe) i wyjścia (na które powinien wypisać wyniki). Te listy powinny mieć koniecznie nazwy: `DANE` (wejście) i `WYNIKI` (wyjście) – dokładnie tak samo, jak w zadaniu „Zabawy w labiryncie" z poprzedniego działu.

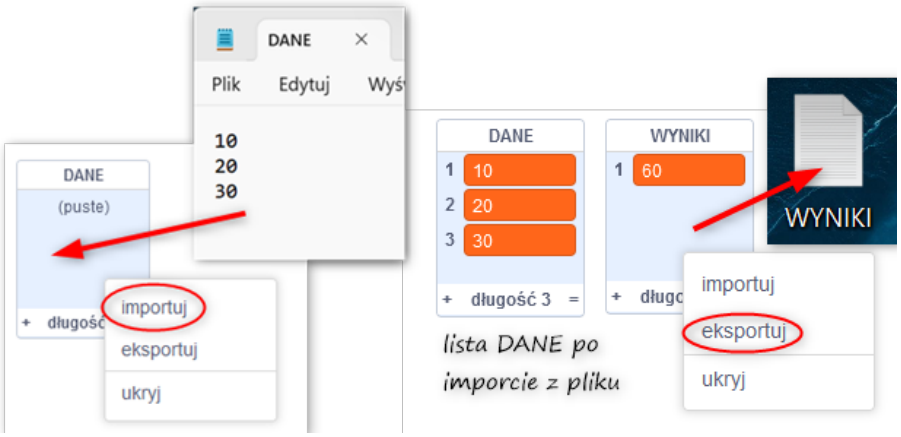
W treści zadania podana będzie zawsze *specyfikacja wejścia*, czyli szczegółowy opis tego, jak powinna wyglądać zawartość listy `DANE` przed uruchomieniem programu[1] oraz *specyfikacja wyjścia*, czyli informacja w jakiej formie program powinien wpisać rezultaty swojego działania na listę `WYNIKI`. Podczas testowania naszego programu, do listy `DANE` możemy wpisywać dane ręcznie (oczywiście zgodnie ze specyfikacją wejścia). Następnie klikamy zieloną flagę i... czekamy aż program się wykona, a potem sprawdzamy, czy to, co wpisał do listy `WYNIKI` odpowiada specyfikacji wyjścia. Kiedy już uznamy, że program działa prawidłowo, możemy wysłać go na sprawdzarkę, która zrobi to samo – wpisze do listy `DANE` odpowiednio przygotowane dane wejściowe, „kliknie" zieloną flagę, zaczeka ściśle określoną ilość czasu i... porówna zawartość listy `WYNIKI` z wzorcowymi

[1] Możesz założyć, że dane wejściowe są *zawsze* zgodne z tym opisem – Twój program nie musi tego sprawdzać.

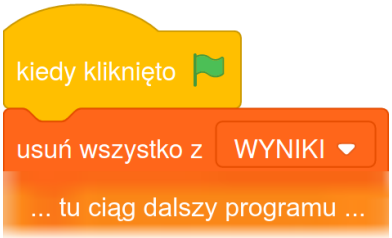
odpowiedziami.

Autor każdego zadania przygotował wiele różnych zestawów `DANE-WYNIKI`, aby sprawdzić poprawne działanie Twojego programu w wielu przypadkach testowych. Zwykle takie kolejne testy są coraz „większe” (np. zawierają coraz większe liczby, czy coraz dłuższe napisy do przetworzenia) i tylko program, który potrafi szybko uporać się również z ostatnimi, największymi testami, może być uznany za stuprocentowo poprawny. Jeśli program działa za długo (bo nasz algorytm ma zbyt dużą *złożoność obliczeniową*), sprawdzarka przerwie go w trakcie wykonania i stwierdzi zapewne, że wartości wpisane na listę `WYNIKI` niestety nie odpowiadają oczekiwanym.

Wynika z tego, że chcąc gruntownie przetestować nasz program powinniśmy również użyć dużych zbiorów danych. Ręczne wpisywanie ich na listę `DANE` mogłoby być zbyt żmudne, ale Scratch udostępnia nam na szczęście metodę alternatywną. Klikając prawym klawiszem myszy na listę, można *zaimportować* do niej dane z przygotowanego wcześniej pliku tekstowego. Podobnie, po zakończeniu działania programu, listę `WYNIKI` można *wyeksportować* do pliku (i porównać z zawartością pliku wzorcowego).

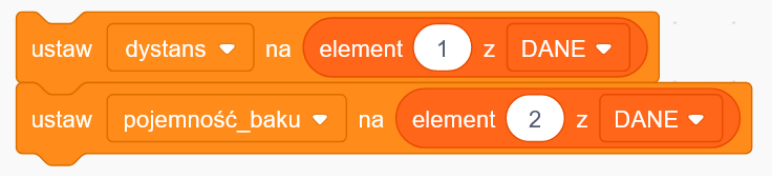


Jest ważne, aby lista `WYNIKI` po wykonaniu programu faktycznie zawierała tylko wymagane wyniki w wymaganym formacie. Należy tu zwrócić uwagę m.in. na fakt, że jeśli podczas testowania uruchamiamy nasz program wielokrotnie, to mogłoby się zdarzyć, że wyniki jego działania zostaną dopisane na końcu wyników z poprzedniego uruchomienia. Na początku programu powinniśmy więc zawsze „wyczyścić” listę wyjściową, np. używając instrukcji `usuń wszystko z WYNIKI`:



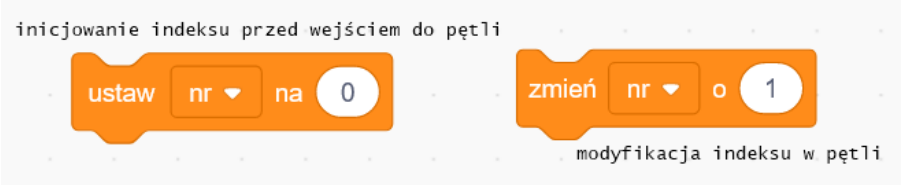
Wysyłając nasz program na sprawdzarkę należy zadbać o to, aby nie pozostawić w nim instrukcji do interaktywnej komunikacji z użytkownikiem-człowiekiem. Dotyczy to zwłaszcza instrukcji wstrzymujących pracę skryptu (np. `zapytaj...i czekaj`), bo – jak już wiemy – czas wykonania naszego programu ma fundamentalne znaczenie i musi być jak najkrótszy, a poza tym sprawdzarka i tak z tych instrukcji nie skorzysta (bo do komunikacji z naszym programem używa list `DANE` i `WYNIKI`).

Wczytywanie pojedynczych danych do programu może wyglądać np. tak:



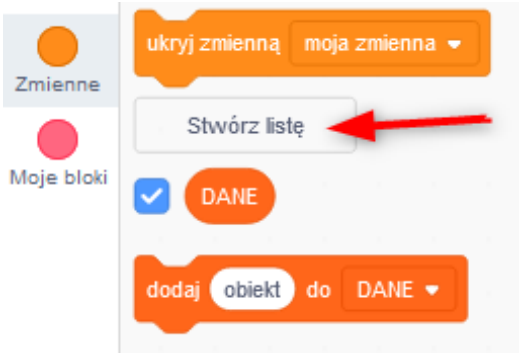
ale wczytując większą liczbę danych będziemy to zwykle robić za pomocą pętli, np.:

Zmienna sterująca pętlą (w tym przykładzie: zmienna `nr`) musi być *inicjowana przed* wejściem do pętli oraz *modyfikowana wewnątrz* niej, np.:

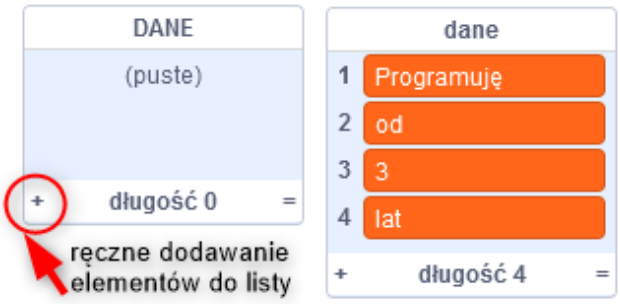


10.2 Operacje na listach

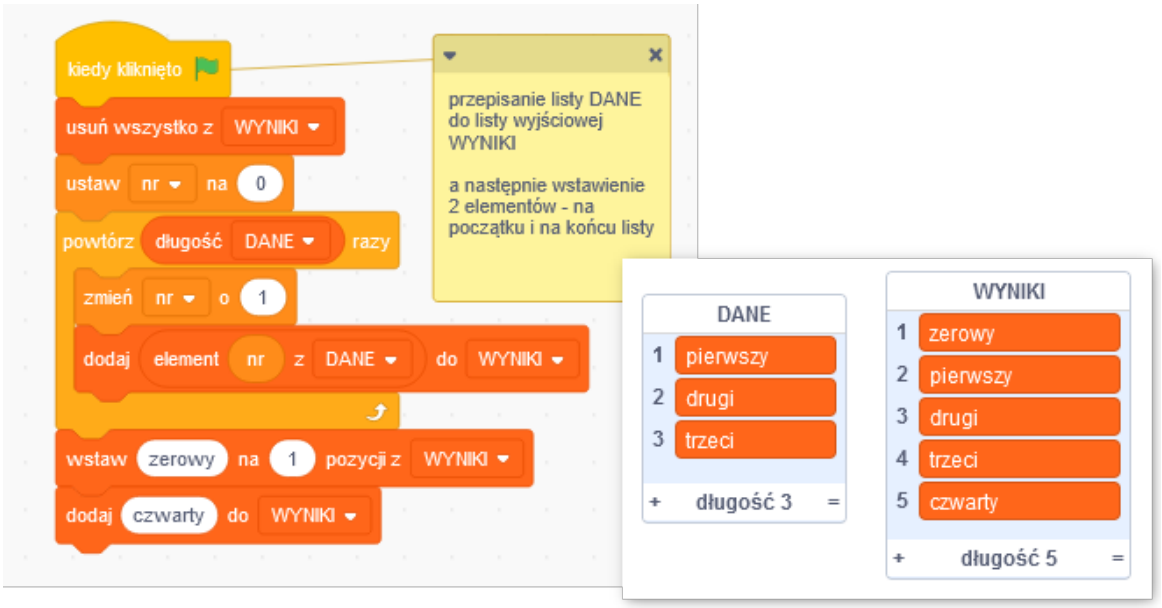
Aby utworzyć listę należy wybrać z menu `Zmienne` polecenie `stwórz listę`. Dopiero wówczas pojawią się dodatkowe bloki operacji na listach (standardowo niewidoczne). Podgląd nowej listy pojawi się również na planszy (w prawej górnej części ekranu).



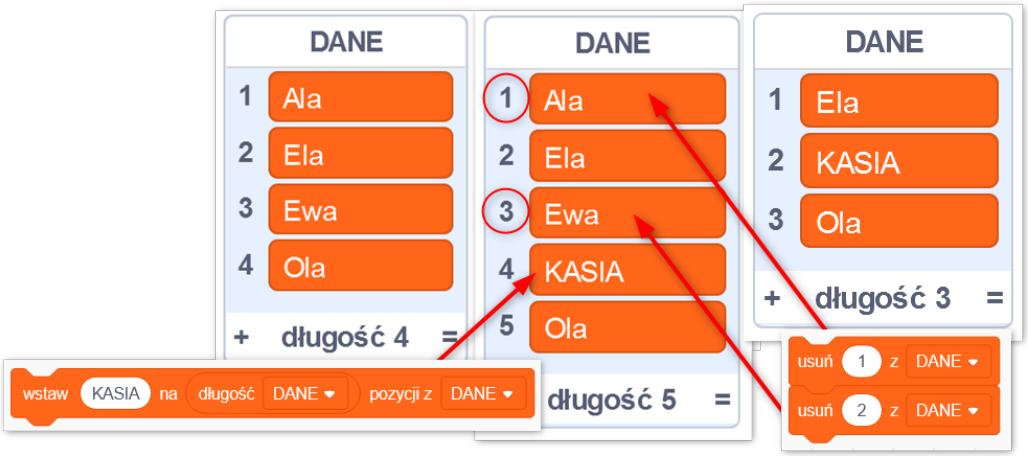
Utworzona lista jest pusta i można dopisywać do niej elementy klikając znak + (w podglądzie listy na planszy). Lista może zawierać zarówno liczby (całkowite lub rzeczywiste), jak i ciągi znaków (napisy). Nadając liście nazwę, należy pamiętać, że Scratch rozróżnia tu małe i wielkie litery. Lista `DANE` i `dane` to zatem dwa różne obiekty, dlatego dobrze jest przyjąć zasadę nazywania list `DANE` i `WYNIKI` wielkimi literami, aby uniknąć ewentualnych pomyłek.



Na poniższym rysunku pokazano przykładowy skrypt, który najpierw przepisuje dane z listy wejściowej `DANE` na listę wyjściową `WYNIKI` a następnie dopisuje do listy `WYNIKI` dwa elementy: „zerowy” – na początku (za pomocą: `wstaw...na 1 pozycji`) oraz „czwarty” – na końcu (`dodaj...do listy`).



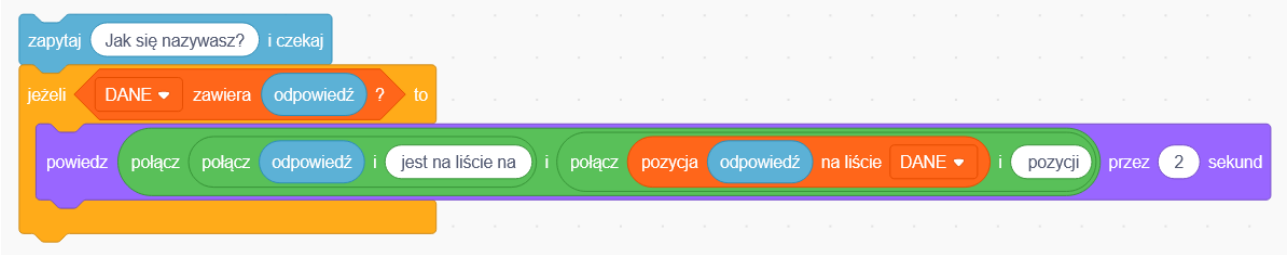
W przypadku, gdy wstawiamy element do środka listy, za pomocą instrukcji `Wstaw...na...pozycji` (przy założeniu, że: $1 \leqslant pozycja \leqslant długość\_listy$) – pozostałe elementy zostają przesunięte „w dół”, aby zrobić mu miejsce. Gdy usuwamy element – pozostałe są przesuwane „w górę”. Widać to na poniższym przykładzie:



Do czteroelementowej listy `DANE` wstawiono najpierw jeden element („KASIA”) a następnie usunięto dwa.

Dodanie elementu „KASIA” na pozycji 4 (równej długości listy) spowodowało przesunięcie elementu „Ola” na miejsce piąte. Natomiast po usunięciu pierwszego elementu („Ala”), nastąpiło ponowne „przenumerowanie” i drugim elementem została „Ewa” (gdyż „Ela” znalazła się na pierwszym miejscu).

Przydatne może się również okazać sprawdzanie, czy lista zawiera dany element, a jeśli tak, to na którym miejscu się on znajduje:



Uwaga: Jak widać z powyższych przykładów, Scratch numeruje elementy listy od 1 (w przeciwieństwie do innych języków, takich jak C++ czy Python, gdzie indeks pierwszego elementu jest równy 0, indeks drugiego jest równy 1, itd.).

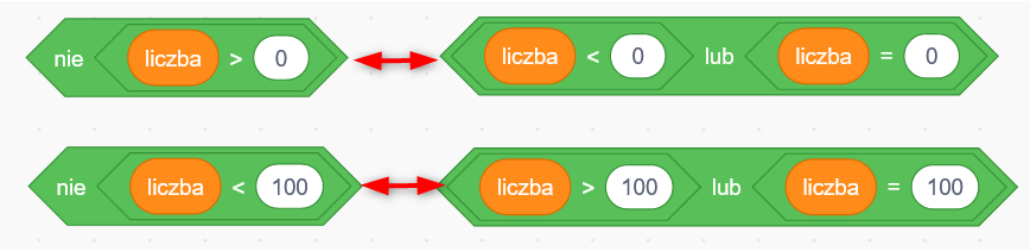
10.3 Wyrażenia arytmetyczne i logiczne

W Scratchu można stosować podstawowe operacje matematyczne (dodawanie, odejmowanie, mnożenie i dzielenie), ale o kolejności wykonywania operacji decyduje sposób łączenia bloków (tak, jakby to było działanie w nawiasie). Rozważmy poniższe przykłady:

W pierwszym przypadku do bloku dodawania wstawiono blok mnożenia, zaś w drugim – do bloku mnożenia – dodawanie. Wynik obu działań jest oczywiście różny.

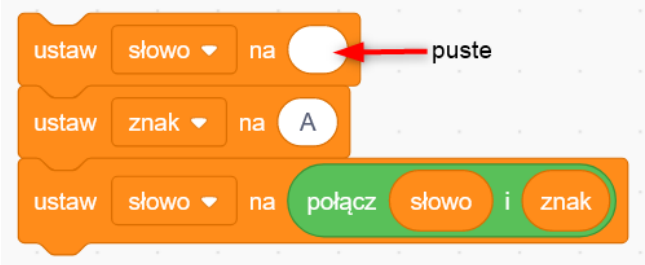
W kontekście zadań z tego działu, warto wspomnieć o przydatnych operacjach, takich jak obliczanie `reszty z dzielenia` oraz `zaokrąglanie` do najbliższej liczby całkowitej. W ostatnim bloku z sekcji „zielonej” (`Wyrażenia`), oprócz `wartości bezwzględnej` i innych funkcji matematycznych, dostępne są również dodatkowe funkcje zaokrągleń: `podłoga` – czyli obcięcie części ułamkowej oraz `sufit` – zaokrąglenie „w górę” do najbliższej liczby całkowitej.

Rozwiązując zadania, będziemy często korzystać ze standardowych operatorów porównania (=, <, >). Warto pamiętać, że można w Scratchu skonstruować również operatory $\leqslant$ oraz $\geqslant$, wykorzystując do tego odpowiednie funkcje logiczne:



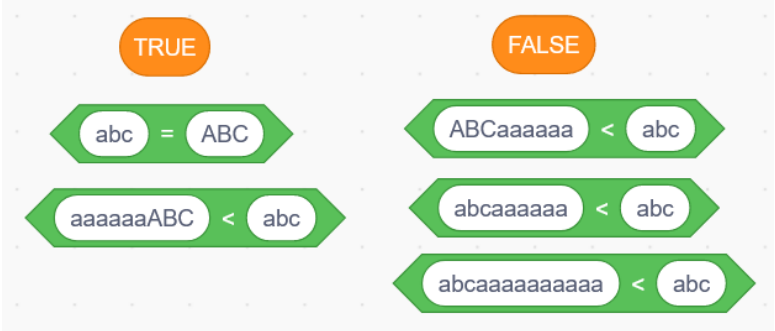
10.4 Operacje na napisach (ciągach znaków)

Oprócz podstawowych operacji arytmetycznych i logicznych, często stosowane są także operacje na tekstach. Za pomocą bloku `połącz...i...` można łączyć ze sobą dowolne ciągi znaków. W szczególności, można np. składać słowo „znak po znaku”:



Można się także odwoływać do dowolnego znaku w tekście (np. pierwszego, i-tego, czy ostatniego):

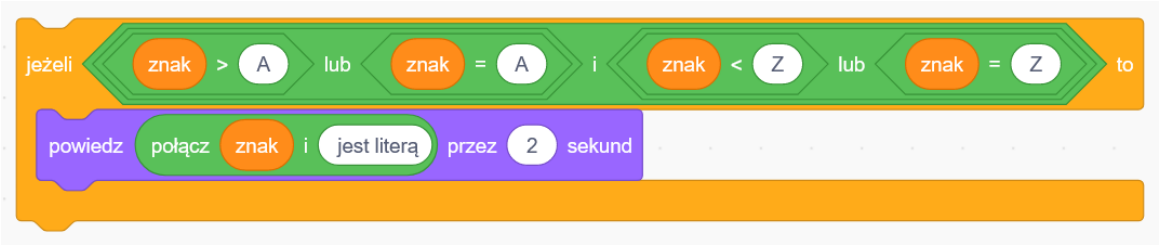
Przy porównywaniu napisów należy pamiętać, że Scratch – w przeciwieństwie do C++ i Pythona – nie rozróżnia wielkich i małych liter, zatem ciągi znaków „ala”, „ALA”, „AlA” oraz „aLa” są uznawane przez Scratcha za identyczne. Niemniej jednak, napisy można w ogólnym przypadku porównywać ze sobą i wynik jest zgodny z porządkiem leksykograficznym (takim jak w słowniku), co ilustrują poniższe przykłady.



Dla porównania – w Pythonie uzyskamy nieco inne wyniki:

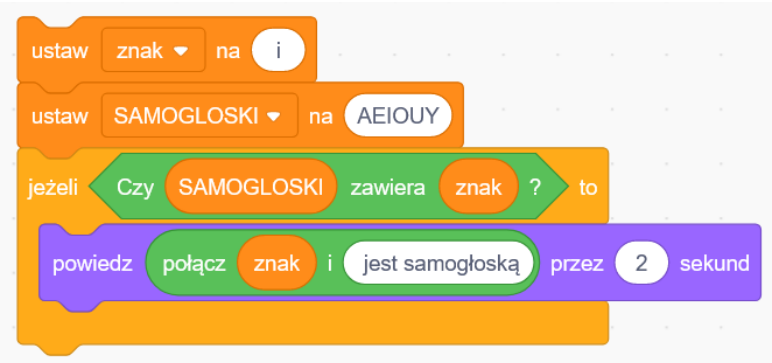
```
# dla poniższego kodu otrzymamy w wyniku: True True False False True
print("ABC"<"abc")          # True
print("ABCaaaaa"<"abc")     # True
print("abcaaaaa"<"abc")     # False
print("aaaaaaabc"<"ABC")    # False
print("aaaaaaabc"<"abc")    # True
```

Warto też zwrócić uwagę, że mimo iż Scratch nie posiada funkcji zamieniającej znak na jego kod ASCII i na odwrót, to można np. łatwo sprawdzić, czy dany znak jest literą. Można tu zastosować wersję dłuższą:



lub (korzystając z zaprzeczenia warunku) krótszą:

Łatwo też sprawdzić (i to bez żmudnego porównywania i stosowania operatorów logicznych), czy dany znak jest samogłoską:



lub czy jakiś ciag znaków jest zawarty w innym ciągu (np. „kot” w słowie „kotek”):

11 (II) – Wakacje – Skrzaty

Autor: Marcin Markowski, Politechnika Wrocławska
Kategoria: Skrzaty (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

11.1 Treść zadania

Nadeszły wyczekiwane przez wszystkich uczniów Bajtocji wakacje! Wyczekiwane zwłaszcza przez uczniów jednego z miast – Bajtysławy – a to dlatego, że w tym roku wszyscy oni wyjeżdżają na letnie kolonie i obozy, w różne ciekawe i pełne atrakcji miejsca Bajtocji. Na dworcu centralnym Bajtysławy stoją już przygotowane pociągi, które przewiozą uczniów – w każde z miejsc docelowych pojedzie jeden pociąg. Zadaniem naczelnika dworca jest teraz kontrola – czy na pewno wszyscy uczniowie zmieszczą się w pociągach. Dostępny jest spis wszystkich pociągów, ale niestety nie ma na nim łącznej liczby miejsc w danym pociągu – jest za to lista wagonów każdego pociągu i liczba miejsc w każdym wagonie. Naczelnik zna również liczbę uczniów, którzy jadą w każde z miejsc docelowych.

Pomóż naczelnikowi dworca przygotować listę kontrolną pociągów i sprawdzić, czy wszyscy uczniowie będą mogli bez przeszkód pojechać na wymarzony wypoczynek!

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania. Wynikiem kontroli każdego z pociągów powinno być słowo TAK, jeżeli w pociągu jest wystarczająca liczba miejsc lub liczba brakujących miejsc, jeśli jest ich za mało (dzięki temu będzie wiadomo ile miejsc w pociągu trzeba dodatkowo zorganizować).

Opis wejścia

Lista `DANE` zawiera liczby potrzebnych miejsc w pociągach i dane pociągów, według następującego schematu:

```
Liczba celów podróży (równa liczbie pociągów)
Liczba potrzebnych miejsc w pociągu 1
Liczba wagonów pociągu 1
Liczba miejsc w wagonie 1 pociągu 1
Liczba miejsc w wagonie 2 pociągu 1
...
Liczba potrzebnych miejsc w pociągu 2
Liczba wagonów pociągu 2
Liczba miejsc w wagonie 1 pociągu 2
```

```
Liczba miejsc w wagonie 2 pociągu 2
...
i tak dalej ...
```

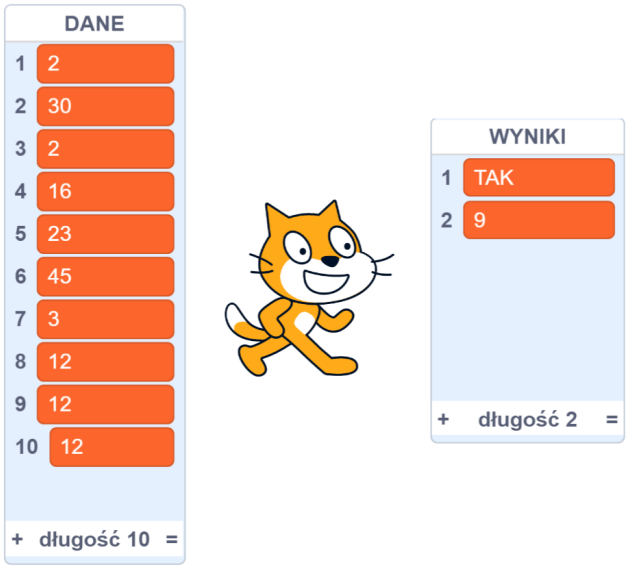
Wiadomo, że w każde z miejsc docelowych na pewno jedzie przynajmniej jeden uczeń, każdy pociąg ma co najmniej jeden wagon, a w każdym wagonie jest co najmniej jedno miejsce dla pasażera.

Opis wyjścia

Twój program po uruchomieniu (za pomocą zielonej flagi) powinien dla każdego pociągu wypisać na liście `WYNIKI` rezultat weryfikacji: TAK, gdy uczniowie się zmieszczą w pociągu lub liczbę brakujących miejsc w przeciwnym przypadku.

Przykład

Na poniższym rysunku pokazano przykładowy wynik działania programu. Uczniowie jadą w dwa miejsca wakacyjne – w Góry Bitoskaliste (pierwszy pociąg) oraz nad Jezioro Scratch-Ness (drugi pociąg). Nazwy miejscowości nie mają tu oczywiście znaczenia, ale ważne jest, że w góry jedzie 30 uczniów, a pierwszy pociąg składa się z dwóch wagonów (16 i 23 miejsca), zaś nad jezioro jedzie 45 uczniów, a drugi pociąg składa się z 3 wagonów (w każdym po 12 miejsc).



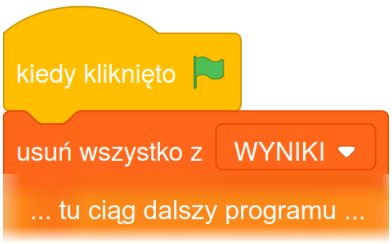
Rysunek 11.1: Przykładowe dane wejściowe i oczekiwane wyniki działania programu

Uzyskane `WYNIKI` jednoznacznie wskazują, że pierwszy pociąg (w Góry Bitoskaliste) ma wystarczającą liczbę miejsc, a w drugim (nad Jezioro Scratch-Ness) brakuje miejsc dla 9 uczniów.

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



11.2 Rozwiązanie

Zadanie warto rozłożyć na podzadania przed rozpoczęciem implementacji. Zwróćmy uwagę, że identyczne czynności trzeba wykonać dla każdego z pociągów (czyli miejsc wakacyjnych). Dane dotyczące każdego z pociągów są następujące:

```
Liczba potrzebnych miejsc w pociągu X
Liczba wagonów pociągu X
Liczba miejsc w wagonie 1 pociągu X
Liczba miejsc w wagonie 2 pociągu X
...
```

Należy obliczyć liczbę dostępnych miejsc w całym pociągu – do tego potrzebna będzie pętla wykonywana tyle razy ile jest wagonów. Należy przy tym pamiętać, że dla każdego pociągu pętla ta może mieć inną liczbę iteracji. Wyliczoną wartość należy porównać z liczbą potrzebnych miejsc i dopisać odpowiedni wynik do listy WYNIKI.

Powyższe czynności należy powtórzyć dla każdego pociągu, zatem potrzebna będzie druga, nadrzędna pętla, wykonywana tyle razy ile jest pociągów.

Pewnym wyzwaniem jest określenie, gdzie zaczynają się dane nowego pociągu (bo każdy może mieć inną liczbę wagonów). Można na dwa sposoby podejść do rozwiązania tego problemu:

1. Usuwać wszystkie wykorzystane elementy listy. Wówczas po zakończeniu obliczeń dla danego pociągu, dane kolejnego będą się zaczynały w pierwszym elemencie listy.
2. Można również zdefiniować zmienną, w której zapamiętamy (w każdej iteracji) indeks pierwszego elementu dotyczącego pociągu (będzie to liczba potrzebnych miejsc) i wykorzystać ją do indeksowanie kolejnych danych. W poniższym przykładzie taką zmienną nazwano `Pierwszy`. Pętlę wykonujemy (`Pierwszy+1`) razy – to liczba wagonów. Obliczoną sumę możemy później porównać z wartością elementu `Pierwszy` (to liczba potrzebnych miejsc).

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/02_Podstawowe_operacje/Wakacje/solution.sb3.

12 (II) – Ukryte hasło – Skrzaty

Autor: Marcin Markowski, Politechnika Wrocławska
Kategoria: Skrzaty (II edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

12.1 Treść zadania

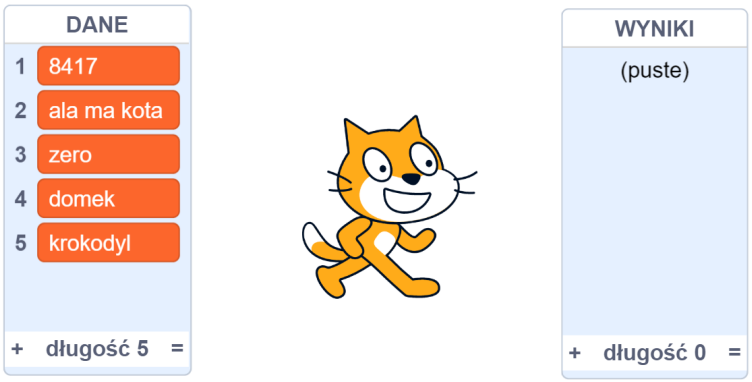
Bitek i Bajtek postanowili opracować szyfr, umożliwiający im przekazywanie sobie nawzajem tajnych haseł. Podstawą systemu szyfrującego jest lista zawierająca łańcuchy znaków (mogą to być słowa lub zupełnie losowe litery, cyfry i inne znaki). Każdy łańcuch służy do szyfrowania jednego znaku tajnego hasła, a szyfruje się go jako numer pozycji tego znaku w łańcuchu. Na przykład, za pomocą ciągu znaków „SCRATCH” można zaszyfrować literę „S” jako „1” (jest pierwszym znakiem w łańcuchu), literę „C” jako „2” lub „6”, itd. Bitek i Bajtek posiadają dwie identyczne listy łańcuchów znaków, mogą teraz przesyłać sobie tajne hasła zaszyfrowane w postaci ciągów cyfr. Napisz program, który pozwoli Bitkowi odszyfrowywać tajne hasła przesyłane przez Bajtka. Bitek posiada listę łańcuchów znaków oraz ciąg cyfr przesłanych przez Bajtka. Cyfr powinno być dokładnie tyle, ile jest łańcuchów szyfrujących – jeśli jest ich więcej lub mniej, oznacza to, że Bajtek pomylił się przy szyfrowaniu.

Zadanie

W nowym, pustym projekcie Scratcha należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `WYNIKI` należy na razie pozostawić pustą. Do listy `DANE` należy wpisać (ręcznie) ciąg cyfr oraz łańcuchy znaków, zgodnie z przykładem poniżej:

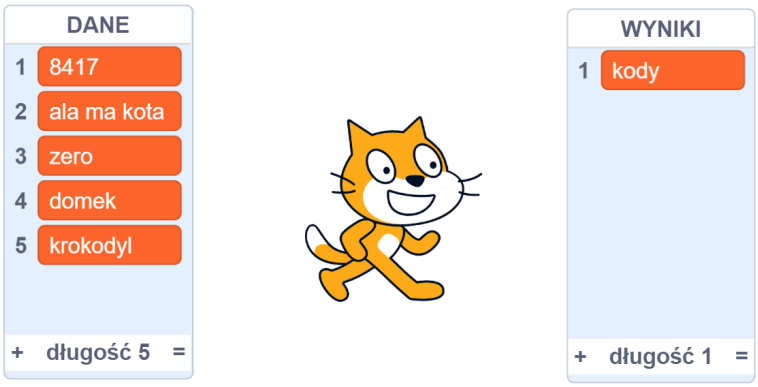


Rysunek 12.1: Układ początkowy

Pierwszy element listy zawiera zakodowane (w postaci ciągu cyfr) tajne hasło. Kolejne elementy listy to łańcuchy znaków używane do szyfrowania i deszyfrowania. Pierwsza cyfra oznacza numer znaku w pierwszym łańcuchu

(czyli drugim elemencie listy), druga cyfra – numer znaku w drugim łańcuchu, i tak dalej.

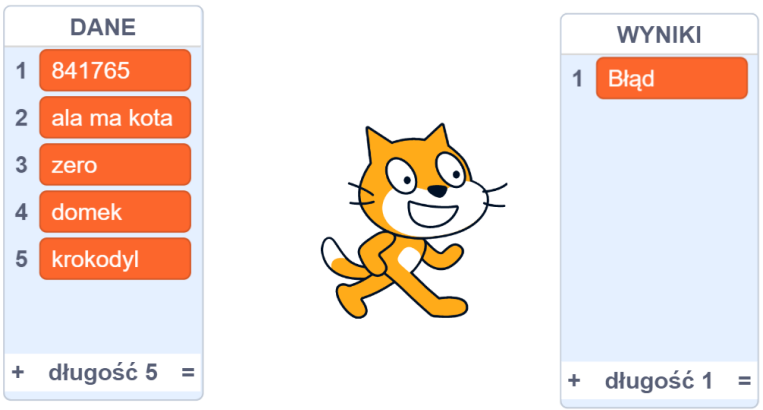
Celem zadania jest napisanie programu (skryptu), który odszyfruje tajne hasło i wpisze je jako pierwszy element listy `WYNIKI`. Na rysunku poniżej widać efekt działania tego programu:



Rysunek 12.2: Przykładowy efekt działania programu

Pierwsza cyfra ciągu cyfr (8) oznacza, że pierwszym znakiem tajnego hasła jest ósmy znak pierwszego łańcucha znaków (łańcuch „Ala ma kota”, jego ósmy znak to litera „k”). Druga cyfra (4) wskazuje czwarty znak drugiego łańcucha („zero”), czyli „o”, i tak dalej. Odszyfrowane tajne hasło to „kody”.

Jeżeli liczba cyfr w ciągu jest inna niż liczba łańcuchów znaków, wówczas program powinien na listę `WYNIKI` wpisać informację „Błąd”. Przykład pokazano na rysunku poniżej – ciąg zawiera 6 cyfr a łańcuchów jest tylko 4 – oznacza to że Bajtek pomylił się przy szyfrowaniu i nie można odszyfrować tajnego hasła.

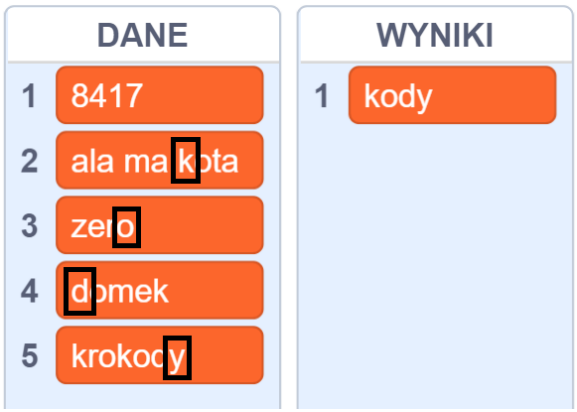


Rysunek 12.3: Przykładowy efekt działania programu

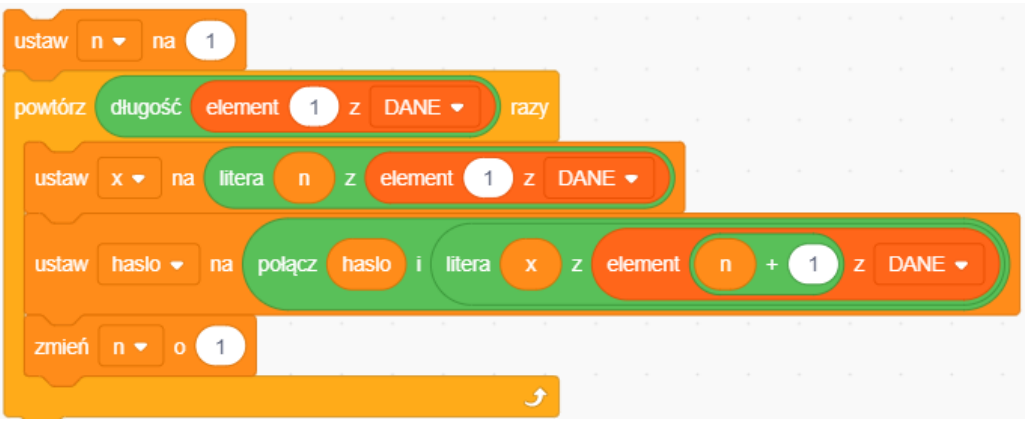
12.2 Rozwiązanie

Zadanie jest zainspirowane jedną z metod szyfrowania z „epoki przedkomputerowej”: komunikujące się osoby posiadały np. tę samą książkę-klucz, a szyfr zawierał zestawy [numer strony, numer wiersza, numer słowa], składające się na zaszyfrowaną wiadomość.

W naszym zadaniu mamy klucz składający się ze słów, a szyfr zawiera numery liter z każdego słowa. Rozwiązanie polega na odczytaniu z każdego słowa (elementu listy wejściowej) wskazanej litery i złożenia z tych liter ukrytego hasła. Przykład poniżej:



Z pierwszego elementu listy trzeba „wyłuskać” każdy kolejny n-ty znak, potraktować go jako jednocyfrową liczbę (x) i odczytać x-tą literę z $(n+1)$-ego elementu listy:



Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/02_Podstawowe_operacje/Ukryte_haslo/solution.sb3.

13 (II) – Dwie cukiernie – Skrzaty

Autor: Magdalena Wachulec, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Skrzaty (II edycja – liga algorytmiczna)
Język programowania: Scratch

13.1 Treść zadania

Na dwóch przeciwległych krańcach Alei Łasuchów znajdują się dwie bardzo popularne cukiernie, z których jedna słynie z tego, że serwuje do słodyczy wyłącznie kawę (w różnej postaci), zaś druga – herbatę. Przy Alei znajduje się dużo domów, których mieszkańcy polubili codzienne wycieczki do jednej z dwóch cukierni. Na zdjęciach satelitarnych uwieczniono licznych amatorów słodyczy spieszących w wybranych kierunkach. Każdy mieszkaniec wychodzi ze swego domu przy Alei Łasuchów i wędruje w górę ulicy („w prawo") w stronę kawiarni, lub w dół ulicy („w lewo") w stronę herbaciarni.
Pomóż kronikarzowi odpowiedzieć na pytanie – ile par (wielbicieli kawy i wielbicieli herbaty) minęło się każdego dnia w drodze do wybranych cukierni.

Zadanie

W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać rezultat swojego działania. Zadanie polega na napisaniu programu, który odczyta z listy `DANE` ciąg słów złożonych z liter K oraz H, opisujących położenie (oraz kierunek) pieszych na drodze, a następnie obliczy ile par osób wędrujących w przeciwnych kierunkach spotkało się w drodze do cukierni każdego dnia (zakładając, że codziennie wszyscy mieszkańcy dotarli do wybranej cukierni). Litera K oznacza osobę idącą w prawo w stronę kawiarni, zaś H – osobę idącą w kierunku przeciwnym – w stronę herbaciarni. Każdy mieszkaniec wyrusza ze swojego domu, przy Alei Łasuchów. Najbliżej herbaciarni znajduje się dom o numerze jeden i numeracja rośnie w górę ulicy.

Opis wejścia

Lista wejściowa `DANE` zawiera pewną liczbę (nie większą niż 100 000) słów złożonych z liter K oraz H, opisujących kierunek ruchu spacerowiczów w poszczególnych dniach. Kolejność liter w słowie odpowiada położeniu domu, z którego wychodzi spacerowicz. Domy numerowane są od początku Alei, zatem dom nr 1 stoi najbliżej herbaciarni, zaś ostatni dom – tuż przed kawiarnią.

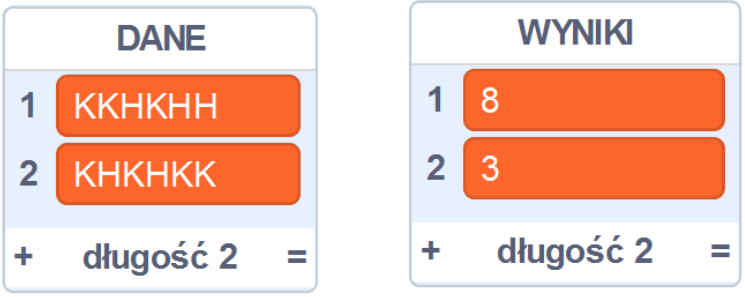
Opis wyjścia

Po zakończeniu Twojego programu lista `WYNIKI` powinna zawierać liczby określające, ile par osób idących

w przeciwnych kierunkach spotkało się w drodze do cukierni każdego dnia.

Przykład

Na poniższym rysunku pokazano przykładowe wyniki działania programu.



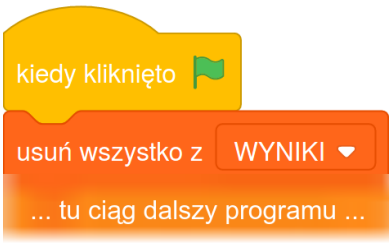
Wyjaśnienie przykładu

W pierwszym dniu minie się 8 par. Miłośnicy kawy z domu nr 1 i nr 2 spotkają w drodze aż 3 miłośników herbaty, ale mieszkaniec domu nr 4 spotka tylko dwie osoby zmierzające do herbaciarni. Oto wykaz par, które miną się po drodze: (1,3), (1,5), (1,6), (2,3), (2,5), (2,6), (4,5), (4,6).
W drugim dniu spotkają się tylko 3 pary, ponieważ mieszkańcy domów 5 i 6 znajdują się na tyle blisko kawiarni, że nikogo po drodze już nie miną. Opis par: (1,2), (1,4), (3,4)

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano poniżej:



13.2 Rozwiązanie

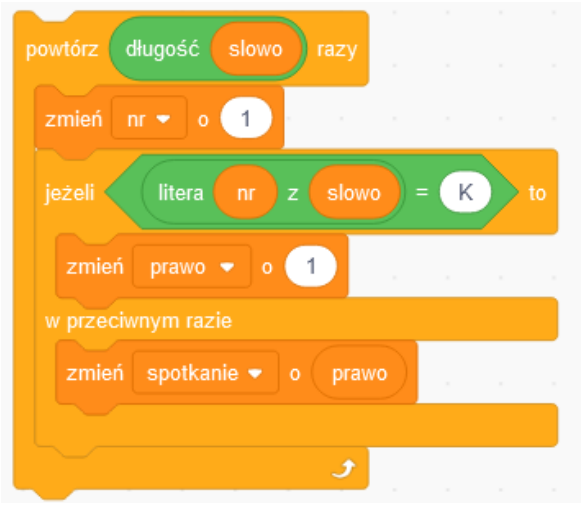
Każdy element listy `DANE` zawiera słowo złożone z liter K oraz H. Litery K oznaczają miłośników kawy, idących w górę ulicy („w prawo"), zaś litery H – miłośników herbaty zmierzających w dół ulicy („w lewo"). Zadanie polega na zliczeniu par osób, które się spotkają.

Dla przykładowych danych KKHKHH można zauważyć, że:

- pierwszy miłośnik kawy spotyka aż 3 miłośników herbaty: (1,3), (1,5), (1,6),
- drugi analogicznie: (2,3), (2,5), (2,6),
- natomiast mieszkaniec domu nr 4 spotyka tylko dwie osoby zmierzające w stronę herbaciarni: (4,5), (4,6).

Proste rozwiązanie mogłoby polegać na przeglądaniu całego ciągu liter (w pętli „głównej" – od pierwszej do ostatniej litery) i zliczaniu dla każdej litery K („idę w prawo") – liczby liter H znajdujących się na prawo od niej. Zauważmy jednak, że to zliczanie też wymagałoby osobnej pętli (przechodzącej po wszystkich literach od bieżącej litery K w prawo), zagnieżdżonej w pętli „głównej". Nasze rozwiązanie miałoby więc złożoność $O(n^2)$, czyli kwadratową (względem długości słowa wejściowego). Rozwiązanie to określimy jako *siłowe* (ang. *brute-force*), albo inaczej: *naiwne* – skuteczne, jednak o zbyt dużej złożoności obliczeniowej, ponieważ dla tego zadania można podać algorytm szybszy.

Rozwiązanie optymalne naszego problemu ma złożoność liniową – wystarczy przejrzeć ciąg liter zaledwie raz. Użyjemy w tym celu pojedynczej pętli, której „licznik" (zmienna `nr`) będzie przyjmować kolejne wartości od 1 do liczby liter w słowie wejściowym:



Na początku zmienne `prawo` oraz `spotkanie` (szukana liczba spotkań) są wyzerowane. Zmienna `prawo` będzie wskazywać aktualną liczbę zwolenników kawiarni (liczba liter K, czyli „idę w prawo"), których napotkaliśmy podczas przeglądania naszego ciągu liter.

Po co nam ten „licznik amatorów kawy"? Otóż, jeśli kolejną osobą okaże się amator herbaty (litera H), to będziemy od razu wiedzieli ilu „kawoszy" minie po drodze – właśnie dokładnie tylu, ilu naliczyliśmy dotąd! Dla każdej litery H („idę w lewo") zwiększamy zatem całkowitą liczbę spotkań (`spotkanie`) o tyle, ile wynosi bieżąca wartość `prawo`.

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/02_Podstawowe_operacje/Dwie_cukiernie/solution.sb3.

14 (II) – Przebieranki Małgosi – Skrzaty

Autor: Magdalena Wachulec, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap finałowy)
Język programowania: Scratch

14.1 Treść zadania

Małgosia uwielbia przebieranki. Zgromadziła całkiem pokaźną kolekcję sukienek, butów i kapeluszy. Małgosia zawsze wkłada wszystkie elementy stroju (kapelusz, sukienkę i buty), ale przy wyborze poszczególnych części garderoby kieruje się następującymi zasadami:

- nigdy nie ubiera się wyłącznie na czarno;
- wybierając kolorową (nie-czarną) sukienkę dobiera do niej oba dodatki w tym samym, co sukienka, kolorze *lub* czarny kapelusz i czarne buty (np. do czerwonej sukienki wkłada czerwony kapelusz oraz czerwone buty lub czarny kapelusz i czarne buty);
- po wyborze czarnej sukienki – dobiera buty i kapelusz w dwóch różnych kolorach (za wyjątkiem czarnego) – np. biały kapelusz, czarna sukienka, czerwone buty.

Małgosia ma ubrania w pięciu kolorach, przy czym używa ich nazw angielskich (black, green, red, white, yellow). Wszystkie elementy stroju są jedyne i niepowtarzalne (różnią się odcieniem, wzorem, krojem, fasonem i innymi szczegółami). A zatem, jeśli Małgosia ma np. trzy zielone kapelusze, to łącząc je z tą samą sukienką i butami uzyskuje trzy *różne* stroje.

Pomóż Małgosi ustalić, na ile sposobów może skompletować swój codzienny strój.

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania. Należy obliczyć, na ile sposobów Małgosia może skompletować swój codzienny strój.

Opis wejścia

Lista `DANE` zawiera:

- k – liczbę kapeluszy;
- kolejno k słów określających kolory kapeluszy (black, green, red, white, yellow);
- s – liczbę sukienek;
- kolejno s słów określających kolory sukienek (black, green, red, white, yellow);
- b – liczbę par butów;
- kolejno b słów określających kolory butów (black, green, red, white, yellow).

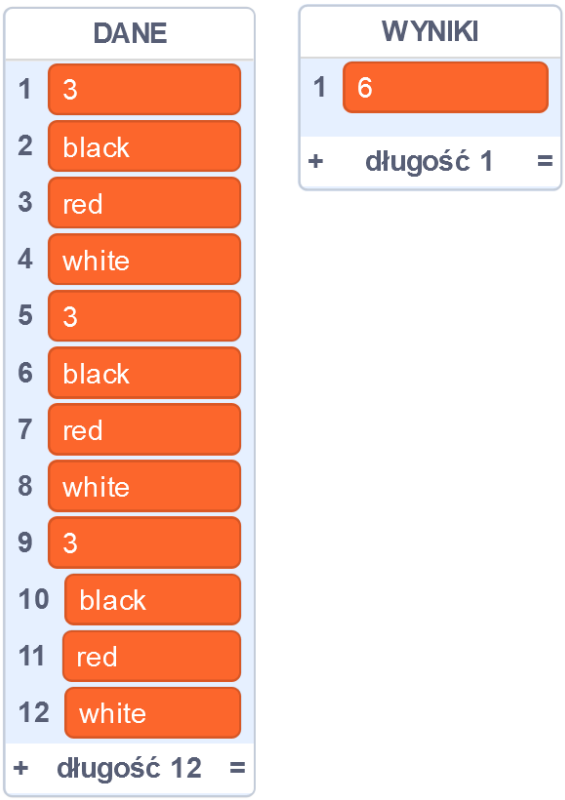
Opis wyjścia

Twój program po uruchomieniu (za pomocą zielonej flagi) powinien wypisać na liście `WYNIKI`, na ile sposobów Małgosia może wybrać codzienne ubranie.

Przykłady

Przykład 1

Na poniższym rysunku przedstawiono przykładowe dane wejściowe i prawidłowy wynik działania programu:



Rysunek 14.1: Przykład 1

Jak widać, Małgosia ma 3 kapelusze (czarny, czerwony i biały) oraz 3 sukienki i 3 pary butów w tych samych kolorach.

Zgodnie ze swoimi zasadami Małgosia może się ubrać na czerwono lub biało (2 sposoby), do czarnej sukienki może włożyć biały kapelusz i czerwone buty lub na odwrót – czerwony kapelusz i białe buty (2 sposoby). Może także wybrać czarne dodatki (buty i kapelusz) i włożyć białą lub czerwoną sukienkę (2 sposoby). W sumie zatem Małgosia ma 6 możliwości i taka jest poprawna odpowiedź wypisana na listę `WYNIKI`.

Przykład 2

Poniżej podano inny przykład zawartości szafy Małgosi:

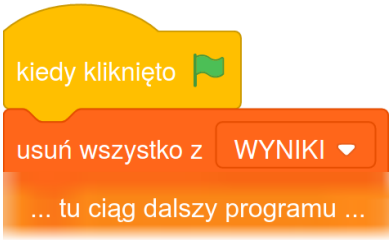
Rysunek 14.2: Przykład 2

Tym razem Małgosia ma 2 kapelusze (czerwony i zielony), 4 sukienki – białą, zieloną i 2 czerwone oraz 3 pary butów (białe, czerwone i czarne).

W tym przypadku Małgosia nie może się ubrać na zielono, bo brakuje jej butów, ani na biało, bo brakuje jej kapelusza. Do czterech kolorowych sukienek nie może dobrać czarnych dodatków (z powodu braku czarnego kapelusza), a brak czarnej sukienki uniemożliwia wybór różnokolorowych dodatków. Jedyna możliwość to wybrać dwie różne czerwone sukienki z czerwonymi dodatkami. W związku z tym, na liście `WYNIKI` program wypisał wartość 2.

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie. Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



14.2 Rozwiązanie

Zadanie polegało na obliczeniu, na ile sposobów można skompletować ubranie, składające się z 3 elementów: kapelusza, sukienki i butów. Każda z części garderoby była niepowtarzalna (fason, krój, materiał itp.) i występowała w jednym z pięciu dostępnych kolorów (black, red, white, green i yellow). Dodatkowym ograniczeniem było to, że nie wszystkie kolory można było ze sobą zestawiać (no cóż... *de gustibus non est disputandum*).

Na początek należało zliczyć, ile sztuk (kapeluszy, par butów i sukienek) zgromadziła Małgosia w poszczególnych kolorach, a liczby te zapisać, dla wygody, w trzech 5-cio elementowych listach (`KAPELUSZE`, `BUTY`, `SUKIENKI`). Oczywiście liczba butów oznacza liczbę kompletnych *par butów*.

Wiedząc, ile jest ubrań w każdym kolorze (przez „buty" rozumiemy parę butów), można było przystąpić do obliczeń:

- `JK` – ubrania jednokolorowe (dla każdego koloru, oprócz czarnego):
  $liczba\ sukienek * liczba\ kapeluszy * liczba\ butow$ (liczymy osobno w każdym z 4 kolorów, a następnie sumujemy);
- `KS` – kolorowe sukienki z czarnymi dodatkami:
  $liczba\ \ NIEczarnych\ sukienek * liczba\ czarnych\ kapeluszy * liczba\ czarnych\ butow$;
- `KD` – czarne sukienki z kolorowymi dodatkami:
  $liczba\ czarnych\ sukienek * liczba\ kolorowych\ dodatkow$.
  Ponieważ *liczba kolorowych dodatków* musi uwzględniać wszystkie możliwe połączenia kolorowych (nieczarnych) kapeluszy z butami innego koloru niż kapelusz (inaczej mówiąc: czerwony kapelusz może być połączony z butami o innym kolorze niż czerwony, zielony – ze wszystkimi kolorami oprócz zielonego itd.), zatem:
  liczba kolorowych dodatków jest **sumą iloczynów**:
  $liczba\ kapeluszy\ w\ kolorze\ X * liczba\ NIEczarnych\ butow\ z\ wyjatkiem\ koloru\ X$
  gdzie X oznacza 4 kolory: *red, white, green i yellow*.

Rozwiązaniem jest suma: $\texttt{JK} + \texttt{KS} + \texttt{KD}$.

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/02_Podstawowe_operacje/Przebieranki_Malgosi/solution.sb3.

15 (II) – Tam i z powrotem – Skrzaty

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Skrzaty (II edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

15.1 Treść zadania

Pomiędzy Alokacją i Binarią – dwoma wyspami na Morzu Bitockim – regularnie kursuje prom. Wykonuje on stałe rejsy na trasie tam i z powrotem – długość tej trasy (odległość miedzy wyspami) wynosi d mil morskich.

Kursowanie promu związane jest z jego tankowaniem. Pojemność baku promu starcza na pokonanie dystansu b mil. Na trasie promu w odległości g mil od Alokacji znajduje się mała wysepka Gazolina – w zasadzie bezludna, jeśli nie liczyć stacji paliwowej, gdzie prom może uzupełnić paliwo.

Pytanie brzmi: ile razy konieczne będzie tankowanie na Gazolinie, aby prom wykonał swój plan, czyli odbył n kursów na swojej trasie? Przez kurs rozumiemy rejs z Alokacji na Binarię lub odwrotnie.

Prom rozpoczyna swoja pracę na Alokacji, z pełnym bakiem.

Zadanie

Twoim zadaniem jest znalezienie odpowiedzi na pytanie postawione powyżej: czy da się zrealizować plan kursów promu, a jeśli tak, to ile tankowań będzie niezbędnych do wykonania planu?

W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać rezultat swojego działania.

Opis wejścia

Lista wejściowa `DANE` zawiera cztery liczby naturalne: d, b, g oraz n:

- d oznacza dystans pomiędzy Alokacją i Binarią ($2 \leq d \leq 10^6$);
- b to pojemność baku w przeliczeniu na mile morskie ($1 \leq b \leq 10^9$);
- g jest odległością pomiędzy Gazoliną i Alokacją ($1 \leq g < d$);
- n to zaplanowana ilość kursów ($1 \leq n \leq 10^5$).

Opis wyjścia

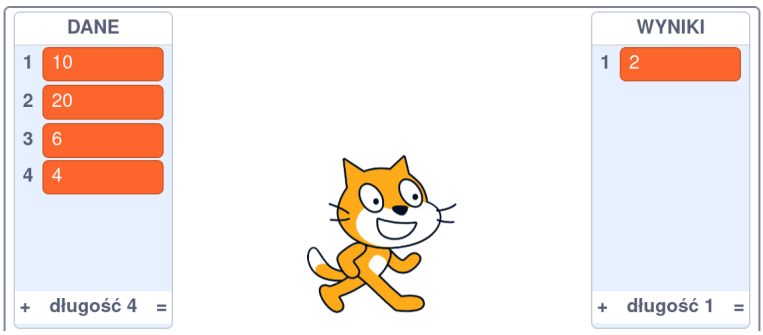
Po zakończeniu Twojego programu, w pierwszym (i zarazem jedynym) elemencie listy `WYNIKI` powinna zna-

leźć się liczba równa minimalnej ilości tankowań lub komunikat **NIE** (wielkimi literami), jeśli planu nie da się wykonać.

Przykłady

Na poniższych rysunkach pokazano przykładowe wyniki działania programu.

Przykład 1

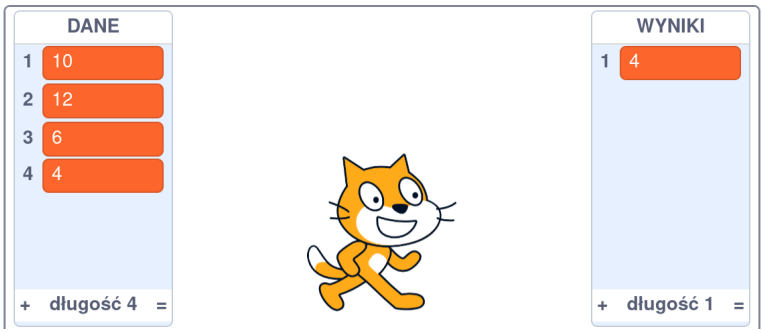


Wyjaśnienie przykładu 1:

Wyspy A(lokacja) i B(inaria) odległe są od siebie o 10 mil morskich, a Gazolina znajduje się o 6 mil od Alokacji (czyli o 4 mile od Binarii). Prom dysponuje bakiem zapewniającym pokonanie dystansu 20 mil i ma wykonać cztery kursy.

Prom płynie z A do B, w drodze powrotnej tankuje paliwa na 14 mil (do pełna), wraca do A, wypływa w kierunku B, tankuje na 12 mil i mając pełny bak może dopłynąć do B, po czym wrócić do A. Liczba tankowań wynosi zatem 2.

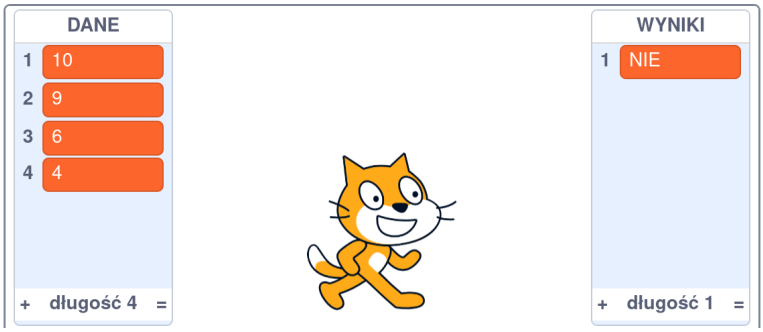
Przykład 2



Wyjaśnienie przykładu 2:

Tankowanie jest konieczne podczas każdego kursu.

Przykład 3



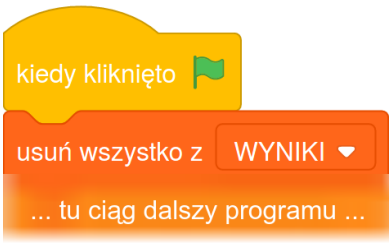
Wyjaśnienie przykładu 3:

Odbycie 4 kursów przy takich warunkach jest niemożliwe.

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładach.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano poniżej:



15.2 Rozwiązanie

W zadaniu mamy prom kursujący pomiędzy wysepkami A oraz B odległymi o d mil morskich. W odległości g od wysepki A znajduje się stacja paliwowa – tak więc cały dystans (przypadający na jeden kurs) możemy podzielić na dwa odcinki: o długości g oraz $d - g$. Ponieważ prom kursuje tam i z powrotem, zatem możemy w sprytny sposób ułatwić sobie nieco zadanie, przed każdym nawrotem podstawiając $g \leftarrow (d - g)$. Dzięki temu będziemy mogli cały czas korzystać z tych samych oznaczeń: g będzie zawsze oznaczało dystans *do stacji paliw* (z bieżącej wyspy), zaś $d - g$ to pozostała do przebycia droga *ze stacji* (do drugiej wyspy). Przykładowo – dla danych z treści zadania, wartość zmiennej g (czyli dystans do stacji paliwowej z tej wyspy, przy której aktualnie znajduje się prom) będzie równa na zmianę: 6, 4, 6, 4, ...

Główna pętla algorytmu wykonuje się dokładnie n razy – zmienna sterująca to k, zmniejszające się od n do 1. Zmienna b oznacza pojemność baku promu, zaś $tank$ to ilość paliwa znajdująca się w baku w danej chwili. Prom nie będzie mógł wykonać danego kursu, jeśli zajdzie jedno z poniższych zdarzeń:

- W baku znajduje się w danym momencie za mało paliwa, aby dotrzeć do stacji, czyli $tank < g$,
- Pojemność baku nie wystarczy na przebycie pozostałej części dystansu, nawet po zatankowaniu do pełna, czyli $b < d - g$.

W takiej sytuacji wypisywany jest negatywny komunikat i program jest przerywany.

Podczas każdego obiegu pętli sprawdzane jest, czy w baku znajduje się dość paliwa, aby dotrzeć do końca kursu (bez tankowania), następnie zawrócić i wtedy zatankować (do pełna):

$$tank \geq d + (d - g).$$

Wyjątkiem jest ostatni kurs (dla $k = 1$), gdzie wystarczy sprawdzić, czy:

$$tank \geq d,$$

ponieważ nie będzie po nim nawrotu.

W obydwu powyższych przypadkach zmienna $tank$ ulega zmniejszeniu o wartość d i z taką wartością jest rozpoczynany następny kurs (jeśli $k > 1$).

Jeśli powyższy warunek nie jest spełniony, wtedy konieczne jest tankowanie. Ilość tankowań zliczana jest w zmiennej $visit$, której wartość jest wypisywana na końcu programu. Skoro ma miejsce tankowanie, prom dotrze do końca tego kursu mając w baku $b - (d - g)$ paliwa, gdyż po zatankowaniu do pełna musi jeszcze przebyć dystans $d - g$.

Omówione tu rozwiązanie rozwiązanie znajdziesz w pliku:

./zadania/02_Podstawowe_operacje/Tam_i_z_powrotem_Skrzaty/solution.sb3.

Zauważmy tu na koniec pewien ważny fakt. Mianowicie, powyższy opis przedstawia rozwiązanie naszego problemu w postaci bardzo ogólnego algorytmu. Tak ogólnego, że w gruncie rzeczy moglibyśmy bez problemu zapisać go w dowolnym języku programowania, w którym istnieją zmienne, pętle, oraz wyrażenia arytmetyczne i warunkowe (czyli praktycznie w każdym *imperatywnym* języku programowania[1]).

Wykażemy to teraz w praktyce, prezentując rozwiązanie zadania „Tam i z powrotem” zaimplementowane w językach C++ i Python (a więc przeznaczone dla kategorii „Gnomy”). W kolejnym dziale będzie to już dla

[1]Oprócz języków imperatywnych istnieją również np. *funkcyjne języki programowania*, *języki deklaratywne* i in., w których programy zapisywane są w nieco inny sposób.

nas regułą (wszystkie rozwiązania będą prezentowane w trzech językach) – tam też znajdziesz więcej informacji o C++ i Pythonie, o podobieństwach i różnicach między nimi, w szczególności w kontekście wczytywania i wypisywania danych. Będziemy również szczegółowo omawiać rozwiązania zapisane w poszczególnych językach programowania. Tu natomiast poprzestaniemy tylko na pokazaniu implementacji algorytmu, który przed chwilą omówiliśmy – do samodzielnej analizy.

Wcześniej jednak podamy jeszcze raz treść zadania – odpowiednio zmodyfikowaną na potrzeby Gnomów (tej konwencji będziemy się odtąd trzymać prezentując zadania „podwójne", czyli przygotowane zarówno z myślą o Skrzatach, jak i o Gnomach). Sam problem jest oczywiście ten sam, ale tekstowy charakter języków C++ i Python wymusza nieco inny sposób formułowania pewnych elementów – zwłaszcza związanych z wejściem i wyjściem programu (więcej o *standardowym wejściu i wyjściu* w tych językach – w kolejnym dziale).

16 (II) – Tam i z powrotem – Gnomy

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gnomy (II edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

16.1 Treść zadania

Pomiędzy Alokacją i Binarią – dwoma wyspami na Morzu Bitockim – regularnie kursuje prom. Wykonuje on stałe rejsy na trasie tam i z powrotem – długość tej trasy (odległość miedzy wyspami) wynosi d mil morskich.

Kursowanie promu związane jest z jego tankowaniem. Pojemność baku promu starcza na pokonanie dystansu b mil. Na trasie promu w odległości g mil od Alokacji znajduje się mała wysepka Gazolina – w zasadzie bezludna, jeśli nie liczyć stacji paliwowej, gdzie prom może uzupełnić paliwo.

Pytanie brzmi: ile razy konieczne będzie tankowanie na Gazolinie, aby prom wykonał swój plan, czyli odbył n kursów na swojej trasie? Przez kurs rozumiemy rejs z Alokacji na Binarię lub odwrotnie.

Prom rozpoczyna swoja pracę na Alokacji, z pełnym bakiem.

Zadanie

Twoim zadaniem jest znalezienie odpowiedzi na pytanie postawione powyżej: czy da się zrealizować plan kursów promu, a jeśli tak, to ile tankowań będzie niezbędnych do wykonania planu?

Twój program powinien czytać dane ze standardowego wejścia i wypisywać swój wynik na standardowe wyjście.

Opis wejścia

Pierwszy i jedyny wiersz danych wejściowych zawiera cztery liczby naturalne d, b, g oraz n:

- d oznacza dystans pomiędzy Alokacją i Binarią ($2 \leq d \leq 10^6$);
- b to pojemność baku w przeliczeniu na mile morskie ($1 \leq b \leq 10^9$);
- g jest odległością pomiędzy Gazoliną i Alokacją ($1 \leq g < d$);
- n to zaplanowana ilość kursów ($1 \leq n \leq 10^5$).

Liczby w wierszu oddzielone są pojedynczymi odstępami.

Opis wyjścia

Program powinien wypisać wiersz tekstu zawierający minimalną ilość tankowań lub komunikat **NIE** (wielkimi literami), jeśli planu nie da się wykonać.

Przykłady

Dla danych wejściowych:

```
10 20 6 4
```

prawidłowym wynikiem jest:

```
2
```

Prom płynie z A do B, w drodze powrotnej tankuje paliwa na 14 mil (do pełna), wraca do A, wypływa w kierunku B, tankuje na 12 mil i mając pełny bak może dopłynąć do B, po czym wrócić do A.

Dla danych wejściowych:

```
10 12 6 4
```

prawidłowym wynikiem jest:

```
4
```

Tankowanie jest konieczne podczas każdego kursu.

Dla danych wejściowych:

```
10 9 6 4
```

prawidłowym wynikiem jest:

```
NIE
```

Odbycie 4 kursów przy takich warunkach jest niemożliwe.

16.2 Rozwiązanie

16.2.1 Python

Źródło: ./zadania/02_Podstawowe_operacje/Tam_i_z_powrotem_Gnomy/solution.py.

```python
import sys
d, b, g, n = map(int, input().split())
tank = b
visit = 0
for k in range(n, 0, -1):
    if tank < g or b < d-g:
        print("NIE")
        sys.exit(0)
    elif tank >= d + ((d-g) if k>1 else 0):
        tank -= d
    else:
        visit += 1
        tank = b - (d-g)
    g = d-g
print(visit)
```

16.2.2 C++

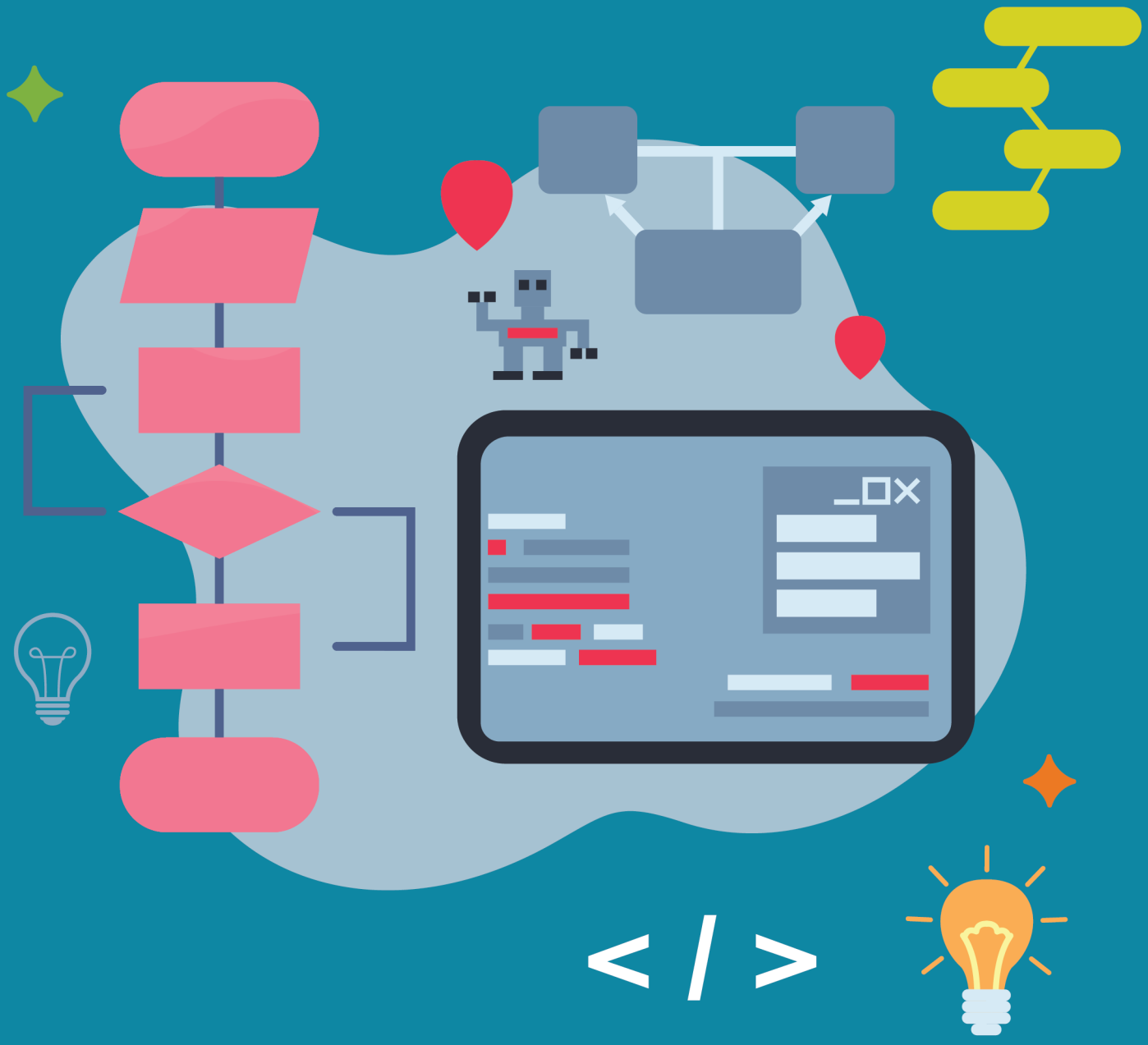
Źródło: ./zadania/02_Podstawowe_operacje/Tam_i_z_powrotem_Gnomy/solution.cpp.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
  int d, b, g, n;
  cin >> d >> b >> g >> n;
  int tank = b, visit{ };
  for(int k=n; k>=1; k--) {
    if((tank < g) || (b < d-g)) {
      cout << "NIE\n";
      return 0;
    }
    else if(tank >= d + ((k>1)? d-g : 0))
      tank -= d;
    else {
      visit++;
      tank = b - (d-g);
    }
    g = d-g;
  }
  cout << visit << '\n';
  return 0;
}
```

Centrum Mistrzostwa Informatycznego

Dział III

Wczytywanie, wypisywanie i przechowywanie danych



17 (III) – Wprowadzenie

Autor: Bartłomiej Stasiak, Politechnika Łódzka

W poprzednich działach omówiliśmy proste zadania z elementami graficznymi i obliczeniowymi oraz implementacje ich rozwiązań w Scratchu. Przechodząc do nieco bardziej złożonych algorytmów, będziemy teraz – podobnie jak w przypadku ostatniego zadania („Tam i z powrotem") – przedstawiać ich rozwiązania zaimplementowane również w innych językach programowania: w C++ i w Pythonie. Języki te oferują programiście dużo szerszy wachlarz możliwości w zakresie dostępnych struktur danych i gotowych algorytmów, szybkości działania oraz alokacji pamięci. Również samo tworzenie programu w środowisku tekstowym, choć nie tak „kolorowe", interaktywne i poglądowe jak w Scratchu, jest jednak dużo wygodniejsze i szybsze.

Należy tu wyraźnie podkreślić, że sam *algorytm* rozwiązujący dany problem może być zaimplementowany w dowolnym języku programowania. Jako ilustrację tego faktu przedstawimy w tym dziale zadania o rozwiązaniach zaimplementowanych zarówno w Scratchu, jak i w Pythonie i w C++ (w kolejnych działach wykorzystywane już będą najczęściej te dwa ostatnie, wygodniejsze języki „tekstowe").

Oczywiście, poszczególne języki programowania bardzo różnią się od siebie pod wieloma względami: zarówno w zakresie składni, czyli sposobu zapisywania programów, jak i samej realizacji procesu ich wykonywania przez komputer. Nie będziemy tu zanadto zagłębiać się w te różnice, ograniczając się tylko do elementów istotnych z punktu widzenia algorytmiki (zakładamy, że czytelnik zna podstawy programowania w danym języku). Tym, od czego trzeba by jednak zacząć, jest zauważenie, w jaki sposób w poszczególnych językach rozwiązana została fundamentalna kwestia komunikacji ze „światem zewnętrznym", czyli wczytywanie i wypisywanie danych.

17.1 Standardowe wejście/wyjście

Jak pamiętamy ze wstępu do poprzedniego działu, skrypty w Scratchu mogą komunikować się z użytkownikiem w sposób interaktywny, za pomocą odpowiednich bloków, jednak w naszych zadaniach algorytmicznych – z uwagi na potrzebę wczytywania i zapisywania naprawdę dużych ilości danych – stosujemy do tego celu specjalne listy `DANE` i `WYNIKI`.

W językach Python i C++, obie formy komunikacji – interaktywna, w której program „rozmawia z użytkownikiem" wypisując na konsoli pytanie i czekając na jego odpowiedź oraz zautomatyzowana, w której program wczytuje całą dużą porcję danych na raz – realizowane są w ten sam sposób, za pomocą mechanizmu tzw. standardowego wejścia/wyjścia. Przykładowo, program `suma.cpp` napisany w C++, wczytujący dwie liczby i wyświetlający ich sumę mógłby wyglądać tak:

```
#include <iostream>
int main() {
        int x, y;
        std::cin >> x;
        std::cin >> y;
        std::cout << x + y << std::endl;
        return 0;
}
```

Komunikacja z użytkownikiem zrealizowana jest tu za pomocą tzw. strumienia wejściowego (`cin`) i wyjściowego (`cout`), co wymaga włączenia do kodu odpowiedniej biblioteki (`iostream`).

Program ten, po kompilacji:

```
g++ suma.cpp -o suma.exe
```

i uruchomieniu w konsoli:

```
suma.exe
```

wczyta od użytkownika dwie liczby i wypisze ich sumę. Przekazanie tych danych może się jednak odbyć w sposób automatyczny, jeśli uruchomimy nasz program następująco:

```
suma.exe < dane.txt > wyniki.txt
```

gdzie `dane.txt` oznacza nazwę pliku zawierającego liczby do dodania, a `wyniki.txt` to nazwa pliku, do którego trafi wynik. Uwaga: w taki sam sposób nasz program będzie uruchamiać sprawdzarka OJ (porównując na końcu zawartość pliku `wyniki.txt` z plikiem wzorcowym, przygotowanym przez autora zadania).

Analogiczny kod w Pythonie:

```
x = int(input())
y = int(input())
print(x + y)
```

zapisany w pliku `suma.py` i uruchomiony następująco:

```
python suma.py < dane.txt > wyniki.txt
```

zadziała tak samo, *przekierowując* zawartość pliku `dane.txt` na standardowe wejście programu, a jego standardowe wyjście – do pliku `wyniki.txt`.

Oczywiście nie używając tych przekierowań:

```
python suma.py
```

będziemy musieli wpisać dane wejściowe w konsoli, w której potem również zostanie wyświetlony wynik.

17.2 Listy i tablice

W powyższych przykładach użyliśmy dwóch zmiennych: `x` i `y` do przechowania wczytanych liczb. Pojęcie *zmiennej* jest oczywiście kluczowe dla programowania (i jak pamiętamy, występuje również w Scratchu). Każda zmienna ma swoją *nazwę* i przechowuje *wartość* określonego *typu* – w naszym przypadku był to typ całkowitoliczbowy (`int`).

Często jednak potrzebujemy przechować wiele wartości tego samego typu i nie potrzebujemy nadawać każdej z nich osobnej nazwy zmiennej. Zamiast tego, umieszczamy je w specjalnej strukturze danych – „*kontenerze*” pozwalającym na przechowywanie nie jednej wartości, a całego ich zbioru (czyli *kolekcji danych*).

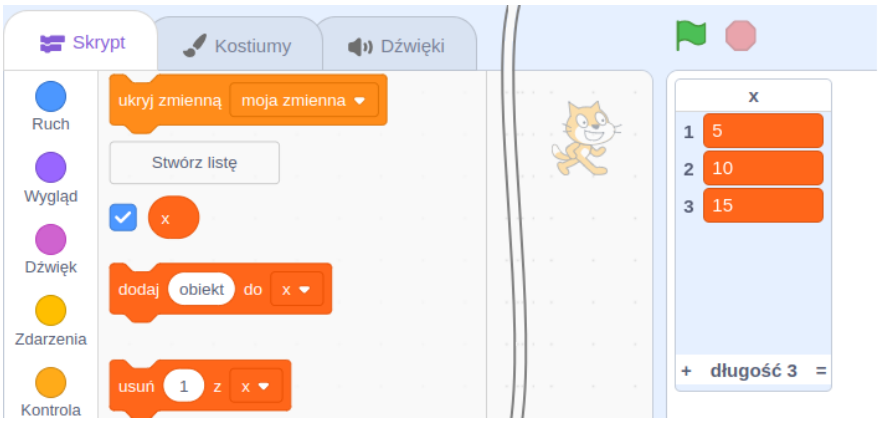
Najpopularniejszy rodzaj kontenera, to tzw. *kontener sekwencyjny*, który zawiera po prostu ciąg elementów (liczb, napisów, etc.) – najczęściej w kolejności w jakiej były do niego dodawane. W języku C++ będzie to np. *tablica*:

```
int x[3] = {5, 10, 15};
```

W Pythonie możemy użyć np. *listy*:

```
x = [5, 10, 15]
```

podobnie jak w Scratchu:

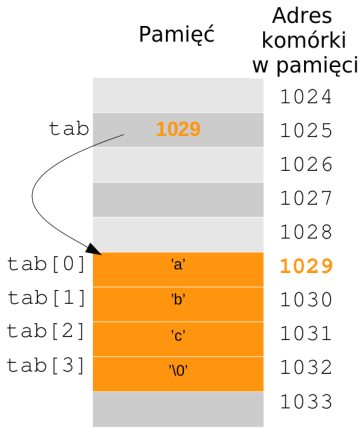


W wielu językach programowania możemy wybrać spośród kilku dostępnych struktur tego typu. Przykładowo, w C++ oprócz prostych tablic, stanowiących element składowy samego języka, możemy również używać bardziej rozbudowanych typów zdefiniowanych przez bibliotekę standardową – np. wektorów (`vector`), czy list wiązanych (`list`). W Pythonie oprócz list (które wbrew nazwie przypominają bardziej wektory lub tablice C++) istnieją także tzw. *krotki* (ang. *tuples*), różniące się od list tym, że są niemodyfikowalne. Warto pamiętać, że różnice między poszczególnymi kontenerami dotyczą nie tylko składni, czyli notacji używanej do zapisu operacji na ich elementach, ale również złożoności obliczeniowej i innych kwestii istotnych z punktu widzenia algorytmiki. Chodzi tu w szczególności o wewnętrzną organizację elementów zapisanych w pamięci. Przykładowo, w tablicy, czy w wektorze elementy zapisywane są zwykle jeden po drugim, w ciągłym obszarze pamięci:

```
char *tab = new char[4];
tab[0] = 'a';
tab[1] = 'b';
tab[2] = 'c';
tab[3] = '\0';
//...
delete [] tab;
```

Przykład dynamicznej alokacji pamięci (operatory `new` i `delete`)

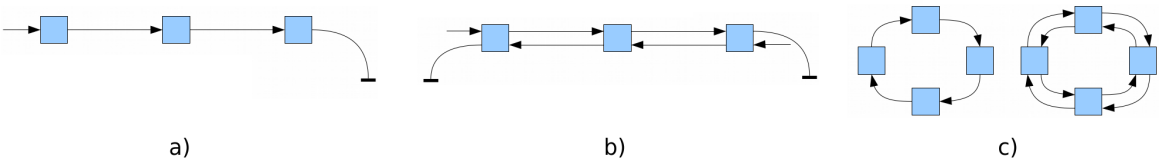
Uwaga: nazwa tablicy w C++ jest wskaźnikiem (adresem) pierwszego elementu



Rysunek 17.1: Deklaracja tablicy znaków w C++ i jej możliwa reprezentacja w pamięci operacyjnej

Taka forma organizacji danych w pamięci sprawia, że dostęp do elementów jest szybki (złożoność $O(1)$, czyli w czasie stałym), ale już np. wstawienie elementu w środek tablicy może wymagać realokacji pamięci i przepisania części elementów (co oznacza złożoność $O(n)$, a więc liniową).

W przypadku listy wiązanej, poszczególne elementy mogą być zapisane w różnych miejscach pamięci, więc potrzebna jest informacja, gdzie je znaleźć. W tym celu do każdego elementu („*węzła*”) listy dołącza się zwykle dodatkowe pole zawierające adres (wskaźnik) elementu kolejnego i/lub poprzedniego:



Rysunek 17.2: Rodzaje list wiązanych. a) Lista jednokierunkowa – każdy węzeł ma wskaźnik do następnego; b) Lista dwukierunkowa – każdy węzeł ma wskaźnik do następnego i poprzedniego; c) Lista cykliczna (jedno i dwukierunkowa)

Wstawianie elementów do listy jest zatem szybkie (brak konieczności przepisywania elementów – wystarczy dodać w odpowiednim miejscu wskaźnik do nowego elementu), ale za to odczyt elementu może wymagać iteracyjnego przejścia do niego od początku lub od końca listy (złożoność $O(n)$).

Oczywiście można powiedzieć, że to są wewnętrzne szczegóły implementacyjne poszczególnych struktur danych, o których korzystający z nich programista nie musi za dużo wiedzieć. Niemniej jednak, wybór odpowiedniego sposobu przechowywania danych, uwzględniający jakie operacje dominują w tworzonym algorytmie, może pozwolić na jego znaczącą optymalizację (zarówno pod względem czasu działania, jak i zajętości pamięci operacyjnej). Będziemy przyglądać się bliżej tym zagadnieniom omawiając poszczególne zadania.

17.3 Histogramy i zliczanie

W zadaniach algorytmicznych typowym zadaniem kontenerów sekwencyjnych jest przechowywanie danych wejściowych i/lub wyjściowych, jak widzieliśmy to na przykładzie zadań w Scratchu. Często jednak konieczne jest utworzenie dodatkowych list lub tablic do przechowywania wyników obliczeń pośrednich. Prostym zastosowaniem tablicy jest np. zliczanie liczby wystąpień elementów w danych wejściowych – podobnie jak robiliśmy to w zadaniu „Przebieranki Małgosi” z poprzedniego działu (mieliśmy tam aż trzy tablice do zliczania poszczególnych części garderoby).

Rozważmy przykład, w którym na wejściu otrzymujemy kilka tysięcy liczb oznaczających wiek użytkowników i jesteśmy zainteresowani ilu z nich ma mniej niż 10 lat, ilu mieści się w przedziale 10 – 20, ilu w przedziale 20 – 30, itd. Możemy to łatwo policzyć, tworząc tablicę, której pierwszym elementem będzie licznik osób o wieku z przedziału 0 – 10 lat, drugim – licznik osób z przedziału 10 – 20 lat, itd. Taką „tablicę liczników” nazywamy *histogramem*.

Użycie histogramu może nam pozwolić na znaczące przyspieszenie działania programu w przypadku niektórych zadań. Konieczność wielokrotnego przeglądania całej listy danych wejściowych można bowiem niekiedy zastąpić jednorazowym jej „przejściem” w celu zbudowania histogramu, który będzie od niej dużo krótszy, a więc szybszy do przetworzenia. W naszym przykładzie z wiekiem użytkowników, histogram będzie mieć zaledwie 10 elementów (zakładając, że nie ma osób starszych niż sto lat). Użycie krótszych przedziałów nieco go wydłuży

(np. podział użytkowników na grupy 0-5 lat, 5-10 lat, ..., 95-100 lat, da histogram 20-elementowy), ale nawet jeśli przypiszemy osobną pozycję histogramu do każdej możliwej liczby lat użytkownika (1 rok, 2 lata, 3 lata, ..., 100 lat), taki histogram (100-elementowy) może być nadal znacząco krótszy niż lista wejściowa. Jeśli zatem uda się skonstruować algorytm w oparciu o przetwarzanie histogramu zamiast listy wejściowej, to możemy istotnie zredukować złożoność obliczeniową naszego rozwiązania. Oczywiście nie w każdym zadaniu się uda – wystarczy, żeby np. kolejność osób na liście wejściowej miała jakieś znaczenie, a wówczas nasz histogram okaże się bezużyteczny (bo – z założenia – pozwala tylko zliczać ile było osób w danym przedziale wiekowym, ignorując informację o ich pozycji na liście wejściowej).

17.4 Stosy i kolejki

Oprócz list, czy tablic ogólnego przeznaczenia, wykorzystuje się często wyspecjalizowane struktury danych o mocno zredukowanym zbiorze operacji, ale zaimplementowanych w efektywny sposób. Najczęściej stosowanymi strukturami tego typu są:

- stos;
- kolejka.

Stos (ang. *stack*) jest strukturą danych, składającą się z liniowo uporządkowanych elementów (tak jak w liście, czy tablicy), ale w której tylko ostatnio dodany element (znajdujący się na „wierzchołku" stosu) jest widoczny. Możemy wyobrazić sobie ten sposób organizacji danych w postaci np. stosu talerzy, czy książek. Po odłożeniu elementu na stos, elementy położone głębiej nie są bezpośrednio dostępne (aby dotrzeć do danego elementu trzeba najpierw pozdejmować to, co na nim „leży"). Stos określamy zatem jako strukturę typu LIFO (ang. *last in, first out* – ostatni, który wszedł wychodzi jako pierwszy).

Stos jest bardzo ważną strukturą danych używaną m.in. przez system operacyjny do alokacji pamięci dla programu podczas jego wykonania (przykładowo: podczas wywołania funkcji, na stosie przechowywane są jej zmienne lokalne). Stos jest również przydatny w wielu różnych problemach algorytmicznych (w szczególności do implementacji algorytmów o charakterze rekurencyjnym[1]).

Podstawowe operacje na stosie to:

- położenie elementu na stosie (ang. *push*);
- pobranie elementu ze stosu (ang. *pop*).

Dodatkowo przydatne mogą być również inne operacje, jak np. sprawdzenie czy stos jest pusty lub odczytanie wierzchołka bez pobierania go (ang. *peek* lub *top*).

Rozważmy poniższy przykład:

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
```

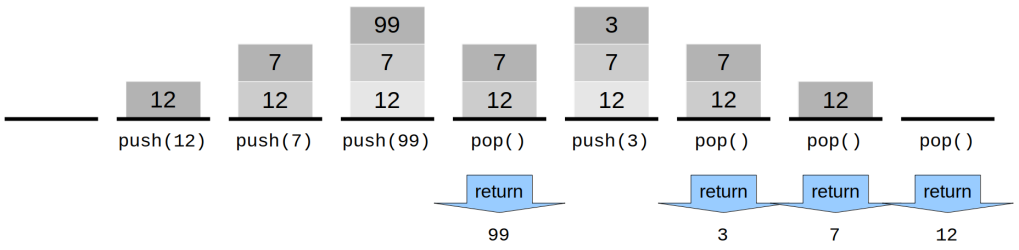
[1] *Rekurencja* albo in. *rekursja* to najprościej mówiąc sytuacja, w której podprogram lub funkcja wywołuje „samą siebie" – o tej ważnej technice konstruowania algorytmów, z której będziemy tu niejednokrotnie korzystać, przeczytasz więcej m.in. w [4][7][22].

```
    stack<int> S;
    S.push(12);
    S.push(7);
    S.push(99);
    cout << S.top() << endl;
    S.pop();
    S.push(3);
    cout << S.top() << endl;
    S.pop();
    cout << S.top() << endl;
    S.pop();
    cout << S.top() << endl;
    S.pop();
    return 0;
}
```

Uzyskamy tu na wyjściu ciąg wartości:

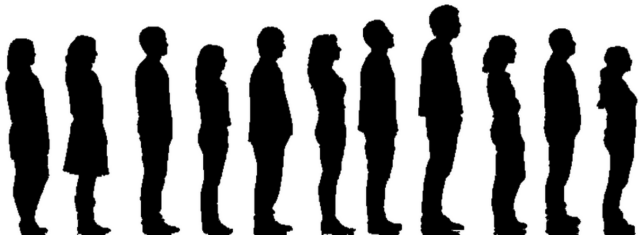
```
99
3
7
12
```

Stan stosu S po wykonaniu kolejnych operacji przedstawiono na poniższym rysunku:



Warto zauważyć, że gdybyśmy najpierw położyli kolejno wszystkie wartości na stosie, a potem je po kolei zdejmowali, to uzyskalibyśmy kolejność odwrotną niż oryginalna.

Inaczej jest w przypadku kolejki (ang. *queue*), w której dodawanie nowych elementów możliwe jest tylko z jednej strony, a usuwanie – tylko z drugiej. Kolejkę określamy zatem jako strukturę typu FIFO (ang. *first in, first out* – pierwszy, który wszedł wychodzi jako pierwszy).



Warto podkreślić, że – w zależności od rozwiązywanego problemu algorytmicznego – wykorzystanie odpowiedniego rodzaju kontenera i kolejności dostępu do jego elementów może stanowić zasadniczy element skutecznego rozwiązania.

17.5 Inne struktury danych

Omówione wyżej przykłady kontenerów sekwencyjnych nie wyczerpują oczywiście wszystkich możliwości w zakresie przechowywania danych, jakie oferują współczesne języki programowania. W tej sekcji przedstawimy pokrótce kilka innych struktur danych, które okażą się bardzo pomocne do rozwiązywania niektórych zadań w tym i w kolejnych działach.

Jednym z bardzo przydatnych, standardowych typów danych w Pythonie jest słownik (`dict`, od ang. *dictionary*), pozwalający na dostęp do elementów nie według ich numerów (indeksów – jak w zwykłej tablicy), a według *kluczy*, które mogą być np. napisami, czy innymi typami danych[2]. To pozwala na bardzo wygodny i naturalny sposób zapisania pewnych operacji w kodzie programu. Przykładowo, jeśli chcemy powiązać imię użytkownika z jego wzrostem, możemy zrobić to tak:

```
wzrost = {"Tomek": 172, "Piotrek": 178, "Bartek": 176, "Magda": 175}
name = input("Podaj imię: ")
print(wzrost[name])
```

Tutaj `wzrost` jest właśnie typu słownikowego, co poznajemy po użyciu nawiasów klamrowych i specyfikacji elementów za pomocą składni `klucz: wartość`. Imię podane po uruchomieniu programu może być bezpośrednio użyte do odczytania wartości wzrostu ze słownika (ostatni wiersz).

Dla porównania, analogiczny program zapisany przy użyciu list mógłby wyglądać tak:

```
imiona = ["Tomek", "Piotrek", "Bartek", "Magda"]
wzrost = [172, 178, 176, 175]
name = input("Podaj imie: ")
idx = imiona.index(name) # znajdź pozycję elementu
print(wzrost[idx])
```

Jak widać, musimy wykonać tu dodatkową operację sprawdzenia, na której pozycji znajduje się podane imię (funkcja `index` wyszukuje pierwsze wystąpienie elementu na liście). Jest to nie tylko mniej wygodne, ale może też być znacząco mniej wydajne, w przypadku dużej liczby elementów[3].

Ujmując ciąg elementów w nawiasy klamrowe zamiast kwadratowych (jak w przypadku list), czy okrągłych (jak w przypadku krotek), ale nie używając dwukropków (jak w przypadku słowników), definiujemy jeszcze inny typ danych zwany *zbiorem* (ang. *set*). Różni się on od poprzednio omawianych tym, że jest nieuporządkowany, tzn. jego elementy nie mają określonej kolejności, a także nie mogą się powtarzać. Odpowiada to pojęciu *zbioru*

[2] Ściślej rzecz biorąc, klucze w słowniku muszą być typu niemodyfikowalnego (ang. *immutable*), takiego jak np. liczba, napis, lub krotka.

[3] Słowniki do szybkiego wyszukiwania elementów używają zwykle tzw. *funkcji skrótu* (ang. *hash*), o której powiemy nieco więcej w dziale VI przy okazji omawiania algorytmów tekstowych.

w rozumieniu matematycznym. Również podstawowe operacje takie jak suma, różnica, iloczyn (przecięcie) zbiorów, czy sprawdzanie przynależności do zbioru są zaimplementowane w sposób zgodny z ich matematycznym pierwowzorem. Praktyczne zastosowanie zbioru w językach Python i C++ zobaczymy już w rozwiązaniu pierwszego zadania w tym dziale: „Czapki i rękawiczki".

Na koniec należy wspomnieć o pewnym bardzo ważnym pojęciu, obecnym zarówno w języku Python, jak i w C++, jakim jest *klasa* (ang. *class*). Klasę możemy traktować – w dużym uproszczeniu – jako formę „grupowania" zmiennych, przy czym zmienne te mają zwykle ściśle określony typ i znaczenie, jednoznacznie ustalone w definicji klasy. Przykładowo, w języku C++ możemy zdefiniować klasę reprezentującą punkt na płaszczyźnie, zawierającą trzy zmienne (pola składowe klasy) – współrzędne x, y i kolor punktu:

```
class Punkt {
public:
    int x;
    int y;
    char color;
};
```

Klasa `Punkt` stanowi w gruncie rzeczy nasz własny, nowy typ danych. Możemy zatem tworzyć obiekty tego typu, dodawać je do kolekcji (list, tablic, wektorów, itd.) oraz odwoływać się do ich pól składowych za pomocą operatora „." (kropki):

```
Punkt p1;
p1.x = 10;
p1.y = 20;
p1.color = 'r';
vector<Punkt> punkty;
punkty.push_back(p1);
cout << punkty[0].x << endl;
```

Użycie klas stanowi fundament zupełnie osobnego *paradygmatu* programowania – tzw. *programowania obiektowego* [19], niezwykle przydatnego zwłaszcza do tworzenia dużych aplikacji i systemów informatycznych[4]. Jednak nawet w niewielkich programach, jakie typowo piszemy w celu rozwiązania naszych problemów algorytmicznych, możliwość definiowania własnych klas i tworzenia obiektów na ich podstawie niekiedy bardzo się przydaje. Warto pamiętać, że w języku C++ zamiast klas możemy również definiować nieco prostsze pod pewnymi względami *struktury* (za pomocą słowa kluczowego `struct`). Przykładowo, pojedynczy węzeł listy wiązanej (dwukierunkowej) możemy zdefiniować tak:

```
struct Wezel {
    int wartosc;
    Wezel* nastepny;
    Wezel* poprzedni;
};
```

[4] Z programowaniem obiektowym związane jest bardzo wiele specyficznych pojęć i mechanizmów, których nie będziemy tu omawiać.

Oczywiście pole `wartosc` mogłoby być dowolnego typu, w zależności od tego, co chcemy przechowywać w naszej liście, natomiast pozostałe dwa pola są typu „wskaźnik do typu `Wezel`", co pozwala na utworzenie powiązań z innymi węzłami. Dostęp do wartości danego węzła jest bezpośredni:

```
Wezel w;
w.wartosc = 12;
```

Możemy również pobrać adres węzła za pomocą operatora `&` i odwoływać się do jego pól za pomocą wskaźnika, naturalnie pamiętając o tym, że użycie wskaźników w C++ wymaga zamiany operatora „." (kropka) na operator „->" (strzałka):

```
Wezel w;
Wezel* wsk = &w;
wsk->wartosc = 15;
```

natomiast dostęp do wartości sąsiednich węzłów zrealizujemy tak:

```
wsk->poprzedni->wartosc = 10;
wsk->nastepny->wartosc = 20;
```

Można zauważyć, że pisanie własnych list wiązanych nie jest zwykle konieczne, z uwagi na dostępność gotowych, standardowych implementacji (oferujących również wygodne mechanizmy dostępu do danych i przechodzenia przez kolejne elementy za pomocą odpowiednich *iteratorów*). W analogiczny sposób można jednak tworzyć bardziej złożone struktury danych, takie jak grafy (p. dział VIII), czy drzewa, o których powiemy sobie więcej w omówieniu ostatniego zadania w tym dziale („Kto stoi przede mną" – wersja dla Gremlinów).

18 (III) – Czapki i rękawiczki – Skrzaty

Autorzy: Marcin Markowski, Politechnika Wrocławska, Przemysław Gordinowicz, Politechnika Łódzka
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

18.1 Treść zadania

Firma odzieżowa produkuje czapki i rękawiczki. Z uwagi na srogą zimę, tegoroczna produkcja zwiększyła się znacznie i firma została zmuszona do wdrożenia specjalnego systemu magazynowego. System ten wymaga, aby każdy produkt w magazynie posiadał niepowtarzalny numer identyfikacyjny (przy czym numer przypisany obu rękawiczkom z danej pary jest oczywiście taki sam). Niestety, na skutek awarii komputera, nazwy wszystkich produktów w magazynie uległy skasowaniu – zostały same numery. Czy jesteś w stanie stwierdzić, ile par rękawiczek znajduje się w magazynie, znając tylko numery identyfikacyjne produktów?

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekt swojego działania, czyli liczbę par takich samych numerów znajdujących się na liście `DANE`.

Opis wejścia

Lista wejściowa `DANE` zawiera n liczb ($2 \leq n \leq 100\,000$) – są to numery identyfikacyjne poszczególnych produktów w magazynie. Każdy numer identyfikacyjny jest liczbą z zakresu od 0 do $100\,000$ i może pojawić się na liście najwyżej dwukrotnie.

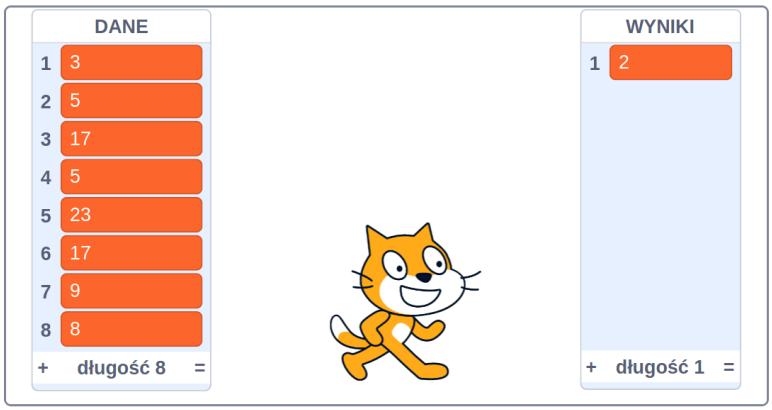
Opis wyjścia

Twój program po uruchomieniu (za pomocą zielonej flagi) powinien umieścić na liście `WYNIKI` jeden element – liczbę par takich samych liczb znajdujących się na liście wejściowej.

Przykład

Na kolejnym rysunku pokazano przykładowy wynik działania programu.

Jak łatwo zauważyć, na liście znajdują się dwie pary takich samych liczb – są to liczby 5 i 17.

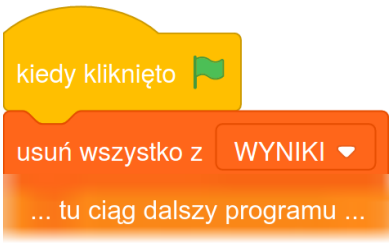


Rysunek 18.1: Przykładowe dane wejściowe i wynik działania skryptu

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

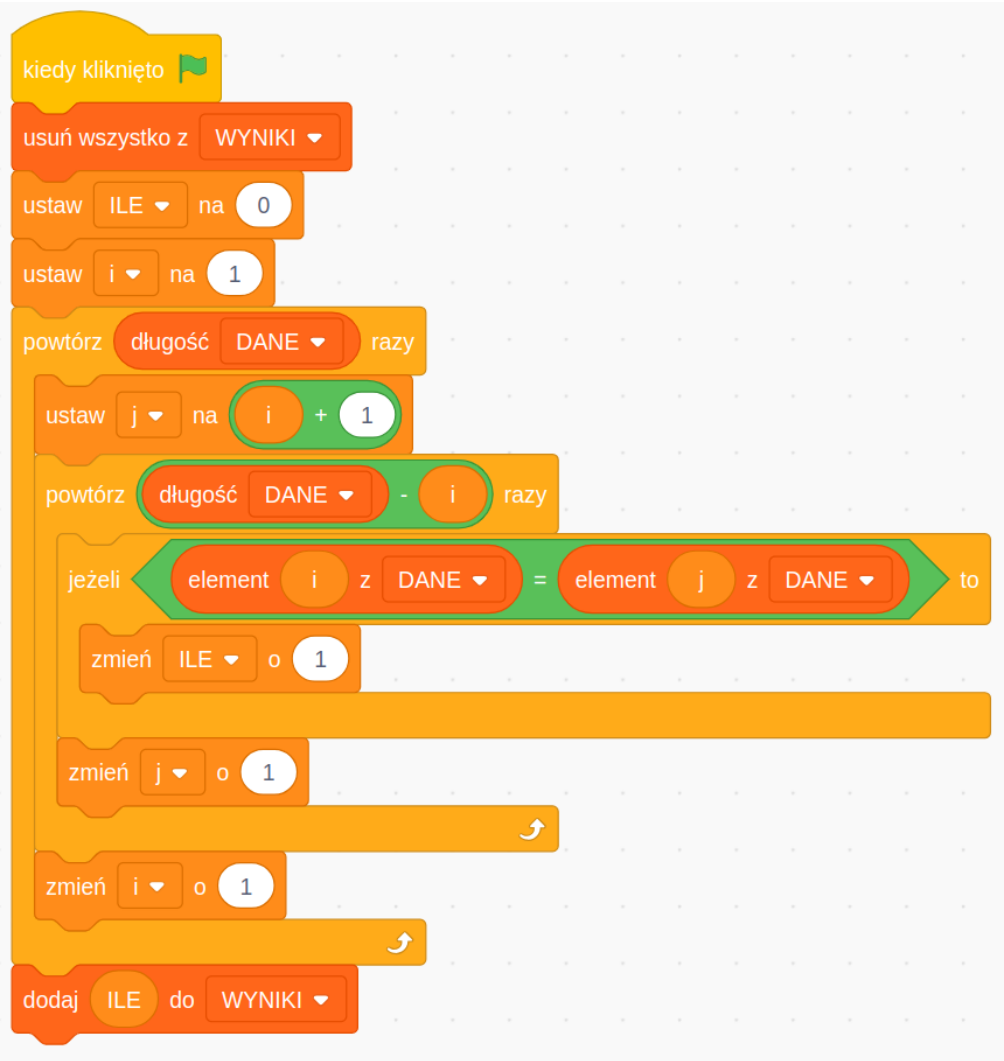
Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



18.2 Rozwiązanie

To zadanie wygląda na proste i w istocie – takie jest. Na jego przykładzie możemy jednak pokazać kilka ciekawych sposobów rozwiązania w oparciu o różne struktury danych.

Pomyślmy jak sami rozwiązalibyśmy ten problem bez pomocy komputera. Możemy np. wziąć pierwszy numer identyfikacyjny z listy i przeszukać jej dalszą część, sprawdzając, czy znajdziemy gdzieś jego „duplikat". Następnie bierzemy drugi identyfikator z listy i robimy to samo. Potem trzeci, czwarty i tak dalej. Zapisując ten pomysł w Scratchu otrzymalibyśmy taki oto program:



Rysunek 18.2: Rozwiązanie naiwne

Zmienna `i` oznacza tutaj numer bieżącego identyfikatora, a zmienna `j` – numer identyfikatora, który z nim aktualnie porównujemy.

Jest to jednak rozwiązanie *siłowe* (naiwne), bo – chociaż skuteczne – ma jednak zbyt dużą złożoność obliczeniową. Ta złożoność to $O(n^2)$, bo dla każdego elementu listy, których jest n, wykonujemy porównania z pozostałymi, których jest też „prawie n" (w powyższym programie zauważmy dwie pętle zagnieżdżone jedna w drugiej). Ściśle rzecz biorąc, dla danego elementu listy wystarczy wykonać porównania tylko z następującymi po nim, bo wszystkie elementy poprzedzające go już zostały sprawdzone w poprzednich krokach (dlatego

w naszym programie licznik `j` drugiej pętli „startuje” od `i+1`). A zatem, dla pierwszego identyfikatora na liście wykonamy $n-1$ porównań, dla drugiego: $n-2$, itd. aż do ostatniego, dla którego liczba porównań wynosi zero (bo został już porównany ze wszystkimi wcześniejszymi elementami).

Ile jest wszystkich porównań? Ze wzoru na sumę n pierwszych wyrazów ciągu arytmetycznego mamy:

$$S = \frac{a_1 + a_n}{2} \cdot n = \frac{(n-1)+0}{2} \cdot n = \frac{n \cdot (n-1)}{2} \,. \tag{18.1}$$

Jak zatem widać, liczba porównań jest ponad dwukrotnie mniejsza niż n^2, jednak nie ma to wpływu na klasę złożoności obliczeniowej, która wciąż wynosi $O(n^2)$. Jeśli rozważymy listę o długości $n = 100\,000$ (zgodnie z treścią zadania, jest to maksymalna możliwa liczba elementów), to nie ma większego znaczenia, czy porównań ma być $n^2 = 10\,000\,000\,000$, czy połowę tej wartości – i tak nie doczekamy się wyniku w żadnym skończonym czasie. Jedynym wyjściem jest znalezienie sposobu zmniejszenia klasy złożoności obliczeniowej.

Z pomocą znów przyjdzie nam nasza intuicja. Mając listę dłuższą niż kilka-kilkanaście elementów, moglibyśmy wpaść na pomysł, żeby wziąć kartkę i przeglądając kolejne identyfikatory, zapisywać na niej każdy z nich w jakiś uporządkowany sposób, np. w kolejności rosnącej. Utworzylibyśmy w ten sposób drugą listę złożoną z tych samych identyfikatorów, ale posortowaną. W praktyce oznaczałoby to tyle, że biorąc każdy następny identyfikator z listy wejściowej musielibyśmy wstawić go w odpowiednie miejsce na kartce, tak aby zachować kolejność rosnącą. Znalezienie tego „odpowiedniego miejsca” byłoby już względnie łatwe (bo identyfikatory na kartce są posortowane), a niejako przy okazji dostalibyśmy rozwiązanie naszego problemu (bo jeśli „odpowiednie miejsce” jest już zajęte, to znaczy, że znaleźliśmy „duplikat”).

Kwestia wstawiania elementu do posortowanej listy jest, generalnie, problemem o złożoności logarytmicznej, czyli $O(\log n)$. Wynika to stąd, że wstawiany element porównujemy najpierw z elementem w środku listy i jeśli jest od niego większy, to pierwszą połowę listy od razu możemy odrzucić (jeśli mniejszy – to odrzucamy drugą). Z pozostałą połową robimy tak samo, tzn. porównujemy wstawiany element z jej środkiem i znów redukujemy listę o połowę, itd. Liczba elementów pozostałych po pierwszym porównaniu wynosi więc $n \cdot \frac{1}{2}$, po drugim: $n \cdot \frac{1}{2} \cdot \frac{1}{2}$, czyli $n \cdot (\frac{1}{2})^2$, po trzecim: $n \cdot (\frac{1}{2})^3$, itd. Algorytm kończy się, gdy po p porównaniach pozostanie nam już tylko jeden element:

$$n \cdot \left(\frac{1}{2}\right)^p = 1 \,. \tag{18.2}$$

Dzieląc obie strony tego wyrażenia przez n mamy:

$$\left(\frac{1}{2}\right)^p = \frac{1}{n} \,, \tag{18.3}$$

i równoważnie:

$$2^p = n \,. \tag{18.4}$$

Stąd więc widać natychmiast, że liczba porównań $p = \log_2(n)$.

W praktyce, wartość ta jest przybliżona, co wynika m.in. z faktu, że w powyższym rozumowaniu pojęcia „środka” i „połowy” listy są nieco umowne (należałoby tu uwzględnić czy bieżący fragment listy ma parzystą, czy nieparzystą liczbę elementów). Niemniej jednak mamy tu zasadniczo do czynienia z procedurą o złożoności $O(\log n)$, czyli logarytmicznej[1]. Biorąc pod uwagę, że w naszym rozwiązaniu musimy powtórzyć ją dla każdego z n elementów, otrzymujemy ostateczną złożoność $O(n \log n)$ (liniowo-logarytmiczną).

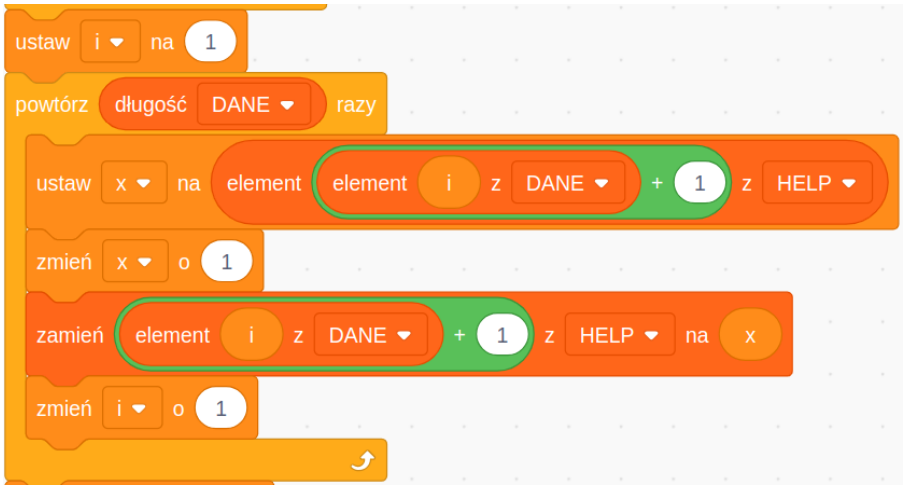
[1] W ogólnym przypadku może to nie być do końca zgodne z prawdą, gdyż sama operacja *wstawienia* elementu może wymagać dodatkowego narzutu czasowego, w zależności od rodzaju użytej struktury danych.

Można zauważyć, że tu również, podobnie jak w rozwiązaniu naiwnym, faktyczna liczba porównań jest nieco mniejsza, z uwagi na fakt, że nasza posortowana lista na początku jest pusta i wydłuża się stopniowo, a dopiero pod koniec algorytmu jej długość zbliża się do n. To nie ma jednak wpływu na klasę złożoności obliczeniowej.

Pytanie, czy złożoność obliczeniową można jeszcze bardziej zredukować w tym zadaniu? Okazuje się, że tak. W tym celu należy wykorzystać podejście oparte na zliczaniu ilości wystąpień poszczególnych elementów za pomocą *histogramu* (p. wprowadzenie do tego działu). Ta metoda jest w pewnym sensie uproszczoną wersją poprzedniej (choć może wymagać użycia większej ilości pamięci). Uproszczoną, bo zamiast pracowicie szukać odpowiedniego miejsca do wstawienia kolejnego elementu, tu *od razu* na początku algorytmu wstawiamy *wszystkie możliwe* elementy, ale zaznaczając przy tym, że każdy z nich wystąpił na razie zero razy. W praktyce, tworzymy po prostu listę pomocniczą (nasz histogram – tu o nazwie `HELP`) zawierającą początkowo same zera:

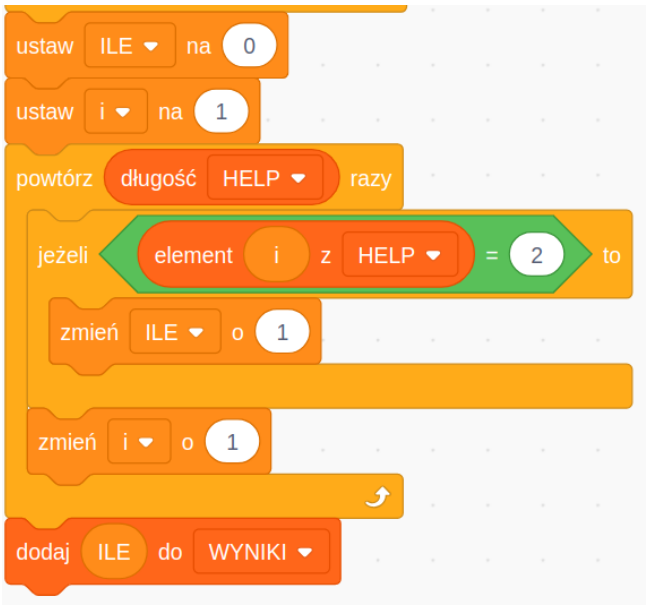
Rysunek 18.3: Inicjalizacja histogramu

Jak pamiętamy, histogram jest w istocie „listą liczników”, a więc jego pierwszy element będzie oznaczać liczbę wystąpień identyfikatora „1”, drugi – liczbę wystąpień identyfikatora „2”, itd. W głównej części naszego algorytmu, przechodzimy przez kolejne elementy listy wejściowej i dla każdego z nich inkrementujemy (zwiększamy o jeden) odpowiedni element histogramu:



Rysunek 18.4: Budowanie histogramu (ciąg dalszy programu)

Przykładowo, dla identyfikatora równego 5, inkrementujemy piąty element histogramu. Zauważmy tu kluczowy fakt, że nie musimy niczego szukać – wartość identyfikatora jest po prostu numerem elementu histogramu do inkrementacji. Wynika stąd, że otrzymany algorytm ma złożoność $O(n)$ (liniową), bo wystarczy nam pojedyncze przejście przez elementy listy wejściowej, aby zbudować histogram. Histogram z kolei mówi nam, ile razy wystąpił identyfikator „1”, ile razy identyfikator „2”, itd., a zatem wystarczy nam na koniec tylko jeden raz przejść przez jego kolejne elementy, aby dowiedzieć się, ile identyfikatorów wystąpiło dwukrotnie:



Rysunek 18.5: Podsumowanie (ciąg dalszy programu)

Jak zatem widać, wykorzystanie histogramu może pozwolić na szybkie rozwiązanie pewnych problemów, choć należy tu zwrócić uwagę na pewien – potencjalnie istotny – szczegół. Otóż histogram ma zawsze tyle elementów, ile *możliwych wartości* może pojawić się na liście wejściowej. Jeśli liczba tych możliwych wartości (tu: identyfikatorów) przekracza znacząco długość tej listy, to użycie histogramu może nie być zasadne. Przykładowo, gdyby liczba produktów w magazynie była ograniczona do 100, ale identyfikatorem mogła być dowolna liczba całkowita od 1 do stu miliardów, to nasz histogram musiałby mieć długość stu miliardów, a więc – po pierwsze – nie zmieściłby się zapewne w pamięci, a – po drugie – ze 100-elementową listą wejściową poradzilibyśmy sobie dużo szybciej bezpośrednio.

Przedstawione tu rozwiązanie znajdziesz w pliku:

./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Skrzaty/solution.sb3.

Dla porównania, pokazane wcześniej rozwiązanie naiwne znajduje się w pliku:

./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Skrzaty/solutionNaive.sb3.

Zanim przejdziemy do kolejnych zadań, pokażemy najpierw – zgodnie z zapowiedzią we wprowadzeniu do niniejszego działu – jak rozwiązać problem „Czapek i rękawiczek” w językach C++ i Python. Problem jest oczywiście ten sam, ponieważ jednak mamy tutaj odmienny sposób wczytywania i wypisywania danych, podamy więc jeszcze raz treść zadania – tym razem w wersji dla Gnomów (podobnie jak zrobiliśmy to w przypadku zadania „Tam i z powrotem”).

19 (III) – Czapki i rękawiczki – Gnomy

Autorzy: Marcin Markowski, Politechnika Wrocławska, Przemysław Gordinowicz, Politechnika Łódzka
Kategoria: Gnomy (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

19.1 Treść zadania

Firma odzieżowa produkuje czapki i rękawiczki. Z uwagi na srogą zimę, tegoroczna produkcja zwiększyła się znacznie i firma została zmuszona do wdrożenia specjalnego systemu magazynowego. System ten wymaga, aby każdy produkt w magazynie posiadał niepowtarzalny numer identyfikacyjny (przy czym numer przypisany obu rękawiczkom z danej pary jest oczywiście taki sam). Niestety, na skutek awarii komputera, nazwy wszystkich produktów w magazynie uległy skasowaniu – zostały same numery. Czy jesteś w stanie stwierdzić, ile par rękawiczek znajduje się w magazynie, znając tylko numery identyfikacyjne produktów?

Zadanie

Otrzymasz listę numerów identyfikacyjnych produktów znajdujących się w magazynie firmy, podanych w losowej kolejności (przy czym każda czapka i każda rękawiczka stanowi osobną pozycję). Na tej podstawie należy obliczyć liczbę par rękawiczek, czyli liczbę par takich samych numerów znajdujących się na podanej liście.

Opis wejścia

W pierwszym wierszu wejścia podana jest liczba n (przy czym $2 \leq n \leq 1\,000\,000$) oznaczająca liczbę produktów na liście. W każdym z kolejnych n wierszy znajduje się jedna liczba całkowita – numer identyfikacyjny produktu. Każdy numer identyfikacyjny jest liczbą z zakresu od 0 do $1\,000\,000\,000$ i może pojawić się na liście najwyżej dwukrotnie.

Opis wyjścia

Twój program powinien wypisać na standardowe wyjście jedną liczbę całkowitą, stanowiącą liczbę par takich samych liczb znajdujących się na liście wejściowej.

Przykład

Dla przykładowego wejścia:

```
8
3
5
17
```

```
5
23
17
9
8
```

prawidłowy wynik to:

```
2
```

Lista ma długość 8 elementów i – jak łatwo zauważyć – znajdują się na niej dwie pary takich samych liczb (5 oraz 17).

19.2 Rozwiązanie

Pomimo innego języka programowania i innego sposobu podawania danych wejściowych/wyjściowych, sam algorytm rozwiązania będzie oczywiście taki sam. Przykładowo, rozwiązanie naiwne w Pythonie może wyglądać np. tak:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Gnomy/solutionNaive.py.

```
1  n = int(input())
2  lista = list()
3  for i in range(n):
4      id = int(input())
5      lista.append(id)
6
7  pairs = 0
8  for i in range(n):
9      id = lista[i]
10     for j in range(i + 1, n):
11         if lista[j] == id:
12             pairs += 1
13 print (pairs)
```

Widzimy tu tę samą charakterystyczną konstrukcję złożoną z dwóch pętli zagnieżdżonych jedna w drugiej (wiersz 8 i 10). Również zakresy obu zmiennych `i` i `j`, oraz sposób porównywania identyfikatorów są analogiczne. Jedynie początek programu jest inny, bo nie mamy tu gotowej listy `DANE` z identyfikatorami, tylko musimy ją sami utworzyć (wiersz 2) i wczytać identyfikatory ze standardowego wejścia, wymuszając przy tym ich interpretację jako liczby całkowite (wiersz 4) i dodać po kolei do listy (wiersz 5). Dodatkowy, pierwszy element, wczytywany osobno (wiersz 1) oznacza liczbę wszystkich identyfikatorów, zgodnie ze specyfikacją wejścia w treści zadania dla Gnomów.

Rozwiązanie naiwne w C++ jest analogiczne:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Gnomy/solutionNaive.cpp.

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main() {
7  //  ios_base::sync_with_stdio(false); // Uzycie tych dwoch wierszy kodu pozwoli na
8  //  cin.tie(nullptr);                 // przyspieszenie operacji wczytywania danych
9    int n;
10   cin >> n;
11   vector<int> tab(n);
12   int id;
13   for (int i = 0; i < n; i++) {
14     cin >> id;
15     tab[i] = id;
```

```
16    }
17    int pairs = 0;
18    for (int i = 0; i < n; i++) {
19      int id = tab[i];
20      for (int j = i + 1; j < n; j++) {
21        if (tab[j] == id) pairs++;
22      }
23    }
24    cout << pairs << endl;
25  }
```

Stosujemy tu strukturę o nazwie `vector` zamiast listy i od razu deklarujemy jaki ma być jej rozmiar (co może zwiększyć wydajność, bo program od razu zaalokuje potrzebną ilość pamięci). Również wczytywanie i wypisywanie danych jest charakterystyczne dla C++ (p. wprowadzenie do niniejszego działu). Warto również zwrócić uwagę na dwie „zakomentowane" linie kodu, tzn. poprzedzone symbolem komentarza `//` (a więc nie uwzględniane przez kompilator). Nie wgłębiając się szczegóły, warto pamiętać, że w zadaniach wymagających wczytania wyjątkowo dużych zbiorów danych, użycie tych dwóch linii (wystarczy je „odkomentować", czyli usunąć znaki `//`) może znacząco przyspieszyć wykonanie programu.

Oczywiście rozwiązanie naiwne nie zadowala nas w pełni. Moglibyśmy więc zaimplementować rozwiązanie szybkie, oparte na histogramie – podobnie jak zrobiliśmy to w Scratchu. Przyjrzyjmy się tu jednak wspomnianej wcześniej metodzie pośredniej, w której każdy kolejny identyfikator „zapisujemy na kartce" w pewien uporządkowany sposób. Nie implementowaliśmy tej metody w Scratchu, z uwagi na dość kłopotliwe wyszukiwanie „odpowiedniego miejsca" do wstawienia nowego elementu. Tu również nie będziemy jej implementować, bo... ktoś zrobił to już za nas!

Spójrzmy na następujący kod w Pythonie:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Gnomy/solution.py.

```
1  n = int(input())
2  zbior = set()
3  for i in range(n):
4      id = int(input())
5      zbior.add(id)
6
7  print(n - len(zbior))
```

Jest on bardzo zwięzły i – choć może trudno w to uwierzyć – to jest właśnie kompletne rozwiązanie zadania „Czapki i rękawiczki". Jak to działa? Wykorzystujemy tu gotową strukturę danych o nazwie `set`, czyli *zbiór* (p. wprowadzenie do tego działu). Do zbioru dodajemy kolejne identyfikatory funkcją `add` (wiersz 5), ale wykorzystujemy tu w sprytny sposób fakt, że zbiór liczb całkowitych nie może zawierać duplikatów (np. liczba 5 może albo należeć do zbioru albo do niego nie należeć, ale nie może do niego należeć dwu-, czy trzykrotnie). Funkcja `add` dokonuje tu więc najpierw niejawnie sprawdzenia czy dodawany element `id` nie należy już przypadkiem do zbioru i – jeśli należy – kończy działanie nie dodając go ponownie. Po wczytaniu wszystkich elementów nasz zbiór będzie zatem zawierał tylko niepowtarzalne identyfikatory (bez duplikatów). Odejmując rozmiar tego zbioru (zwracany przez funkcję `len` – od ang. *length*, czyli „długość") od liczby `n` *wszystkich*

identyfikatorów podanych na wejście (a więc z duplikatami), otrzymujemy oczywiście samą liczbę duplikatów, czyli oczekiwany wynik.

Analogiczne rozwiązanie w C++ przedstawiamy poniżej:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/CzapkiIrekawiczki_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <set>

using namespace std;

int main() {
//  ios_base::sync_with_stdio(false);
//  cin.tie(nullptr);
  int n;
  cin >> n;
  set<int> zbior;
  int id;
  for (int i = 0; i < n; i++) {
    cin >> id;
    zbior.insert(id);
  }
  cout << n - zbior.size() << endl;
}
```

Na koniec zapytajmy o złożoność obliczeniową naszego rozwiązania. Zależy ona oczywiście od tego w jaki sposób funkcja wstawiająca nowy identyfikator do zbioru sprawdza, czy nie został on już wcześniej dodany. Możemy tu jednak założyć, że jest to realizowane w efektywny sposób, czyli podobnie, jak omówiliśmy to podając rozwiązanie dla Scratcha, a zatem złożoność całego rozwiązania nie będzie gorsza niż liniowo-logarytmiczna.

20 (III) – Poczekalnia – Gnomy

*Autor: **Jan Filipowicz**, **Politechnika Łódzka***
*Kategoria: **Gnomy (I edycja – zawody algorytmiczne – etap lokalny)***
*Język programowania: **C++/Python***

20.1 Treść zadania

Przychodnia lekarska w Bajtocji przyjmuje codziennie wielu pacjentów. W poczekalni ważny jest porządek wśród oczekujących, dlatego każdy pacjent zaraz po wejściu otrzymuje numerek i ustawia się na końcu kolejki. Pierwszy pacjent danego dnia otrzyma numer 1, kolejny numer 2, itd. Niektóre przypadki są jednak pilniejsze od innych – stąd wyróżnia się pacjentów zwykłych oraz priorytetowych. Jeżeli doktor prosi o wejście kolejnej osoby, a w kolejce znajdują się jacykolwiek pacjenci priorytetowi, to pacjenci zwykli mają obowiązek ich przepuścić. Oczywiście w przypadku wielu pacjentów priorytetowych, wchodzi ten z nich, który przyszedł wcześniej.
Znając kolejność zdarzeń, które pewnego dnia miały miejsce w bajtockiej przychodni, ustal, jakie były numery kolejnych pacjentów wchodzących do gabinetu lekarskiego.

Opis wejścia

Pierwszy i jedyny wiersz standardowego wejścia zawiera łańcuch n znaków ($n \leq 1\,000\,000$), którego znaki odpowiadają kolejnym zdarzeniom, jakie miały miejsce w przychodni. Łańcuch ten składa się wyłącznie z następujących liter:

- `'Z'` – pacjent zwykły wchodzi do poczekalni;
- `'P'` – pacjent priorytetowy wchodzi do poczekalni;
- `'G'` – doktor przyjmuje do gabinetu pacjenta z kolejki.

Istnieje gwarancja, że zawsze, gdy doktor ma przyjąć pacjenta, w kolejce znajduje się przynajmniej jeden pacjent oraz że wszyscy pacjenci zostaną prędzej czy później przyjęci.

Opis wyjścia

Na standardowe wyjście należy w osobnych wierszach wypisać liczby całkowite oznaczające numery pacjentów w takiej kolejności, w jakiej wchodzili do gabinetu doktora.

Przykład

Dla danych wejściowych:

```
ZPZGGPZPGGGG
```

poprawnym wynikiem jest:

```
2
1
4
6
3
5
```

Wyjaśnienie przykładu

W kolejce najpierw ustawiają się pacjenci o numerach 1, 2, 3, przy czym pacjent 2 posiada priorytet. Gdy doktor zaczyna wpuszczać pacjentów, jako pierwszy wchodzi więc pacjent priorytetowy 2, a później pacjent zwykły 1. Następnie w kolejce ustawiają się pacjenci 4, 5, 6, z których pacjenci 4 i 6 posiadają priorytet. Wchodzą oni zatem do gabinetu lekarskiego jako pierwsi, a potem w kolejce nie znajduje się już żaden pacjent priorytetowy, więc pacjenci zwykli mogą kolejno odbyć swoje wizyty.

20.2 Rozwiązanie

To zadanie również posiada dwie wersje – dla Skrzatów i dla Gnomów, lecz tym razem zaprezentowaliśmy wersję dla Gnomów jako pierwszą. Wynika to stąd, że w językach Python i C++ mamy gotową strukturę danych do reprezentacji kolejki, która przyda się nam do rozwiązania.

Samo rozwiązanie nie narzuca się tu od razu. Najprostsze wydaje się wczytywanie kolejnych liter ze standardowego wejścia tak długo dopóki nie napotkamy litery `G`. Po napotkaniu `G` musimy cofnąć się do początku naszej listy liter i przeszukać ją ponownie, aby stwierdzić jaki numer ma pierwszy pacjent priorytetowy, który jeszcze nie wszedł do gabinetu i ten numer wypisujemy na wyjście. Jeśli nie znajdziemy żadnego pacjenta priorytetowego do wypisania – powtarzamy analogicznie wyszukiwanie dla pacjentów zwykłych.

Cofanie się do początku i wielokrotne przeszukiwanie listy od razu sugeruje, że nie uzyskamy optymalnej złożoności obliczeniowej. Możemy próbować poprawić sytuację np. zapamiętując pozycję ostatnio przyjętego pacjenta (zarówno priorytetowego, jak i zwykłego), ale istnieje prostsze i skuteczne rozwiązanie oparte na – wspomnianych wyżej – kolejkach.

Należy mianowicie utworzyć *dwie* kolejki – jedną dla pacjentów priorytetowych, a drugą – dla zwykłych i w trakcie wczytywania danych od razu wstawiać pacjentów do odpowiedniej kolejki. Po wczytaniu litery `G` wystarczy sprawdzić, czy kolejka pacjentów priorytetowych jest pusta; jeśli nie – bierzemy z niej pierwszego pacjenta, a jeśli tak – pobieramy pierwszego pacjenta z kolejki pacjentów zwykłych[1]

Poniższy kod w Pythonie realizuje dokładnie tę procedurę:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Poczekalnia_Gnomy/solution.py.

```
from collections import deque

s = input()
normal = deque()
priority = deque()
number = 1
for c in s:
  if c == 'Z':
    normal.append(number)
    number += 1
  if c == 'P':
    priority.append(number)
    number += 1
  if c == 'G':
    if priority:
      print(priority.popleft())
    else:
      print(normal.popleft())
```

[1]Uwaga: nie powinniśmy mylić występującej tu „kolejki pacjentów priorytetowych” z pewną bardzo użyteczną strukturą danych zwaną *kolejką priorytetową* [2][7]. W kolejce priorytetowej każdy element ma przypisaną liczbę określającą jego priorytet (czyli „ważność”) i pobieranie elementów następuje wg priorytetów („najważniejszy” na początku), niezależnie od kolejności w jakiej elementy były dodawane (p. zadanie „Skarbce Franka Cenciaka” z następnego działu). W tym zadaniu natomiast pacjentów dzielimy po prostu na dwie grupy (zwykłych i priorytetowych), ale w obrębie danej grupy kolejność na wyjściu jest taka sama, jak na wejściu.

Używamy tu gotowej struktury danych typu kolejka `deque` (ang. *double-ended queue*) z biblioteki `collections`, która jest zaimplementowana w taki sposób, aby operacje wstawiania i pobierania elementów z początku/końca były jak najszybsze. Tak naprawdę, moglibyśmy osiągnąć to samo za pomocą zwykłej listy (wstawiając pacjentów na jej końcu, a pobierając z początku), ale czas działania naszego programu byłby z pewnością znacznie dłuższy.

Zauważmy jeszcze, że kolejka, której tu używamy jest dwustronna (*double-ended*), więc możemy równie dobrze wstawiać i pobierać elementy zarówno na początku, jak i na końcu. Dla ustalenia uwagi przyjmijmy, że elementy dodajemy „z prawej strony" kolejki funkcją `append`, a pobieramy „z lewej" funkcją `popleft` (alternatywnie, moglibyśmy je wstawiać „z lewej" funkcją `appendleft`, a pobierać z prawej funkcją `pop`).

Przedstawiony kod jest prosty i przejrzysty. Warto w nim zwrócić uwagę na fakt, że dane wejściowe wczytujemy tym razem bez pomocy pętli, jedną instrukcją `input` (bo zawarte są w jednym wierszu) oraz na sposób sprawdzania, czy kolejka pacjentów priorytetowych jest pusta, poprzez bezpośrednie użycie nazwy kolejki `priority` w wyrażeniu warunkowym (wiersz 15).

Rozwiązanie w C++ przedstawia się następująco:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Poczekalnia_Gnomy/solution.cpp.

```
1  #include <iostream>
2  #include <queue>
3  #include <string>
4
5  int main() {
6    std::ios_base::sync_with_stdio(false);
7    std::cin.tie(nullptr);
8    std::string s;
9    std::cin >> s;
10   std::queue<int> normal;
11   std::queue<int> priority;
12   int number = 1;
13   for (int i = 0; i < (int)s.size(); i++) {
14     switch (s[i]) {
15       case 'Z':
16         normal.push(number);
17         number++;
18         break;
19       case 'P':
20         priority.push(number);
21         number++;
22         break;
23       case 'G':
24         if (priority.empty()) {
25           std::cout << normal.front() << '\n';
26           normal.pop();
27         } else {
28           std::cout << priority.front() << '\n';
29           priority.pop();
30         }
31         break;
```

```
32        }
33      }
34      return 0;
35    }
```

Tu również mamy dwie kolejki: dla pacjentów zwykłych (kolejka `normal`) i priorytetowych (`priority`), odpowiednio, przy czym tu także wykorzystujemy gotową strukturę kolejki (`queue`). Elementy na koniec kolejki wstawiamy tu funkcją `push`, a pobieramy z jej początku funkcją `pop`, przy czym do samego odczytania wartości elementu znajdującego się na początku kolejki służy osobna funkcja `front`.

21 (III) – Poczekalnia – Skrzaty

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

21.1 Treść zadania

Przychodnia lekarska w Bajtocji przyjmuje codziennie wielu pacjentów. W poczekalni ważny jest porządek wśród oczekujących, dlatego każdy pacjent zaraz po wejściu otrzymuje numerek i ustawia się na końcu kolejki. Pierwszy pacjent danego dnia otrzyma numer 1, kolejny numer 2, itd. Niektóre przypadki są jednak pilniejsze od innych – stąd wyróżnia się pacjentów zwykłych oraz priorytetowych. Jeżeli doktor prosi o wejście kolejnej osoby, a w kolejce znajdują się jacykolwiek pacjenci priorytetowi, to pacjenci zwykli mają obowiązek ich przepuścić. Oczywiście w przypadku wielu pacjentów priorytetowych, wchodzi ten z nich, który przyszedł wcześniej.

Znając kolejność zdarzeń, które pewnego dnia miały miejsce w bajtockiej przychodni, ustal, jakie były numery kolejnych pacjentów wchodzących do gabinetu lekarskiego.

Zadanie

W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `WYNIKI` należy na razie pozostawić pustą. Do listy `DANE` należy wpisać (ręcznie) znaki odpowiadające kolejnym zdarzeniom, jakie miały miejsce w przychodni:

- `Z` – pacjent zwykły wchodzi do poczekalni;
- `P` – pacjent priorytetowy wchodzi do poczekalni;
- `G` – doktor przyjmuje do gabinetu pacjenta z kolejki.

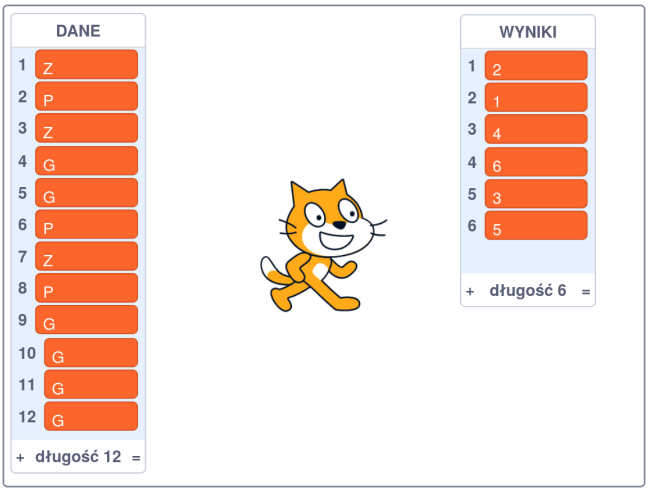
Musi być przy tym spełniony warunek mówiący, że zawsze, gdy doktor ma przyjąć pacjenta, w kolejce znajduje się przynajmniej jeden pacjent oraz że wszyscy pacjenci zostaną prędzej czy później przyjęci.

Celem zadania jest napisanie programu (skryptu), który wpisze do listy `WYNIKI` liczby całkowite oznaczające numery pacjentów w takiej kolejności, w jakiej wchodzili do gabinetu doktora.

Przykład

Przykład przedstawiono poniżej na rysunku 21.1. W kolejce najpierw ustawiają się pacjenci o numerach 1, 2, 3, przy czym pacjent 2 posiada priorytet. Gdy doktor zaczyna wpuszczać pacjentów (litera `G` na pozycji 4 i 5), jako pierwszy wchodzi więc pacjent priorytetowy 2, a później pacjent zwykły 1. Następnie w kolejce ustawiają się

pacjenci 4, 5, 6 (pozycje 6, 7, 8), z których pacjenci 4 i 6 posiadają priorytet. Wchodzą oni zatem do gabinetu lekarskiego jako pierwsi, a potem w kolejce nie znajduje się już żaden pacjent priorytetowy, więc pacjenci zwykli mogą kolejno odbyć swoje wizyty.

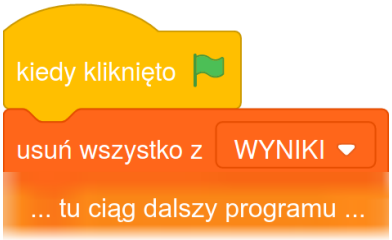


Rysunek 21.1: Przykładowe dane wejściowe i wynik działania skryptu

Uwagi

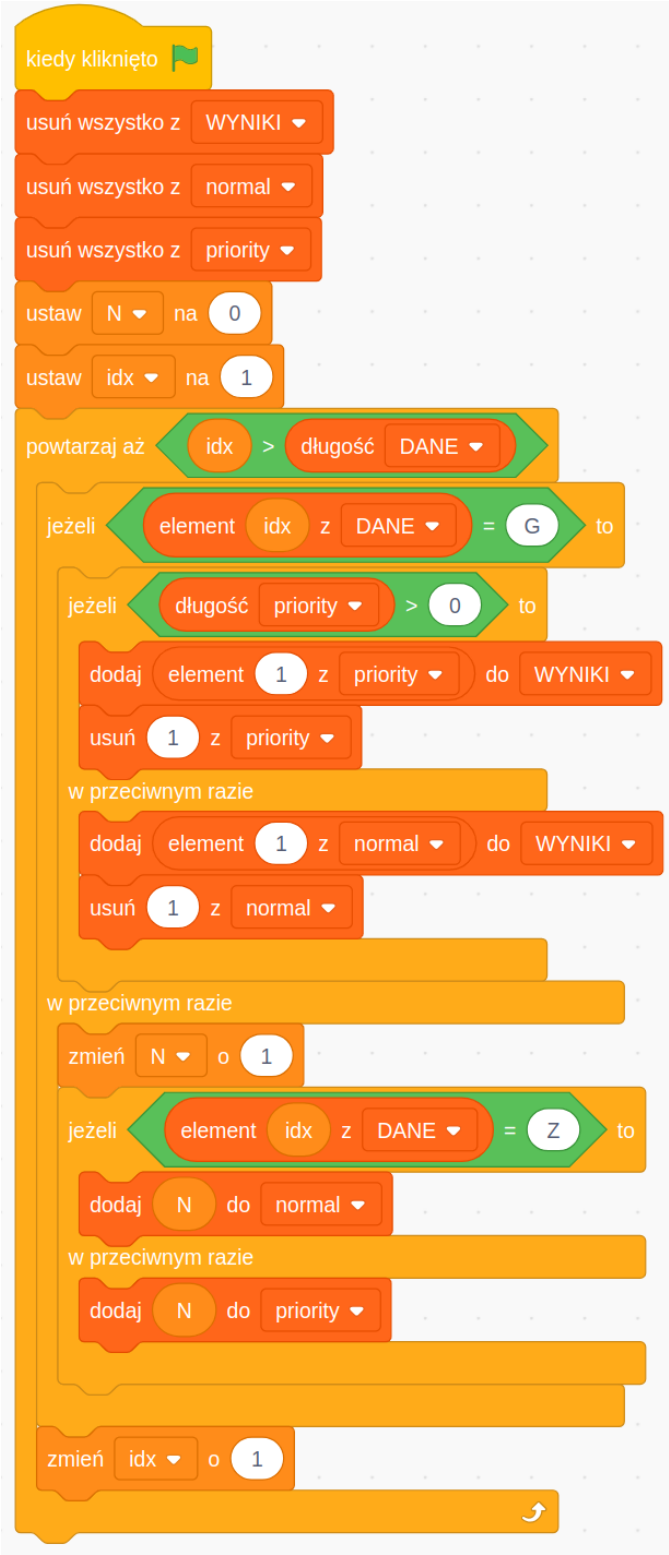
Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



21.2 Rozwiązanie

Implementacja w Scratchu rozwiązania przedstawionego wyżej napotyka pewne trudności: nie dysponujemy tu bowiem gotową strukturą danych typu kolejka. Możemy jednak zaimplementować ją własnoręcznie za pomocą zwykłej listy, tworząc potrzebną nam kolejkę dla pacjentów zwykłych i drugą – dla priorytetowych.



Rysunek 21.2: Zadanie „Poczekalnia” – kod programu

Jak widać, pacjenci dodawani są w naturalny sposób na końcu listy (odpowiednio: `priority` lub `normal`), a pobierani z początku (który jest każdorazowo usuwany). Zmienna N jest odpowiednikiem zmiennej `number` z przykładów w C++/Pythonie i służy oczywiście do przechowywania bieżącego numeru pacjenta.

Przedstawione tu rozwiązanie znajdziesz w pliku:

./zadania/03_Wczytywanie_wypisywanie/Poczekalnia_Skrzaty/solution.sb3.

22 (III) – Antimatter Collider – Gnomy *(zad.w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka

Kategoria: Gnomy (IV edycja – zawody algorytmiczne – etap finałowy)

Język programowania: C++/Python

22.1 Task description

Scientists at the Bytesian Antimatter Collider are studying behavior of subatomic antimatter particles ("antiparticles"). Every time a new experiment is performed, scientists first note down the initial arrangement of particles and antiparticles in a sequence, then start the experiment, and finally compare the practical and theoretical results.

There are 10 types of particles under investigation, along with their corresponding antiparticles – for convenience, the particles are labeled with capital letters "A" to "J" and their corresponding antiparticles are labeled with the same lowercase letters "a" to "j". Whenever a particle is directly next to its corresponding antiparticle with nothing else in between, both the particle and the antiparticle annihilate each other, i.e. suddenly disappear. Other particles move closer together as a result to fill out the empty space.

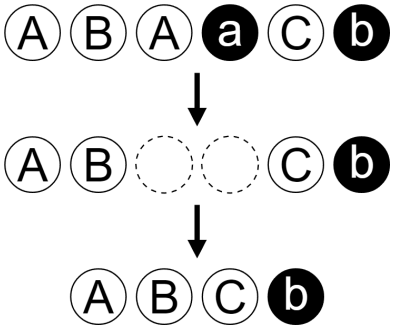


Figure 22.1: Particle annihilation illustrated: in the sequence of particles A B A anti-A, C, anti-B, adjacent particles A and anti-A disappear

Task

Write a program that will find the theoretical final result for a given sequence of particles and antiparticles.

The experiment might take several phases – first some annihilations take place, then the particles move closer together, which causes more annihilations, etc.

Sometimes a particle may be adjacent to two of its corresponding antiparticles (or vice versa) – one on each side. In those cases, the pair that disappears is the one that allows for more annihilations to take place during that phase. For example, the sequence "aAaA" will always disappear instantly (instead of the middle pair "Aa" disappearing during the first phase and the remaining "aA" during the second). If there are multiple ways to achieve the maximum number of annihilations, one of them is chosen at random, e.g. in "BbB" either the first pair "Bb" or the second pair "bB" is chosen – in either case leaving behind "B". Because particles of the same type are identical to each other, this choice never affects the outcome.

Input description

The standard input contains one line – a non-empty string of uppercase letters from `A` to `J` and lowercase letters from `a` to `j`. This is the initial sequence of particles and antiparticles. Its length does not exceed 10 000 000 letters.

Output description

The standard output should contain one line – a string of letters (possibly empty) in the same format as the input, denoting the final result of the experiment – a stable sequence of particles in which no more annihilations may occur.

Example

For sample input:

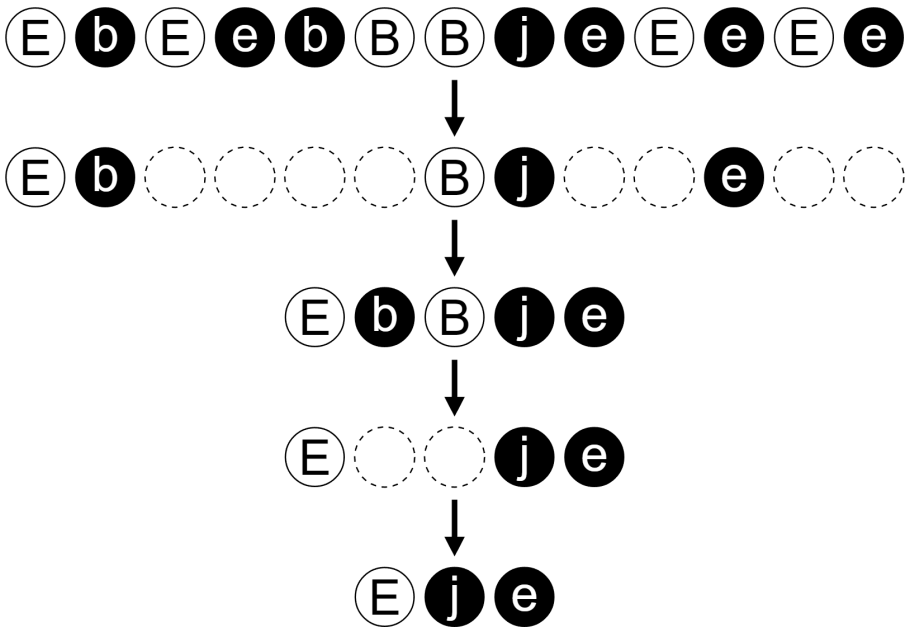
```
EbEebBBjeEeEe
```

the correct output is:

```
Eje
```

Explanation

The experiment has been illustrated below. On the right side, any of the three "e" antiparticles could have remained in the sequence, with identical results.



22.2 Rozwiązanie

W zadaniu należało ustalić oczekiwany wynik eksperymentu, w którym mamy ustawione w sekwencji cząstki i antycząstki różnych rodzajów. W każdej fazie eksperymentu, jeżeli obok siebie znajduje się cząstka i odpowiadająca jej antycząstka, to obie z nich znikają, co może dalej doprowadzić do zderzenia kolejnych dwóch cząstek w kolejnej fazie. Eksperyment toczy się aż do osiągnięcia stabilnego stanu, który należy podać jako odpowiedź.

Niezależnie od podejścia, przeprowadzanie dosłownej symulacji eksperymentu okaże się prawdopodobnie zbyt wolne, by uzyskać pełne punkty za zadanie. Liczba faz eksperymentu może rosnąć liniowo wraz z długością sekwencji – nietrudno sobie na przykład wyobrazić stan początkowy taki jak:

```
AAAAAAAAAAAAAAAAAAAAaaaaaaaaaaaaaaaaaaaa
```

Wówczas zderzenia cząstek z antycząstkami zachodzą wyłącznie w jednym punkcie na środku sekwencji. Eksperyment składałby się z 20 faz, a nasz algorytm symulacyjny w każdej fazie musiałby ten punkt zderzenia ponownie odnaleźć przeszukując sekwencję. Łączny czas wykonania w najgorszym wypadku rósłby zatem kwadratowo względem długości sekwencji.

Okazuje się, że wprowadzony podział eksperymentu na fazy w ogóle nie jest potrzebny, tj. kolejność usuwania sąsiadujących ze sobą cząstek i antycząstek nie ma znaczenia. Spostrzeżenie tego faktu nie jest trudne, ale dowód okazuje się niespodziewanie skomplikowany. Poniżej dowiedziemy prawdziwości tego twierdzenia poprzez tzw. dowód *nie wprost*, czyli założymy, że jest inaczej, i wykażemy, że prowadzi to do sprzeczności.

Załóżmy, że istnieje stan początkowy o skończonej długości, z którego poprzez dobór kolejności usuwania liter możemy otrzymać *różne* stabilne stany końcowe. Jeśli takich stanów początkowych jest więcej, oznaczmy przez S najkrótszy z nich.

Zauważmy, że po wykonaniu pierwszego kroku w S, na pewno otrzymamy stan krótszy, a zatem stan, który prowadzi tylko do jednego możliwego stanu końcowego (bo S jest najkrótszym stanem o wielu możliwych stanach końcowych). W takim razie już pierwszy krok musi dać się wykonać na przynajmniej dwa sposoby prowadzące do różnych stanów końcowych. Wybierzmy dowolne dwa z takich sposobów oznaczając je w następujący sposób: jeżeli usuniemy parę liter (t_1, t_2), to otrzymamy stan S_t, zaś jeśli usuniemy parę liter (u_1, u_2), to otrzymamy stan S_u. Stany S_t oraz S_u prowadzą zatem do różnych stanów końcowych – każdy z nich dokładnie do jednego.

Są tu dwa przypadki do rozważenia. Jest możliwe, że pary (t_1, t_2) i (u_1, u_2) mają element wspólny, tj. częściowo nachodzą na siebie. Wówczas mamy do czynienia z sytuacją omówioną w treści zadania, np. z łańcuchem znaków „AaA”. Wiemy, że wówczas nie ma znaczenia, którą parę wybierzemy – pozostanie nam „A”. To by znaczyło, że $S_t = S_u$, co jest niemożliwe, więc ten przypadek możemy wykluczyć.

W przeciwnym razie, pary nie mają żadnej części wspólnej i są od siebie niezależne. To znaczy, że po usunięciu jednej z nich, druga pozostaje nietknięta i *vice versa*, czyli w S_t wciąż istnieje para (u_1, u_2), zaś w S_u istnieje para (t_1, t_2). Wiemy, że z osobna S_t i S_u mają tylko jeden możliwy stan końcowy, a w takim razie możemy w nich wykonywać dalsze kroki w dowolny sposób, by go osiągnąć. To znaczy, że w następnej kolejności z S_t możemy bez wahania usunąć parę (u_1, u_2) otrzymując S'_t, a z S_u możemy usunąć (t_1, t_2) otrzymując S'_u. Zarówno S'_t jak i S'_u są jednak niczym innym jak stanem S pozbawionym liter $\{t_1, t_2, u_1, u_2\}$, a więc są sobie równe, czyli prowadzą do jednakowego stanu końcowego. To znaczy, że S_t oraz S_u także prowadzą do tego

samego stanu końcowego, co łamie nasze założenia i prowadzi do sprzeczności.

To dowodzi, że każdy stan początkowy posiada tylko jeden odpowiadający mu stan końcowy, więc litery możemy usuwać w dowolnej kolejności.

Prowadzi to do bardzo prostego w implementacji rozwiązania z wykorzystaniem struktury stosu. Zamiast operować bezpośrednio na łańcuchu liter (co mogłoby być kosztowne – usuwanie ze środka łańcucha wymaga przesunięcia wszystkich dalszych elementów) będziemy tworzyć stan końcowy na bieżąco, przechodząc po łańcuchu od lewej do prawej.

Stos będzie reprezentował nasz konstruowany stan końcowy. Dla każdej napotkanej w łańcuchu litery sprawdzamy, czy zajdzie jej zderzenie z literą na szczycie stosu, tj. czy są to litery tego samego rodzaju i różnej wielkości. Jeśli tak, to nie dodajemy napotkanej litery i usuwamy literę ze szczytu stosu (to odsłania poprzedni element stosu, dając dostęp do wcześniejszych liter w celu umożliwienia dalszych zderzeń w przyszłości). W przeciwnym razie, umieszczamy tę nową literę na szczycie stosu. W rezultacie na stosie nigdy nie pojawią się obok siebie dwie litery, między którymi powinno dojść do zderzenia, więc po jednokrotnym przejściu całego łańcucha, przechowuje on od razu stabilny stan końcowy eksperymentu.

Ponieważ wszystkie operacje wykonywane w pętli mają czas stały, taki algorytm ma liniowy czas działania.

Jak w wielu przypadkach, w praktyce zamiast wykorzystania udostępnianej przez język struktury stosu, łatwiej użyć bardziej uniwersalnych struktur, np. listy w języku Python albo klasy `vector` lub `string` w C++. Wówczas łatwiej jest choćby odczytać pod koniec wynik, który algorytm umieszcza wszak na stosie w kolejności od spodu do szczytu, a więc odwrotnej do tej, w której stos pozwala pobierać elementy.

W języku Python, nasze rozwiązanie może wyglądać tak:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Antimatter_Collider_Gnomy/solution.py.

```
1  s = input()
2  stos = []
3  for c in s:
4      if stos and c.swapcase() == stos[-1]:
5          stos.pop()
6      else:
7          stos.append(c)
8  print(''.join(stos))
```

Używamy tu zwykłej listy, która – co warto zauważyć – udostępnia programiście funkcje pozwalające na naturalne i wygodne traktowanie jej jako stos (`append` dodaje element na końcu listy, a `pop` usuwa element z końca). W wierszu 4 używamy nazwy `stos` w wyrażeniu warunkowym, aby sprawdzić, czy stos jest niepusty oraz funkcji `swapcase` do zamiany wielkości litery wczytanej z wejścia i porównania jej z ostatnim elementem stosu (`stos[-1]`). Wiersz 5, to oczywiście „anihilacja” cząstek (litera wczytana z wejścia „anihiluje” literę ze szczytu stosu), a w ostatnim wierszu „sklejamy” wszystkie litery ze stosu do postaci jednego `stringa` i wypisujemy na wyjście.

Na następnej stronie przedstawiamy wersję w C++.

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Antimatter_Collider_Gnomy/solution.cpp.

```cpp
#include <cctype>
#include <iostream>
#include <string>

bool sa_przeciwne(char a, char b) {
  if (std::isupper(a))
    return std::tolower(a) == b;
  else
    return std::toupper(a) == b;
}

int main() {
  std::string s;
  std::cin >> s;
  std::string stos;
  for (const auto& c : s) {
    if (!stos.empty() && sa_przeciwne(c, stos.back()))
      stos.pop_back();
    else
      stos.push_back(c);
  }
  std::cout << stos << '\n';
  return 0;
}
```

Algorytm jest oczywiście ten sam, ale zamiana wielkości i porównywanie liter jest tu nieco bardziej kłopotliwe, więc zostało zaimplementowane w postaci osobnej funkcji `sa_przeciwne`. Warto również zwrócić uwagę na wykorzystanie zmiennej typu `string` do reprezentacji naszego stosu. Jest to w gruncie rzeczy dość naturalny wybór, zważywszy na to, że mamy tu faktycznie do czynienia z *ciągiem liter*, a ponadto typ `string` udostępnia nam specjalne funkcje do dodawania i usuwania litery na ostatniej pozycji (`push_back` i `pop_back`).

23 (III) – Antimatter Collider – Skrzaty *(zad. w jęz. angielskim)*

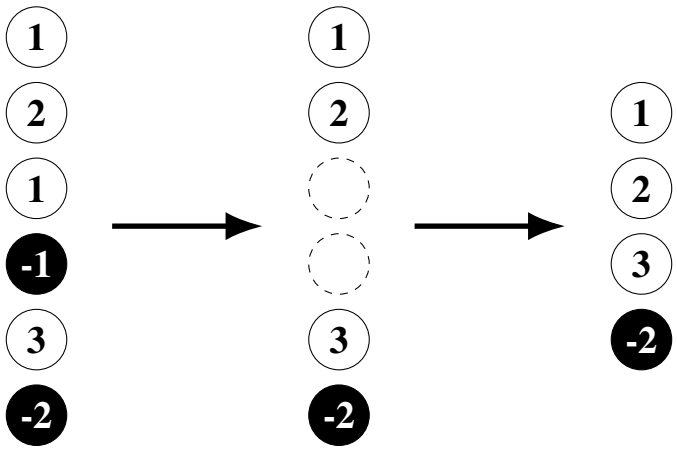
Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap finałowy)
Język programowania: Scratch

23.1 Task description

Scientists at the Bytesian Antimatter Collider are studying behavior of subatomic antimatter particles ("antiparticles"). Every time a new experiment is performed, scientists first note down the initial arrangement of particles and antiparticles in a sequence, then start the experiment, and finally compare the practical and theoretical results.

There are 10 types of particles under investigation, along with their corresponding antiparticles – for convenience, the particles are labeled with natural numbers (1) to (10) and their corresponding antiparticles are labeled with the negative numbers (-1) to (-10), so that a particle and its antiparticle are represented by opposite numbers. Whenever a particle is directly next to its corresponding antiparticle with nothing else in between, both the particle and the antiparticle annihilate each other, i.e. they suddenly disappear. Other particles move closer together as a result to fill out the empty space.

In the following figure an example is presented, in which the initial sequence of particles (on the left) contains adjacent particle-antiparticle pair: (1) and (-1). This pair is therefore annihilated as shown on the right.



Task

Write a program that will find the theoretical final result for a given sequence of particles and antiparticles. Start from a new, empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `DANE` should be manually filled with example input data, while the list `WYNIKI` is a place where your program – executed by clicking the green flag – should output the obtained result.

Note, that the experiment might take several phases – first some annihilations take place, then the particles move closer together, which causes more annihilations, etc. Note also, that sometimes a particle may be adjacent to two of its corresponding antiparticles (or vice versa) – one on each side. In those cases, the pair that disappears is the one that allows for more annihilations to take place during that phase. For example, the sequence: (-8) (8) (-8) (8) will always disappear instantly (instead of the middle pair (8) (-8) disappearing during the first phase and the remaining (-8) (8) during the second one). If there are multiple ways to achieve the maximum number of annihilations, one of them is chosen at random, e.g. in (3) (-3) (3) either the first pair (3) (-3) or the second pair (-3) (3) is chosen – in either case leaving behind a single particle (3). Because particles of the same type are identical to each other, this choice never affects the outcome.

Input description

The list `DANE` contains up to 100 000 elements, representing the consecutive particles in the input sequence. Each element is a whole number from the set:

$\{-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

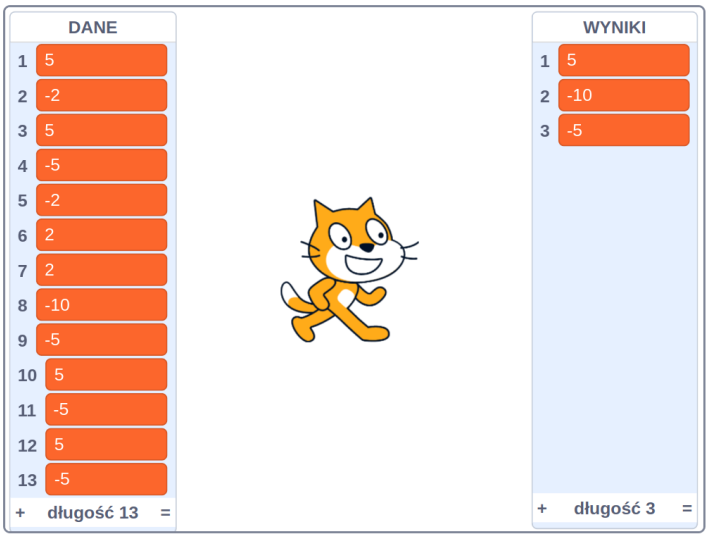
The positive numbers represent particles and the negative numbers represent antiparticles, as described above.

Output description

The output list `WYNIKI` should contain a sequence (possibly empty) of numbers in the same format as the input, denoting the final result of the experiment – a stable sequence of particles in which no more annihilations may occur.

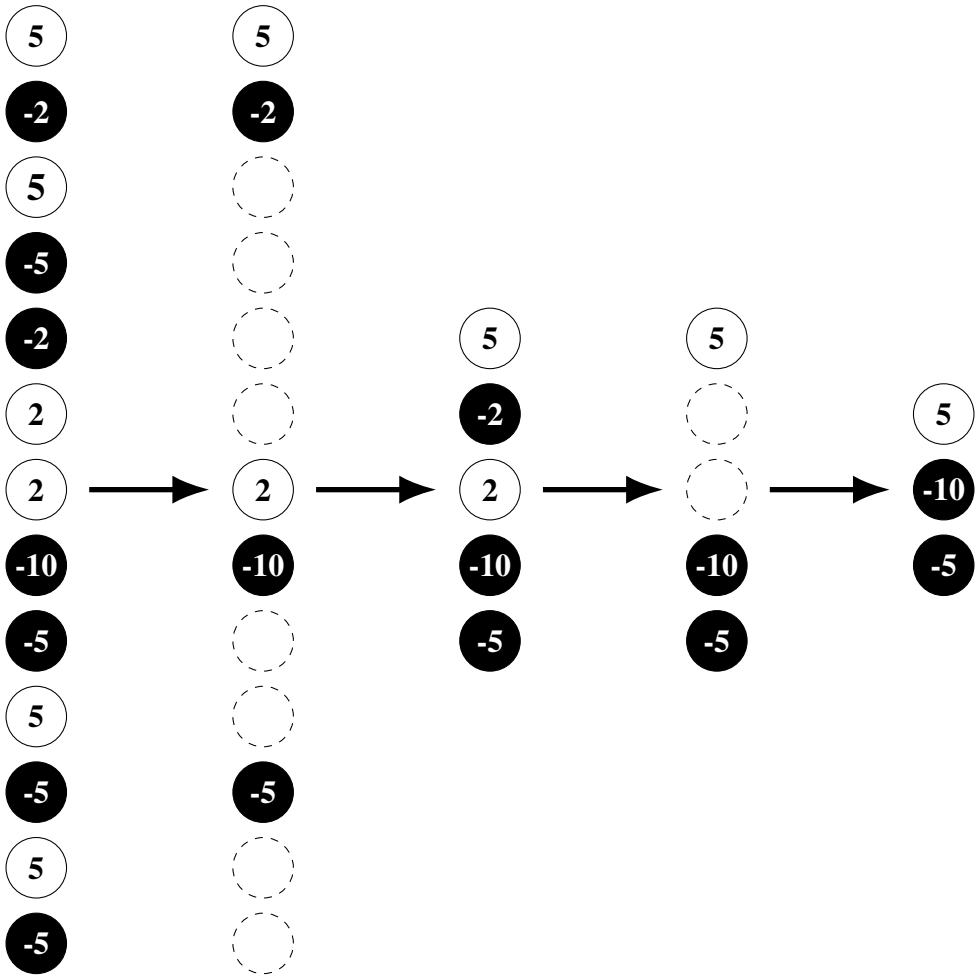
Example

In the following figure an example of input data and the correct result are presented:



Explanation

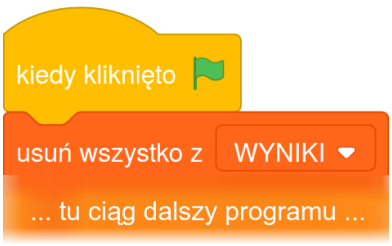
The experiment has been illustrated below. Note, that any of the three (-5) antiparticles at the bottom could have remained in the sequence, with identical results.



Remarks

Please remember, that the list `DANE` should be filled-in manually, by typing the input data with the keyboard. It is a good idea to test the program also with input values other than these in the examples.

The content of the list `WYNIKI` should always be initially removed. Your program should therefore start as demonstrated below:

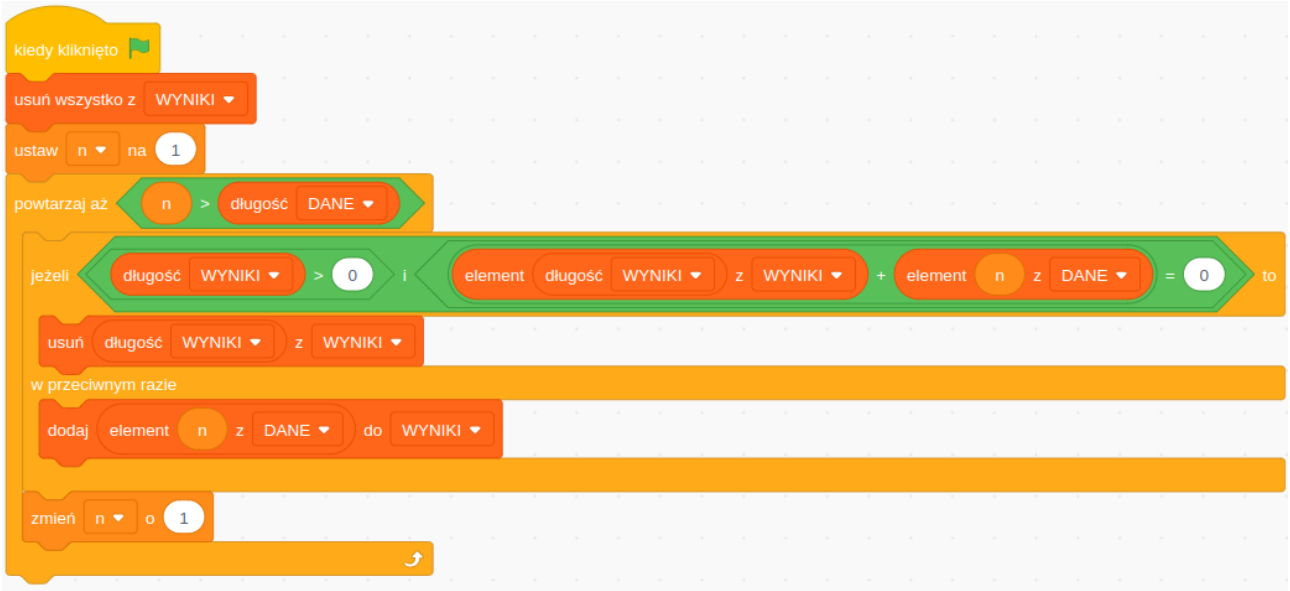


23.2 Rozwiązanie

Treść zadania w wersji dla Skrzatów musiała zostać sformułowana nieco inaczej, z uwagi na problem z rozróżnianiem wielkości znaków w Scratchu. Przyjęto więc proste założenie, że cząstki reprezentowane są przez liczby naturalne, przy czym liczby przeciwne względem siebie oznaczają parę cząstka-antycząstka. Wykrycie takiej pary jest dzięki temu wyjątkowo proste – wystarczy dodać obie liczby i sprawdzić, czy wynik jest zerowy.

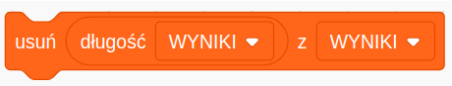
Poza tą różnicą, problem jest ten sam i rozwiązujemy go analogicznie – za pomocą stosu.

Oczywiście nie mamy w Scratchu osobnej struktury danych reprezentującej stos, więc używamy tu zwykłej listy. Nic nie stoi na przeszkodzie, żeby wykorzystać w tym celu bezpośrednio listę `WYNIKI` (dzięki czemu nie musimy dodatkowo przepisywać zawartości stosu „na wyjście”, a kod robi się tym samym bardzo zwięzły):



Przyjmujemy, że ostatni element listy `WYNIKI` stanowi wierzchołek stosu, a zatem położenie na nim nowego elementu wymaga tylko bloku `dodaj`. Do odczytania elementu z wierzchołka stosu musimy już podać jego numer, ale ponieważ jest to zawsze ostatni element listy `WYNIKI`, więc jego numer równy jest po prostu długości tej listy i odczytujemy go tak:

Podobnie realizujemy funkcjonalność operacji pop zdejmującej element ze stosu:



Przedstawione tu rozwiązanie znajdziesz w pliku:

./zadania/03_Wczytywanie_wypisywanie/Antimatter_Collider_Skrzaty/solution.sb3.

24 (III) – Fotka – Gnomy

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

24.1 Treść zadania

Dwie szkoły podstawowe – nr 128 i nr 256 – rywalizowały niedawno w lokalnych zawodach programistycznych. Każda ze szkół wystawiła taką samą liczbę zawodników, a po zawodach wszyscy razem ustawili się w jeden szereg i unieśli otrzymane dyplomy, by fotograf zrobił im wspólne zdjęcie. Ów fotograf miał widać własny zamysł na zdjęcie, bowiem ustawił zawodników na zmianę z jednej i drugiej szkoły – najpierw ze szkoły nr 128, potem nr 256, potem znów nr 128, itd.

Naturalnie, każda ze szkół stara się przedstawić swoją reprezentację w jak najlepszym świetle, a na dyplomach wyraźnie widać wyniki poszczególnych uczniów, dlatego w gazetce szkolnej zostanie zamieszczony jedynie odpowiednio dobrany spójny wycinek zdjęcia – taki, na którym różnica między sumą punktów osiągniętą przez własnych uczniów a sumą punktów przeciwników jest jak największa (oczywiście na korzyść danej szkoły). Dla każdej z dwóch szkół, znajdź najkorzystniejszą wartość tej różnicy.

Opis wejścia

Pierwszy wiersz standardowego wejścia zawiera pojedynczą liczbę parzystą n ($2 \leq n \leq 1\,000\,000$), oznaczającą łączną liczbę uczestników zawodów. W drugim wierszu wejścia znajduje się ciąg n liczb całkowitych $w_1, w_2, ..., w_n$ ($0 \leq w_i \leq 100\,000$) oddzielonych pojedynczymi spacjami. Są to wyniki kolejnych uczestników licząc od lewej strony szeregu. Pierwszy z wyników należy do zawodnika szkoły nr 128.

Opis wyjścia

Na standardowe wyjście należy wypisać dwa wiersze. W pierwszym wierszu powinna znaleźć się różnica punktów na fragmencie zdjęcia zamieszczonym w gazetce szkoły nr 128. W drugim, analogicznie, różnica punktów, ale dla szkoły nr 256.

Przykład

Dla danych wejściowych:

```
6
60 10 35 100 80 65
```

poprawnym wynikiem jest:

```
85
100
```

Wyjaśnienie przykładu

Gazetka szkoły podstawowej nr 128 wybierze fragment zawierający zawodników z wynikami $60, 10, 35$. Różnica punktów na tym zakresie wynosi $(60 + 35) - 10 = 85$. Na pewno chętnie zamieściłaby także swojego ucznia z wynikiem 80, ale ten od pozostałych oddzielony jest przeciwnikiem o wyniku 100, co zbytnio działałoby na jej niekorzyść. Gazetka szkoły nr 256 zamieści po prostu fragment zawierający wyłącznie swojego reprezentanta, który zdobył 100 punktów.

24.2 Rozwiązanie

W tym zadaniu łatwo jest skonstruować algorytm naiwny, który będzie liczył różnicę punktów w *każdym możliwym* fragmencie ciągu wejściowego. Wymaga to użycia trzech (zagnieżdżonych) pętli, ponieważ:

- każdy element może być początkiem naszego fragmentu (pierwsza pętla);
- dla każdego początku, każdy z następujących po nim elementów może być końcem (druga pętla);
- dla każdego początku i każdego końca trzeba przetworzyć wszystkie elementy zawarte między nimi (trzecia pętla).

Jak łatwo zauważyć otrzymamy w ten sposób algorytm o złożoności $O(n^3)$.

Przykładową implementację rozwiązania naiwnego w Pythonie przedstawiamy poniżej:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution-naive.py.

```
1  n = int(input())
2  w = [int(x) for x in input().split()]
3  low = 0
4  high = 0
5  for i in range(n):
6      for j in range(i, n):
7          sum = 0
8          for k in range(i, j + 1):
9              if k % 2 == 0:
10                 sum += w[k]
11             else:
12                 sum -= w[k]
13         low = min(low, sum)
14         high = max(high, sum)
15 print(high)
16 print(-low)
```

Wspomniane trzy pętle znajdują się w wierszach 5, 6 i 8, a używane w nich zmienne reprezentują odpowiednio indeks początku fragmentu (`i`), indeks końca fragmentu (`j`) i indeks bieżącego elementu (`k`).

Dla każdego fragmentu obliczamy różnicę punktów między szkołami za pomocą zmiennej `sum`. Zmienna ta jest zerowana dla każdego fragmentu (wiersz 7), a następnie dodajemy do niej punkty szkoły 128 (czyli elementy o indeksach 0, 2, 4, ... – wiersz 10) i odejmujemy punkty szkoły 256 (elementy o indeksach 1, 3, 5, ... – wiersz 12). Po przetworzeniu fragmentu sprawdzamy, czy uzyskana w nim suma jest większa od największej, jaką dotąd uzyskaliśmy (we wcześniejszych fragmentach). Tę najlepszą dotychczasową wartość przechowujemy w zmiennej `high`, którą – w przypadku uzyskania wyniku jeszcze lepszego – aktualizujemy w wierszu 14.

Wynik dla szkoły 256 moglibyśmy otrzymać analogicznie, dodając punkty szkoły 256 i odejmując punkty szkoły 128, ale ponieważ sprowadza się to do zmiany znaku każdego z sumowanych elementów, wystarczy więc zamiast tego po prostu zmienić znak wyniku. A zatem korzystamy z tej samej zmiennej `sum`, z tym że interesuje nas tu nie maksymalna, a minimalna jej wartość spośród wszystkich fragmentów (wiersz 13). Będzie to oczywiście wartość niedodatnia, którą negujemy przy wypisywaniu (wiersz 16).

Na następnej stronie przedstawiono implementację rozwiązania naiwnego w C++.

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution-naive.cpp.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  std::vector<int> w(n);
  for (int i = 0; i < n; i++) {
    std::cin >> w[i];
  }
  long long min = 0;
  long long max = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      long long sum = 0;
      for (int k = i; k <= j; k++) {
        if (k % 2 == 0)
          sum += w[k];
        else
          sum -= w[k];
      }
      if (sum < min)
        min = sum;
      if (sum > max)
        max = sum;
    }
  }
  std::cout << max << '\n' << -min << '\n';
  return 0;
}
```

Odpowiednikami zmiennych `low` i `high` są tu zmienne `min` i `max`; oprócz tego fragmenty kodu odpowiedzialne za wczytywanie danych (wiersze 5-12) oraz za szukanie maksimum i minimum (wiersze 24-27) są tu nieco bardziej rozbudowane, ale sam algorytm jest oczywiście taki sam jak w wersji dla Pythona.

Rozwiązanie naiwne jest dobrym punktem wyjścia do optymalizacji naszego algorytmu. Rozważmy, co dzieje się we wnętrzu środkowej pętli – iterującej „po `j`", czyli po końcu fragmentu. Obliczamy w niej sumę elementów rozpoczynając od `i`-tego, ale zauważmy, że dla każdego `j` robimy to od nowa, a przecież `i` się nie zmienia (zmieni się dopiero po zakończeniu wszystkich powtórzeń środkowej pętli). A zatem wystarczy inicjalizować naszą sumę tylko raz dla danego początku fragmentu `i`, a przy każdym powtórzeniu środkowej pętli, czyli przy przesunięciu końca fragmentu o jeden element, wystarczy tylko ten element do tej sumy dodać (lub odjąć). W ten sposób otrzymujemy następujące rozwiązanie (p. kod na następnej stronie):

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution-intermediate.py.

```python
n = int(input())
w = [int(x) for x in input().split()]
for i in range(len(w)):
    if i % 2 != 0:
        w[i] = -w[i]
low = 0
high = 0
for i in range(n):
    sum = 0
    for v in w[i:]:
        sum += v
        low = min(low, sum)
        high = max(high, sum)
print(high)
print(-low)
```

Mamy tu już tylko dwie zagnieżdżone pętle (wiersz 8 i 10), więc możemy szacować złożoność obliczeniową na $O(n^2)$. Zauważmy tu jeszcze jedną optymalizację – kwestię zmiany znaku co drugiego elementu uwzględniamy tym razem zaraz po wczytaniu danych (wiersze 3-5), jeszcze przed główną częścią algorytmu w wierszach 8-13. Dzięki temu możemy to zrobić jednorazowo, podczas gdy w rozwiązaniu naiwnym dla każdego k-tego elementu niepotrzebnie sprawdzaliśmy wielokrotnie czy trzeba go dodać, czy odjąć.

Analogiczny kod w wersji C++:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution-intermediate.cpp.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  std::vector<int> w(n);
  for (int i = 0; i < n; i++) {
    std::cin >> w[i];
    if (i % 2)
      w[i] = -w[i];
  }
  long long min = 0;
  long long max = 0;
  for (int i = 0; i < n; i++) {
    long long sum = 0;
    for (int j = i; j < n; j++) {
      sum += w[j];
      if (sum < min)
        min = sum;
      if (sum > max)
        max = sum;
```

```
      }
    }
    std::cout << max << '\n' << -min << '\n';
    return 0;
}
```

Warto zadać sobie pytanie, czy możliwa jest dalsza optymalizacja naszego rozwiązania. Obecnie rozpoczynamy od każdego możliwego (`i`-tego) elementu i sumujemy elementy następujące po nim, szukając największej możliwej wartości tej sumy (dla szkoły 256 – najmniejszej). Czy musimy jednak faktycznie za każdym razem rozpoczynać sumowanie na nowo? Przypuśćmy, że rozpoczęliśmy od pierwszego elementu i obliczyliśmy maksymalną sumę. Teraz chcemy rozpocząć na nowo od drugiego elementu i sprawdzić, czy uzyskamy wyższą sumę. Ale to byłoby możliwe, tylko gdyby pierwszy element był ujemny (gdyby jego obecność „psuła" nam wynik, patrząc z punktu widzenia szkoły 128). Ponieważ jednak jest nieujemny, to nie ma potrzeby tego sprawdzać.

Idąc dalej, zastanówmy się, czy jest sens rozpoczynać od trzeciego elementu, tzn. czy możemy uzyskać większą maksymalną sumę pomijając dwa pierwsze elementy. Oczywiście tak, ale tylko wtedy, gdy te dwa pierwsze elementy „psują" nam wynik, czyli gdy ich suma jest mniejsza od zera. Analogicznie rozumujemy dla trzech, czterech, ... początkowych elementów.

Kluczowe jest zauważenie następujących faktów:

Stwierdzenie, czy powinniśmy rozpocząć na nowo od danego elementu wymaga zsumowania wszystkich wcześniejszych i sprawdzenia, czy ich suma jest większa od zera;

- jeśli jest – sumujemy spokojnie dalej (być może dochodząc do końca ciągu i wypisując najwyższą zaobserwowaną sumę jako wynik);
- jeśli nie jest (początkowe elementy „psują" wynik) – powinniśmy rozpocząć na nowo od bieżącego elementu, jednakże zauważamy tu (patrząc na kod obu wcześniejszych rozwiązań), że tak naprawdę sprowadza się to tylko do wyzerowania („zresetowania") zmiennej `sum` i *nie ma potrzeby rozpoczynania na nowo* (wystarczy wyzerować `sum` i pętla „idzie dalej").

Łącząc powyższe obserwacje, stwierdzamy, że do rozwiązania wystarczy nam pojedyncze przejście przez listę wejściową, a zatem otrzymujemy kod o złożoności $O(n)$ (liniowej), pokazany poniżej:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution.py.

```
n = int(input())
w = [int(x) for x in input().split()]
for i in range(len(w)):
    if i % 2 != 0:
        w[i] = -w[i]
low = 0
high = 0
low_sum = 0
high_sum = 0
for score in w:
    low_sum += score
    low = min(low, low_sum)
```

```
        low_sum = min(low_sum, 0)
        high_sum += score
        high = max(high, high_sum)
        high_sum = max(high_sum, 0)
    print(high)
    print(-low)
```

Jak widać, zamiast pojedynczej sumy bieżącej `sum` mamy tu osobno sumę dla szkoły 128 (`high_sum`) i dla szkoły 256 (`low_sum`). Sprawdzanie, czy `sum_high` nie jest mniejsza od zera realizujemy tu w zwięzły sposób za pomocą instrukcji `max` (jeśli `sum_high` byłaby ujemna, to zostanie ustawiona na zero). Analogicznie upewniamy się, że `sum_low` jest ujemna – jeśli nie, zerujemy ją za pomocą funkcji `min`. Wykorzystanie zmiennych `high` i `low` jest takie samo jak w poprzednich wersjach algorytmu.

Wersja rozwiązania liniowego w C++ zaprezentowana jest poniżej – poza innymi nazwami zmiennych, mamy tu oczywiście zaimplementowany dokładnie ten sam algorytm:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Fotka_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  std::vector<int> w(n);
  for (int i = 0; i < n; i++) {
    std::cin >> w[i];
    if (i % 2)
      w[i] = -w[i];
  }
  long long min = 0, min_sum = 0;
  long long max = 0, max_sum = 0;
  for (int i = 0; i < n; i++) {
    min_sum += w[i];
    if (min_sum < min)
      min = min_sum;
    if (min_sum > 0)
      min_sum = 0;
    max_sum += w[i];
    if (max_sum > max)
      max = max_sum;
    if (max_sum < 0)
      max_sum = 0;
  }
  std::cout << max << '\n' << -min << '\n';
  return 0;
}
```

25 (III) – Fotka – Skrzaty

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

25.1 Treść zadania

Dwie szkoły podstawowe – nr 128 i nr 256 – rywalizowały niedawno w lokalnych zawodach programistycznych. Każda ze szkół wystawiła taką samą liczbę zawodników, a po zawodach wszyscy razem ustawili się w jeden szereg i unieśli otrzymane dyplomy, by fotograf zrobił im wspólne zdjęcie. Ów fotograf miał widać własny zamysł na zdjęcie, bowiem ustawił zawodników na zmianę z jednej i drugiej szkoły – najpierw ze szkoły nr 128, potem nr 256, potem znów nr 128, itd.

Naturalnie, każda ze szkół stara się przedstawić swoją reprezentację w jak najlepszym świetle, a na dyplomach wyraźnie widać wyniki poszczególnych uczniów, dlatego w gazetce szkolnej zostanie zamieszczony jedynie odpowiednio dobrany spójny wycinek zdjęcia - taki, na którym różnica między sumą punktów osiągniętą przez własnych uczniów a sumą punktów przeciwników jest jak największa (oczywiście na korzyść danej szkoły). Dla każdej z dwóch szkół, znajdź najkorzystniejszą wartość tej różnicy.

Zadanie

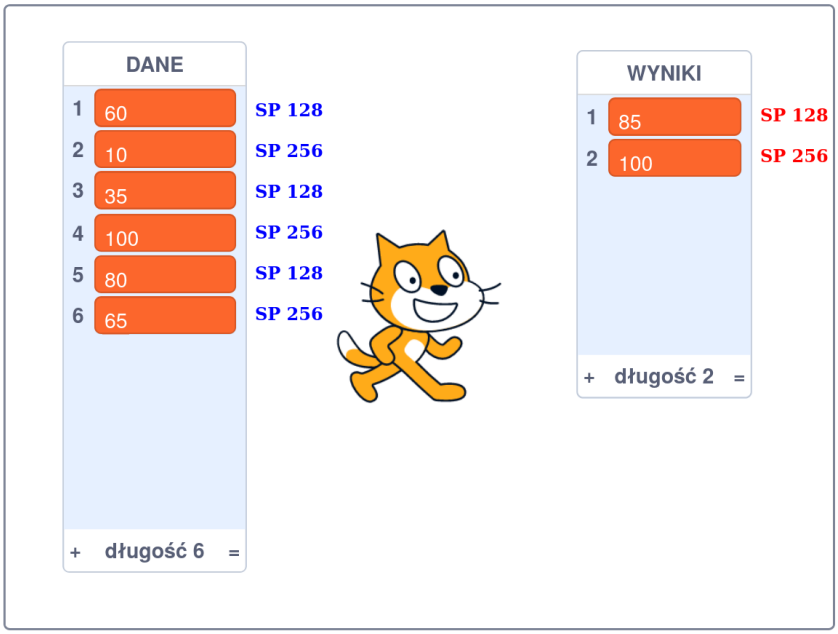
W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `WYNIKI` należy na razie pozostawić pustą. Do listy `DANE` w kolejnych pozycjach należy wpisać (ręcznie) n liczb całkowitych $w_1, w_2, ..., w_n$ ($0 \leq w_i \leq 100\,000$), określających wyniki kolejnych uczestników licząc od lewej strony szeregu. Liczba uczestników n jest parzysta. Pierwszy z wyników należy do zawodnika szkoły nr 128.

Celem zadania jest napisanie programu (skryptu), który wpisze do listy `WYNIKI` dwie liczby. Na pierwszej pozycji listy `WYNIKI` powinna znaleźć się różnica punktów na fragmencie zdjęcia zamieszczonym w gazetce szkoły nr 128. Na drugiej pozycji, analogicznie, różnica punktów, ale dla szkoły nr 256. Na następnej stronie pokazano przykład działania programu (niebieskie i czerwone numery szkół podano tylko dla ilustracji).

Przykład



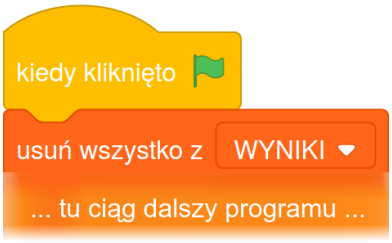
Wyjaśnienie

Gazetka szkoły podstawowej nr 128 wybierze fragment zawierający zawodników z wynikami $60, 10, 35$. Różnica punktów na tym zakresie wynosi $(60 + 35) - 10 = 85$. Na pewno chętnie zamieściłaby także swojego ucznia z wynikiem 80, ale ten od pozostałych oddzielony jest przeciwnikiem o wyniku 100, co zbytnio działałoby na jej niekorzyść. Gazetka szkoły nr 256 zamieści po prostu fragment zawierający wyłącznie swojego reprezentanta, który zdobył 100 punktów.

Uwagi

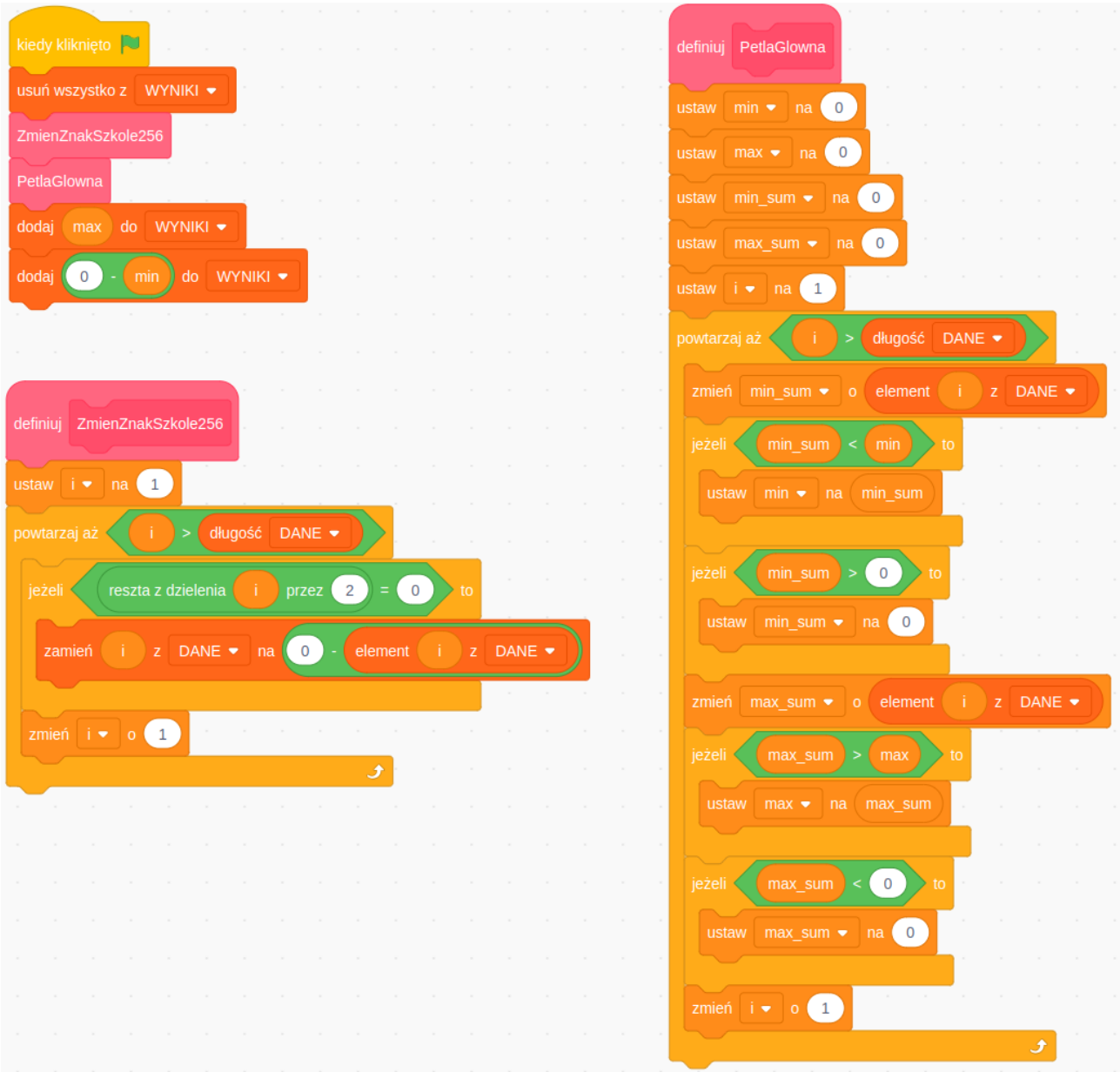
Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



25.2 Rozwiązanie

Przedstawione poniżej rozwiązanie zadania „Fotka” zaimplementowane w Scratchu odpowiada dokładnie ostatniej wersji (zoptymalizowanej) w C++/Pythonie, którą przedstawiliśmy wyżej.



Rysunek 25.1: Zadanie „Fotka” – rozwiązanie

Rozwiązanie to znajdziesz w pliku:

./zadania/03_Wczytywanie_wypisywanie/Fotka_Skrzaty/solution.sb3.

26 (III) – Kto stoi przede mną – Gnomy

*Autor: **Andrzej Dyrek**, **Akademia Górniczo-Hutnicza w Krakowie***
*Kategoria: **Gnomy (IV edycja – zawody algorytmiczne – etap regionalny)***
*Język programowania: **C++/Python***

26.1 Treść zadania

Brajanek wrócił ze szkoły bardzo podekscytowany (chłopiec uczęszcza do Królewskiej Szkoły Konkwistadorów i Matadorów w Bitawie – elitarnej placówki dla szlachetnie urodzonych młodzieńców). Halinka, mama Brajanka, zaczęła go wypytywać o szkolne wrażenia.

– *Co robiliście na informatyce?* – zagadnęła syna.

– *Ustawialiśmy się jeden za drugim i wyciągaliśmy włoski!* – pochwalił się Brajanek – *to jest: wnioski!*

– *Jakie wnioski?* – zdziwiła się Halinka.

– *Patrzyliśmy, kto stoi przed nami i Kewinek musiał powiedzieć, w jakiej kolejności stoimy!* – wyjaśniał Brajanek.

– *Ale jak to, czy Kewinek nie mógł po prostu wymienić kolejnych uczniów stojących w rzędzie?* – dopytywała się mama.

– *No nie mógł, bo go w szafie zamknęliśmy!* – roześmiał się Brajanek.

Sprawa miała się tak: uczniowie (ponumerowani od 1 do n) rzeczywiście ustawili się losowo w kolejce tak, że każda osoba widziała tylko kolegów stojących przed nią. Na przykład dla $n = 5$ mogło to wyglądać następująco:

$$\leftarrow \mathbf{2} \quad \leftarrow \mathbf{3} \quad \leftarrow \mathbf{5} \quad \leftarrow \mathbf{1} \quad \leftarrow \mathbf{4}$$

Wszyscy uczniowie zwróceni są w lewą stronę, zatem uczeń o numerze $\mathbf{2}$ nie widzi nikogo, uczeń $\mathbf{3}$ widzi tylko ucznia $\mathbf{2}$, uczeń $\mathbf{5}$ widzi uczniów $\mathbf{2}$ i $\mathbf{3}$, i tak dalej (Kewinek dalej siedzi w szafie).

Nauczyciel zbiera od każdego ucznia informację, ilu widzi uczniów o numerach większych od jego własnego. Na przykład uczeń o numerze $\mathbf{2}$ podaje wartość 0, podobnie jak uczniowie o numerach $\mathbf{3}$ i $\mathbf{5}$.

Uczeń $\mathbf{1}$ podaje liczbę 3, bo widzi przed sobą uczniów $\mathbf{2}$, $\mathbf{3}$ oraz $\mathbf{5}$ (wszyscy mają numery większe niż $\mathbf{1}$).

Z kolei uczeń $\mathbf{4}$ podaje wartość 1, bo wśród widzianych przez niego czterech uczniów tylko uczeń $\mathbf{5}$ ma numer większy od $\mathbf{4}$.

Całość informacji wygląda zatem tak:

$$\mathbf{0} \quad \mathbf{0} \quad \mathbf{0} \quad \mathbf{3} \quad \mathbf{1}$$

Pytanie brzmi: czy Kewinek jest w stanie odtworzyć kolejność uczniów w kolejce – tylko na podstawie powyższych informacji?

– *Powiedz mi jeszcze, synku* – zapytała na koniec mama – *czy zawsze podajecie Kewinkowi poprawne i rzetelne informacje?*

– *Nie, mamo, czasem robimy go w bolka!* – uśmiechnął się chłopiec.

Zadanie

Napisz program, który wczyta informacje, jakie otrzymuje Kewinek i wyznaczy kolejność uczniów w kolejce – lub zawyrokuje, że otrzymane dane są nieprawidłowe (czyli nie opisują rzeczywistej kolejności uczniów).

Opis wejścia

Dane wejściowe składają się z dwóch wierszy.

Pierwszy wiersz zawiera liczbę naturalną N oznaczającą ilość uczniów stojących w kolejce ($2 \leq N \leq 12\,000$ – to duża szkoła!).

W drugim wierszu znajduje się N liczb naturalnych mniejszych od N, oddzielonych pojedynczymi odstępami.

Opis wyjścia

Program powinien wypisać wiersz tekstu zawierający listę N numerów uczniów stojących w kolejce. Pomiędzy numerami należy umieścić pojedyncze spacje.

W przypadku, gdy dane wejściowe są nieprawidłowe, należy wypisać słowo **NIE** (wielkimi literami).

Przykład

Dla danych wejściowych opisanych w treści zadania:

```
5
0 0 0 3 1
```

program powinien wypisać:

```
2 3 5 1 4
```

Dla następujących danych wejściowych:

```
3
0 1 2
```

program powinien wypisać:

```
3 2 1
```

Uczniowie stoją tu w kolejności malejących numerów.

Z kolei dla danych wejściowych:

```
3
0 2 2
```

program powinien wypisać:

```
NIE
```

Dane wejściowe nie odpowiadają tu żadnej kolejności numerów trzech uczniów.

26.2 Rozwiązanie

Zgodnie z treścią zadania, na wejściu mamy listę wartości liczbowych, przy czym i-ty element tej listy informuje nas o tym, ilu uczniów o numerach większych od własnego widzi i-ty uczeń w kolejce.

Będziemy przetwarzać tę listę od lewej do prawej, przy czym naszym celem będzie, aby po przetworzeniu każdej i-tej wartości mieć i dzieci ustawionych już we właściwej kolejności. Ich faktyczne numery będą jednak początkowo nieistotne dla nas. Dla każdego i możemy założyć, że mamy i dzieci o (tymczasowych) numerach $1, 2, ..., i$, a interesować nas będzie LISTA POZYCJI, na których stoją.

Rozważmy sytuację po wczytaniu i+pierwszego dziecka. Wiemy na której pozycji stoi (oczywiście na pozycji $i+1$), ale pytanie – jaki ma numer? To łatwo stwierdzić, analizując wczytaną wartość wejściową k, oznaczającą – zgodnie z treścią zadania – liczbę dzieci o numerach większych, stojących na wcześniejszych pozycjach. Gdyby wartość k wynosiła np. 0, to oznaczałoby, że nasze i+pierwsze dziecko ma największy numer spośród wszystkich $i+1$ dzieci (czyli musi mieć numer $i+1$), bo wszystkie, które widzi mają numery mniejsze. Dodamy więc jego pozycję $(i+1)$ na i+pierwszym miejscu (czyli na końcu) LISTY POZYCJI. Gdyby natomiast k było równe np. 1, oznaczałoby to, że nasze i+pierwsze dziecko widzi dokładnie jedno dziecko o numerze większym od siebie, czyli o numerze $i+1$ (bo tyle jest dzieci), więc samo musi mieć numer o jeden mniejszy niż liczba dzieci (a więc numer i). Wstawilibyśmy więc jego pozycję (czyli $i+1$) na i-tym miejscu LISTY POZYCJI (czyli przedostatnim). Podobnie, gdyby k było równe 2, wstawilibyśmy $i+1$ na przed-przedostatnim miejscu LISTY POZYCJI, itd. Reasumując, dla każdej wczytanej wartości k, znajdującej się na i-tym miejscu naszej listy wejściowej, wstawiamy do LISTY POZYCJI wartość i na pozycji $i-k$.

Przetwarzamy w analogiczny sposób kolejne wartości wejściowe, a na koniec LISTA POZYCJI będzie zawierać pozycje dzieci o kolejnych numerach. Wystarczy ją odwrócić (tzn. wypisać numery dzieci o kolejnych pozycjach).

Rozważmy przykład z treści zadania:

```
0 0 0 3 1
```

Wczytujemy pierwszą wartość (0). Oznacza ona, że pierwszy uczeń nie widzi nikogo o numerze większym niż swój, co zresztą oczywiste (jest pierwszy, więc w ogóle nikogo nie widzi). Jego pozycję (1) wstawiamy do – początkowo pustej – LISTY POZYCJI[1]:

```
LISTA POZYCJI: 1
numer dziecka: 1
```

Wczytujemy drugą wartość (0). Oznacza ona, że drugi uczeń też nie widzi nikogo o numerze większym niż swój. Musi mieć zatem największy możliwy numer, równy dotychczasowej liczbie dzieci (czyli 2). Jego pozycję (2) wstawiamy więc na końcu LISTY POZYCJI:

```
LISTA POZYCJI: 1 2
numer dziecka: 1 2
```

Analogicznie, uczeń na trzeciej pozycji też nie widzi nikogo o numerze większym niż swój. Musi mieć zatem

[1] „numer dziecka” w dolnym wierszu jest na razie tymczasowy – będziemy tam wstawiać po prostu kolejne liczby naturalne.

największy możliwy numer, równy dotychczasowej liczbie dzieci (czyli 3). Jego pozycję (3) wstawiamy znów na końcu LISTY POZYCJI:

```
LISTA POZYCJI: 1 2 3
numer dziecka: 1 2 3
```

Czwarty uczeń widzi trójkę dzieci o numerach większych, co oznacza, że wszystkie wcześniejsze, które przed nim stoją, mają numery większe, a zatem on sam musi mieć numer najmniejszy. Jego pozycję (4) wstawimy więc na samym początku LISTY POZYCJI.

```
LISTA POZYCJI: 4 1 2 3
numer dziecka: 1 2 3 4
```

Zauważmy tu, że to wstawienie spowodowało zmianę numerów pierwszej trójki dzieci (dziecko stojące przed chwilą na początku kolejki ma teraz numer 2, itd.). Tak jak zauważyliśmy wcześniej, faktyczne numery dzieci są początkowo nieistotne (uwzględnimy je dopiero na końcu).

Wczytujemy teraz ostatni element wejścia. Uczeń na piątej pozycji widzi jedno dziecko o numerze większym od własnego, co oznacza, że sam musi mieć numer przedostatni. Wstawiamy więc jego pozycję (5) na przedostatnim miejscu LISTY POZYCJI (między dwójkę i trójkę):

```
LISTA POZYCJI: 4 1 2 5 3
numer dziecka: 1 2 3 4 5
```

Na koniec wystarczy wypisać numery dzieci (z dolnego wiersza) w kolejności ich pozycji na LIŚCIE POZYCJI:

```
LISTA POZYCJI: 1 2 3 4 5
numer dziecka: 2 3 5 1 4 <- LISTA WYNIKOWA DO WYPISANIA NA WYJŚCIE
```

Zauważmy, że odpowiada to posortowaniu dzieci wg ich pozycji na LIŚCIE POZYCJI (co oczywiście możemy tutaj uzyskać dużo prościej, bez użycia algorytmu sortowania, po prostu wstawiając od razu kolejne elementy na właściwe miejsce).

Na kolejnej stronie prezentujemy przykładową implementację w języku Python. Zauważmy, że na wstępie sprawdzamy tu, czy podane wejście jest poprawne (wiersz 5), tzn. czy któreś dziecko przypadkiem nie twierdzi, że widzi więcej dzieci niż faktycznie przed nim stoi (abstrahując od tego czy mają numery mniejsze czy większe od niego). Właściwy algorytm to pętla w wierszu 9, a jej podstawowym zadaniem jest wstawianie kolejnych elementów na odpowiednie miejsca naszej LISTY POZYCJI (tablicy `tt`), zgodnie z podanym wcześniej opisem. W wierszu 11 inicjalizujemy zerami tablicę wynikową `odp`, którą następnie wypełniamy właściwymi wartościami w odpowiedniej kolejności (wiersz 13). W przypadku naszego przykładu najpierw na pozycji 4 wpiszemy 1, następnie na pozycji 1 wpiszemy 2, na pozycji 2 wpiszemy 3, na pozycji 5 wpiszemy 4, a w ostatnim kroku na pozycji 3 wpiszemy 5.

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gnomy/solution.py.

```python
def solve():
  nn = int(input())
  wejscie = [int(i) for i in input().split()]
  for i in range(nn):
    if wejscie[i]>i:
      print("NIE")
      return
  tt = []
  for i in range(nn):
    tt.insert(i-wejscie[i], i)
  odp = [0]*nn
  for i in range(nn):
    odp[tt[i]] = i
  print(' '.join(str(i+1) for i in odp))

if __name__=="__main__":
  solve()
```

Analogiczna implementacja w języku C++ wygląda następująco:

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int nn;
  cin >> nn;
  vector<int> dane(nn);
  for(int i = 0; i < nn; i++) {
    cin >> dane[i];
    if(dane[i] > i) {
      cout << "NIE\n";
      return 0;
    }
  }

  vector<int> tt;
  for (int i = 0; i < nn; i++) {
    tt.insert(tt.begin() + i - dane[i], i);
  }
  vector<int> odp(nn, 0);
  for(int i = 0; i < nn; i++)
    odp[tt[i]] = i + 1;
  for(int i : odp)
    cout << i << " ";
  cout << endl;
  return 0;
}
```

27 (III) – Kto stoi przede mną – Skrzaty

Autorzy: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap regionalny)
Język programowania: Scratch

27.1 Treść zadania

Brajanek wrócił ze szkoły bardzo podekscytowany (chłopiec uczęszcza do Królewskiej Szkoły Konkwistadorów i Matadorów w Bitawie – elitarnej placówki dla szlachetnie urodzonych młodzieńców). Halinka, mama Brajanka, zaczęła go wypytywać o szkolne wrażenia.

– *Co robiliście na informatyce?* – zagadnęła syna.

– *Ustawialiśmy się jeden za drugim i wyciągaliśmy włoski!* – pochwalił się Brajanek – *to jest: wnioski!*

– *Jakie wnioski?* – zdziwiła się Halinka.

– *Patrzyliśmy, kto stoi przed nami i Kewinek musiał powiedzieć, w jakiej kolejności stoimy!* – wyjaśniał Brajanek.

– *Ale jak to, czy Kewinek nie mógł po prostu wymienić kolejnych uczniów stojących w rzędzie?* – dopytywała się mama.

– *No nie mógł, bo go w szafie zamknęliśmy!* – roześmiał się Brajanek.

Sprawa miała się tak: uczniowie (ponumerowani od 1 do n) rzeczywiście ustawili się losowo w kolejce tak, że każda osoba widziała tylko kolegów stojących przed nią. Na przykład dla $n = 5$ mogło to wyglądać następująco:

$$\leftarrow \mathbf{2} \quad \leftarrow \mathbf{3} \quad \leftarrow \mathbf{5} \quad \leftarrow \mathbf{1} \quad \leftarrow \mathbf{4}$$

Wszyscy uczniowie zwróceni są w lewą stronę, zatem uczeń o numerze **2** nie widzi nikogo, uczeń **3** widzi tylko ucznia **2**, uczeń **5** widzi uczniów **2** i **3**, i tak dalej (Kewinek dalej siedzi w szafie).

Nauczyciel zbiera od każdego ucznia informację, ilu widzi uczniów o numerach większych od jego własnego. Na przykład uczeń o numerze **2** podaje wartość 0, podobnie jak uczniowie o numerach **3** i **5**.

Uczeń **1** podaje liczbę 3, bo widzi przed sobą uczniów **2**, **3** oraz **5** (wszyscy mają numery większe niż **1**).

Z kolei uczeń **4** podaje wartość 1, bo wśród widzianych przez niego czterech uczniów tylko uczeń **5** ma numer większy od **4**. Całość informacji wygląda zatem tak:

$$\mathbf{0} \quad \mathbf{0} \quad \mathbf{0} \quad \mathbf{3} \quad \mathbf{1}$$

Pytanie brzmi: czy Kewinek jest w stanie odtworzyć kolejność uczniów w kolejce – tylko na podstawie powyższych informacji?

– *Powiedz mi jeszcze, synku* – zapytała na koniec mama – *czy zawsze podajecie Kewinkowi poprawne i rzetelne informacje?*

– *Nie, mamo, czasem robimy go w bolka!* – uśmiechnął się chłopiec.

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania.

Program z listy `DANE` powinien wczytać informacje, jakie otrzymuje Kewinek i wyznaczyć kolejność uczniów w kolejce – lub zawyrokować, że otrzymane dane są nieprawidłowe (czyli nie opisują rzeczywistej kolejności uczniów).

Opis wejścia

Lista wejściowa `DANE` zawiera N elementów ($2 \leq N \leq 200\,000$ – to duża szkoła!). Każdy element to pojedyncza liczba naturalna mniejsza od N.

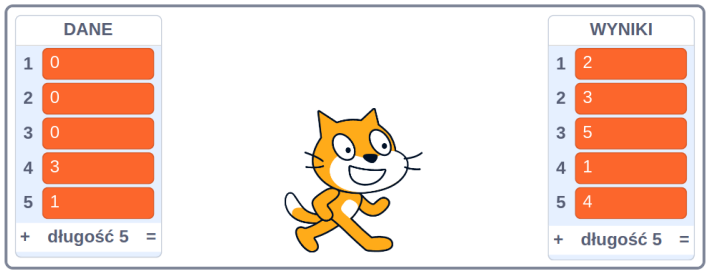
Opis wyjścia

Twój program po uruchomieniu (za pomocą zielonej flagi) powinien wypisać, na liście wyjściowej `WYNIKI`, N numerów uczniów stojących w kolejce. W przypadku, gdy dane wejściowe są nieprawidłowe, na liście `WYNIKI` należy umieścić tylko jeden element: słowo **NIE** (wielkimi literami).

Przykłady

Przykład 1

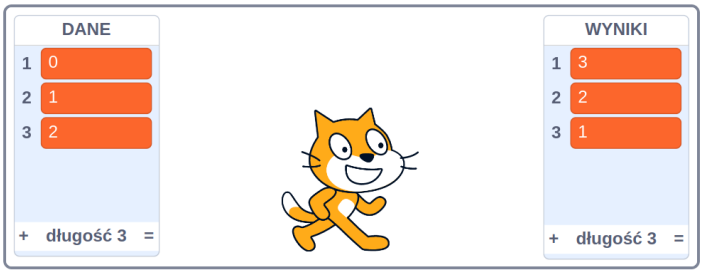
Na poniższym rysunku pokazano wynik działania programu dla przykładowych danych opisanych w treści zadania:



Rysunek 27.1: Przykładowe dane wejściowe i oczekiwane wyniki (przykład 1)

Przykład 2

Poprawne wyniki dla innych danych wejściowych pokazano poniżej:

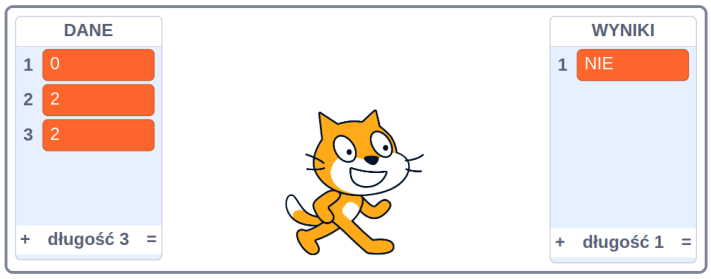


Rysunek 27.2: Przykładowe dane wejściowe i oczekiwane wyniki (przykład 2)

Jak widać, uczniowie stoją tu w kolejności malejących numerów.

Przykład 3

Jeszcze inny przykład zaprezentowano poniżej:



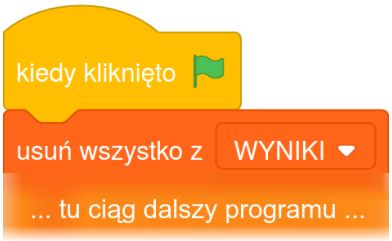
Rysunek 27.3: Przykładowe dane wejściowe i oczekiwane wyniki (przykład 3)

Dane wejściowe nie odpowiadają tu żadnej kolejności numerów trzech uczniów.

Uwagi

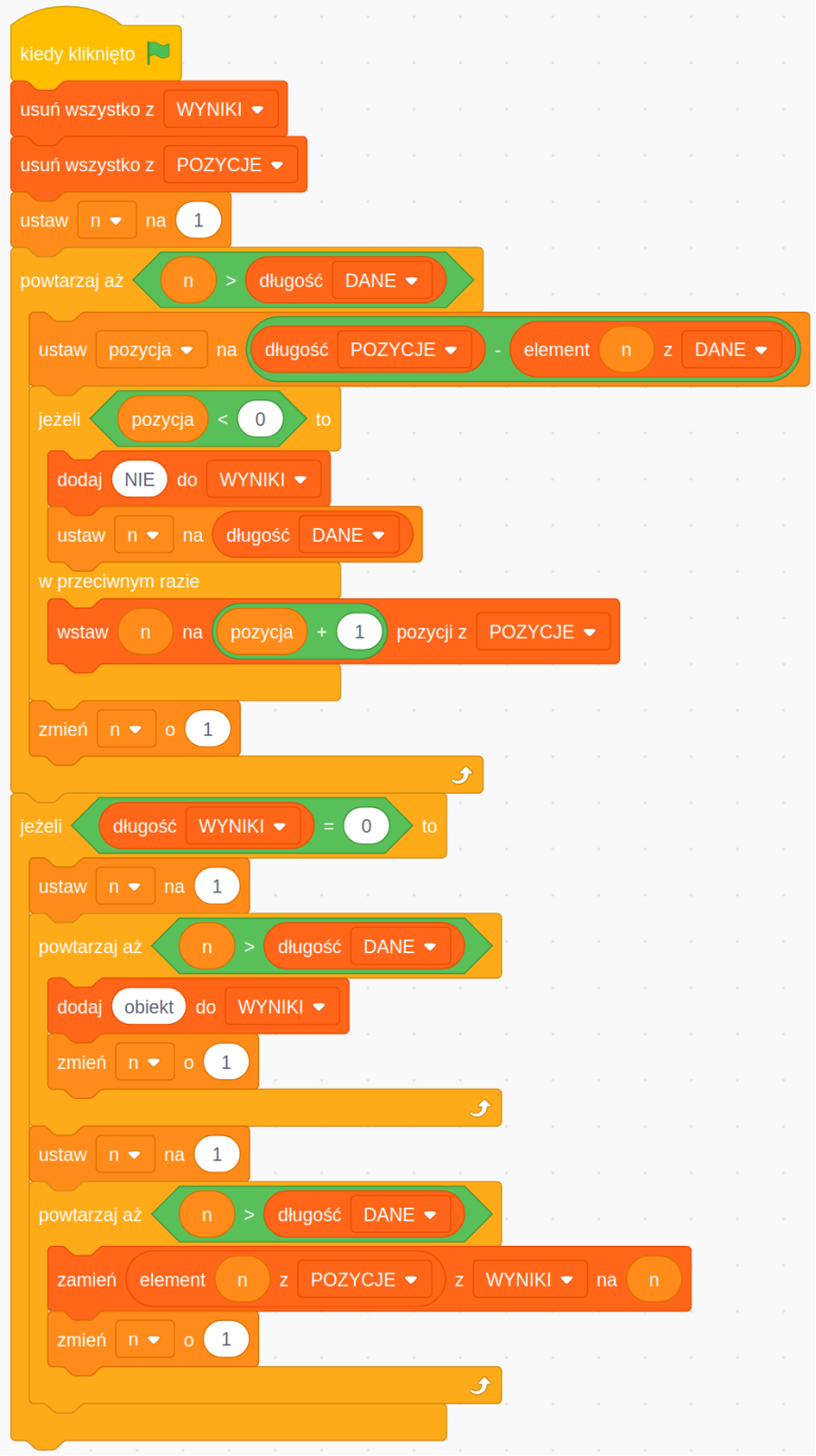
Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



27.2 Rozwiązanie

Rozwiązanie zadania „Kto stoi przede mną” w Scratchu jest oparte na tym samym algorytmie, co w przypadku rozwiązania dla Gnomów:



Rysunek 27.4: Rozwiązanie – kod programu w Scratchu

Różnice są niewielkie i nie mają istotnego znaczenia. Możemy zauważyć, że w pierwszej pętli nie tylko sprawdzamy, czy dane ustawienie jest w ogóle możliwe (jeśli nie – wypisujemy na wyjściu „NIE” i przerywamy pętlę ustawiając jej licznik `n` od razu na wartość końcową), ale jednocześnie od razu wypełniamy listę `POZYCJE`. Jest to więc w zasadzie główna część algorytmu.

Jeśli na liście `WYNIKI` nie pojawiło się „NIE”, wypełniamy ją analogicznie jak robiliśmy to w Pythonie w wierszach 12,13 (w wersji C++: w wierszach 22, 23). Zwróćmy tu jeszcze tylko uwagę na specyficzny sposób inicjalizacji listy wynikowej w każdym z języków. W języku Python używaliśmy (wiersz 11) konstrukcji `[0]*nn`, oznaczającej utworzenie listy zawierającej `nn`-krotnie powtórzoną wartość `[0]`. W C++ użyliśmy konstruktora struktury `vector` z dwoma argumentami oznaczającymi liczbę elementów i wartość inicjującą (wiersz 21). Tu natomiast wypełniamy ręcznie listę `WYNIKI` napisami „obiekt”, choć oczywiście można by w tym celu użyć dowolnego innego napisu bądź liczby. W każdym z tych języków cel jest taki sam – aby lista/tablica wynikowa miała właściwą długość zanim zaczniemy wpisywać do niej obliczone wartości (bo – jak pokazaliśmy w omówieniu rozwiązania dla Gnomów – kolejność wpisywania może być dowolna).

Przedstawiony wyżej skrypt znajdziesz w pliku:

./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Skrzaty/solution.sb3.

28 (III) – Kto stoi przede mną – Gremliny

Autorzy: ***Andrzej Dyrek***, ***AGH w Krakowie***, ***Andrzej Jastrzębski***, ***Politechnika Gdańska***
Kategoria: ***Gremliny (IV edycja – zawody algorytmiczne – etap regionalny)***
Język programowania: ***C++/Python***

28.1 Treść zadania

Brajanek wrócił ze szkoły bardzo podekscytowany (chłopiec uczęszcza do Królewskiej Szkoły Konkwistadorów i Matadorów w Bitawie – elitarnej placówki dla szlachetnie urodzonych młodzieńców). Halinka, mama Brajanka, zaczęła go wypytywać o szkolne wrażenia.

– *Co robiliście na informatyce?* – zagadnęła syna.

– *Ustawialiśmy się jeden za drugim i wyciągaliśmy włoski!* – pochwalił się Brajanek – *to jest: wnioski!*

– *Jakie wnioski?* – zdziwiła się Halinka.

– *Patrzyliśmy, kto stoi przed nami i Kewinek musiał powiedzieć, w jakiej kolejności stoimy!* – wyjaśniał Brajanek.

– *Ale jak to, czy Kewinek nie mógł po prostu wymienić kolejnych uczniów stojących w rzędzie?* – dopytywała się mama.

– *No nie mógł, bo go w szafie zamknęliśmy!* – roześmiał się Brajanek.

Sprawa miała się tak: uczniowie (ponumerowani od 1 do n) rzeczywiście ustawili się losowo w kolejce tak, że każda osoba widziała tylko kolegów stojących przed nią. Na przykład dla $n = 5$ mogło to wyglądać następująco:

$$\leftarrow \mathbf{2} \quad \leftarrow \mathbf{3} \quad \leftarrow \mathbf{5} \quad \leftarrow \mathbf{1} \quad \leftarrow \mathbf{4}$$

Wszyscy uczniowie zwróceni są w lewą stronę, zatem uczeń o numerze **2** nie widzi nikogo, uczeń **3** widzi tylko ucznia **2**, uczeń **5** widzi uczniów **2** i **3**, i tak dalej (Kewinek dalej siedzi w szafie).

Nauczyciel zbiera od każdego ucznia informację, ilu widzi uczniów o numerach większych od jego własnego. Na przykład uczeń o numerze **2** podaje wartość 0, podobnie jak uczniowie o numerach **3** i **5**.

Uczeń **1** podaje liczbę 3, bo widzi przed sobą uczniów **2**, **3** oraz **5** (wszyscy mają numery większe niż **1**).

Z kolei uczeń **4** podaje wartość 1, bo wśród widzianych przez niego czterech uczniów tylko uczeń **5** ma numer większy od **4**.

Całość informacji wygląda zatem tak:

$$\mathbf{0} \quad \mathbf{0} \quad \mathbf{0} \quad \mathbf{3} \quad \mathbf{1}$$

Pytanie brzmi: czy Kewinek jest w stanie odtworzyć kolejność uczniów w kolejce – tylko na podstawie powyższych informacji?

– *Powiedz mi jeszcze, synku* – zapytała na koniec mama – *czy zawsze podajecie Kewinkowi poprawne i rzetelne informacje?*

– *Nie, mamo, czasem robimy go w bolka!* – uśmiechnął się chłopiec.

Zadanie

Napisz program, który wczyta informacje, jakie otrzymuje Kewinek i wyznaczy kolejność uczniów w kolejce – lub zawyrokuje, że otrzymane dane są nieprawidłowe (czyli nie opisują rzeczywistej kolejności uczniów).

Opis wejścia

Dane wejściowe składają się z dwóch wierszy.

Pierwszy wiersz zawiera liczbę naturalną N oznaczającą ilość uczniów w kolejce ($2 \leq N \leq 1\,000\,000$ – to duża szkoła!).

W drugim wierszu znajduje się N liczb naturalnych mniejszych od N, oddzielonych pojedynczymi odstępami.

Opis wyjścia

Program powinien wypisać wiersz tekstu zawierający listę N numerów uczniów stojących w kolejce. Pomiędzy numerami należy umieścić pojedyncze spacje.

W przypadku, gdy dane wejściowe są nieprawidłowe, należy wypisać słowo **NIE** (wielkimi literami).

Przykład

Dla danych wejściowych opisanych w treści zadania:

```
5
0 0 0 3 1
```

program powinien wypisać:

```
2 3 5 1 4
```

Dla następujących danych wejściowych:

```
3
0 1 2
```

program powinien wypisać:

```
3 2 1
```

Uczniowie stoją tu w kolejności malejących numerów.

Z kolei dla danych wejściowych:

```
3
0 2 2
```

program powinien wypisać:

```
NIE
```

Dane wejściowe nie odpowiadają tu żadnej kolejności numerów trzech uczniów.

28.2 Rozwiązanie

Zadanie „Kto stoi przede mną” jest pierwszym zadaniem, które prezentujemy również w wersji przeznaczonej dla szkół ponadpodstawowych (czyli dla kategorii „Gremliny”). W kolejnych działach takich zadań będzie się pojawiać coraz więcej i będą one implementowane już tylko w językach C++ i Python. Nie oznacza to oczywiście, że nie da się ich rozwiązać w Scratchu (co widać chociażby na przykładzie obecnego zadania), ale ich implementacja może być żmudna, a często również nie dość efektywna obliczeniowo.

Treść zadania „Kto stoi przede mną” w wersji dla Gremlinów jest zasadniczo taka sama, jak w wersji dla Skrzatów i Gnomów – jedyną różnicą jest rozmiar wejścia (liczba uczniów w tym przypadku może sięgać miliona). Również sposób rozwiązania jest analogiczny, natomiast zwiększenie długości danych wejściowych ujawnia istnienie „wąskiego gardła” w zaprezentowanym wcześniej algorytmie (dla Gnomów i Skrzatów). Tym wąskim gardłem jest wstawianie kolejnych elementów do LISTY POZYCJI.

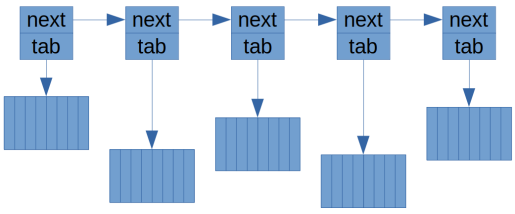
Jak pamiętamy, elementy były wstawiane w różnych miejscach LISTY POZYCJI, pomiędzy elementy wstawione wcześniej (np. w naszym przykładzie piątka była wstawiana między dwójkę i trójkę). Operacja ta ma niestety zasadniczo złożoność $O(n)$, czyli liniową. Jeśli używana przez nas struktura przechowuje dane w ciągłym obszarze pamięci, to wstawienie nowego elementu w środku wymaga przesunięcia (przepisania w pamięci) wszystkich kolejnych elementów o jedną pozycję (żeby „zrobić miejsce” dla nowego). Z drugiej strony, użycie listy wiązanej też niewiele nam pomoże, bo co prawda wstawienie nowego elementu będzie można wykonać w czasie stałym $O(1)$, ale *znalezienie odpowiedniego miejsca* do wstawienia będzie wymagać dojścia do niego element-po-elemencie, co oznacza znów złożoność $O(n)$.

Istnieją na szczęście pewne metody pozwalające na zmniejszenie złożoności obliczeniowej, choć sama ich implementacja może być nieco bardziej skomplikowana. W pierwszej kolejności rozważmy strukturę typu „lista tablic” (*arraylist*). Jest to – jak sama nazwa wskazuje – struktura stworzona z połączenia listy i tablic (stanowiących węzły tej listy). Każdy węzeł listy zawiera tu dwa pola: wskaźnik do kolejnego węzła (`next`) i tablicę (`tab`) o rozmiarze mniejszym od pewnego parametru s (wyznaczanego później):

```
class WezelListy:
    next = null
    tab = []
```

Zamiana zwykłej tablicy:

na strukturę tego typu:



wymaga odpowiedniego sposobu wykonywania operacji na niej – w szczególności wstawiania nowych elementów.

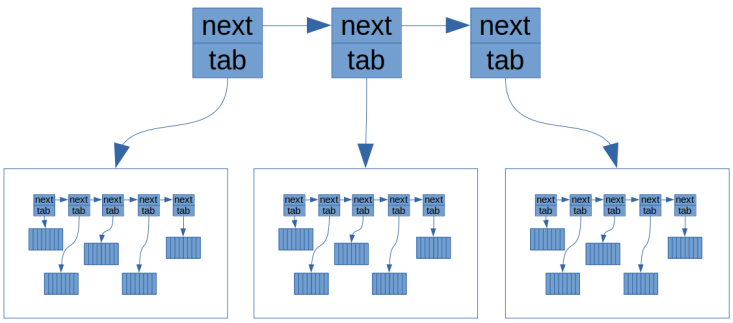
Dodanie elementu na pewną pozycję odbywa się następująco: najpierw wyszukujemy węzeł listy, do którego mamy dodać element, następnie dodajemy element do tablicy z tego węzła. Jeśli tablica, do której chcemy dodać element, osiągnęła już rozmiar s elementów, to najpierw dzielimy tablicę na dwie równe części, a następnie dodajemy element klasycznie do jednej z tych tablic. Zauważmy, że w większości przypadków tablice będą miały rozmiar pomiędzy $s/2$, a s. Oczywiście wyjątkiem jest dodanie pierwszych $s/2$ elementów – wtedy będziemy mieli listę jednoelementową z tablicą o rozmiarze mniejszym niż $s/2$.

Zauważmy, że wystarczy tylko ustalić odpowiednio wartość s, żeby algorytm działał „szybko". Niech n będzie liczbą wszystkich elementów w strukturze. Przejście przez wszystkie węzły listy będzie można wykonać maksymalnie w czasie $\frac{n}{s/2}$. Jednocześnie dodawanie do tablicy można wykonać w czasie s. Jeśli ustalimy $s = \sqrt{n}$, to przejście przez wszystkie elementy listy i dodawanie do tablicy daje złożoność $O(\sqrt{n})$.

Jedną z możliwych implementacji tej struktury (w Pythonie) przedstawiono poniżej:

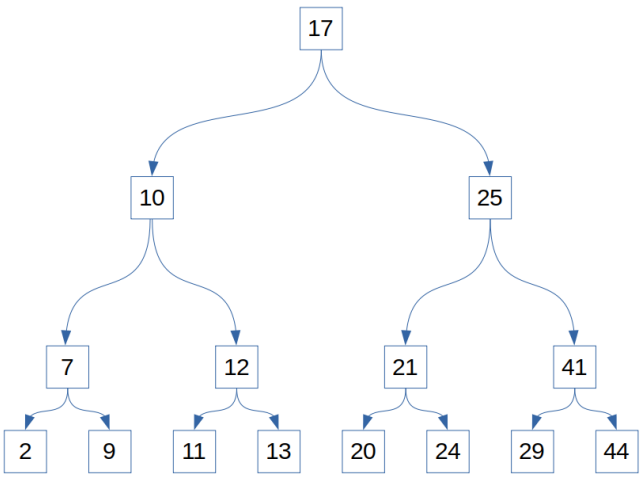
```
#v - wartość, którą chcemy dodać
#p - pozycja na którą chcemy dodać wartość
#s - ograniczenie rozmiaru tablic
#r - bieżący węzeł listy (początkowo wskazuje na pierwszy węzeł)
while len(r.tab)<p: # wyszukanie odpowiedniej listy
    p -= len(r.tab)
    r = r.next
if len(r.tab)==s: # Czy w tablicy w tym węźle skończyło się miejsce?
    tmp = WezelListy() # Tak - musimy utworzyć nowy węzeł...
    r.next, tmp.next = tmp, r.next
    r.tab, tmp.tab = r.tab[:s/2], r.tab[s/2:] #... i podzielić tablicę na pół
    #ewentualne przejście do węzła tmp
    if len(r.tab)<p:
        p -= len(r.tab)
        r = r.next
r.tab.insert(p, v)
```

Idąc dalej, możemy pomyśleć o stworzeniu bardziej zaawansowanej struktury: listy-list-tablic, czyli listy, dla której elementami węzłów są listy, dla których z kolei elementami węzłów są tablice:



Rozmiar „wewnętrznych" list jest ograniczony przez s_1, a rozmiar tablic jest ograniczony przez s_2. Odpowiednio ustalając ograniczenia tj. $s_1 = s_2 = \sqrt[3]{n}$ możemy uzyskać złożoność dodawania elementu do tej struktury

rzędu $O(\sqrt[3]{n})$. Takich struktur z zagłębieniami list w listach, w listach... możemy robić arbitralnie wiele, aż dojdziemy do struktury podobnej do *drzewa* [7][2][4][22][17]:



Drzewa stanowią w istocie bardzo ważną strukturę danych, wykorzystywaną często w bibliotekach standardowych różnych języków programowania np. do wydajnej implementacji słowników (`dict`), zbiorów (`set`) lub kolejek priorytetowych. Implementacje oparte na drzewach mają zwykle złożoność $O(\log n)$.

Przedstawione wyżej drzewo jest przykładem *drzewa wyszukiwań binarnych* (ang. *binary search tree*, BST), często wykorzystywanego w praktyce do przechowywania (i szybkiego wyszukiwania) danych. Nie wchodząc zbytnio w szczegóły[1], możemy zauważyć, że drzewo to – rosnąc oczywiście „w dół", jak większość drzew w informatyce – rozgałęzia się w każdym węźle na dwa *poddrzewa* (dlatego nazywamy je drzewem *binarnym*), przy czym wszystkie wartości w lewym poddrzewie są mniejsze, a w prawym – większe, niż w wartość w tym węźle. Jak łatwo zauważyć, umożliwia to szybkie wyszukiwanie danych – przykładowo, chcąc znaleźć wartość 21, rozpoczynamy od góry (czyli od *korzenia* naszego drzewa), a ponieważ $21 > 17$, przechodzimy do prawego poddrzewa. W korzeniu tego poddrzewa znajdujemy wartość 25, a ponieważ $21 < 25$, więc tym razem przechodzimy do lewego poddrzewa, w którego korzeniu znajdujemy szukaną liczbę. Zauważmy, że liczba porównań jest ograniczona przez wysokość (liczbę poziomów) drzewa, która z kolei – dla naszego *zrównoważonego* drzewa[2] – zależy od logarytmu liczby przechowywanych elementów.

Oprócz szybkiego wyszukiwania, drzewo BST pozwala również na łatwe uzyskanie posortowanego ciągu swoich elementów. Zasada jest prosta: dla danego węzła podajemy zawsze najpierw zawartość jego lewego poddrzewa, potem wartość samego węzła, a na końcu zawartość jego prawego poddrzewa. Podawanie zawartości poddrzew także przebiega wg tej zasady (czyli stosujemy ją w sposób *rekurencyjny*). Startujemy zawsze od korzenia, czyli – dla powyższego drzewa – od elementu 17. Jednak zanim go wypiszemy, musimy podać jego lewe poddrzewo, o korzeniu 10, ale tu też najpierw „wchodzimy do lewego poddrzewa" (o korzeniu 7), którego lewe poddrzewo składa się już tylko z pojedynczego elementu (2), więc możemy go po prostu wypisać. Następnie – w myśl naszej zasady – wypiszemy zawartość korzenia (7), a potem pojedynczy element z prawego poddrzewa (9). Te trzy wypisane dotąd elementy (2, 7, 9), to zawartość lewego poddrzewa o korzeniu 10, który teraz wypisujemy, a następnie wchodzimy do prawego poddrzewa i – analogicznie wypisujemy jego zawartość (11, 12, 13).

[1] O drzewach – traktowanych jako szczególny przypadek *grafu* – powiemy trochę więcej (i w bardziej formalny sposób) w dziale VIII.

[2] W praktyce zdarza się, że lewe i prawe poddrzewo różnią się rozmiarem – takie „niezrównoważenie" drzewa jest zwykle zjawiskiem niepożądanym.

Mając zatem wypisane:

2, 7, 9, 10, 11, 12, 13;

czyli całe lewe poddrzewo, dopisujemy wartość korzenia (17), a następnie – analogicznie wypisujemy prawe poddrzewo, otrzymując ostatecznie:

2, 7, 9, 10, 11, 12, 13, 17, 20, 21, 24, 25, 29, 41, 44.

Ta kolejność „przechodzenia" elementów drzewa binarnego określana jest jako porządek *inorder* [7]. Przyjmując następującą definicję węzła w języku C++:

```
struct Wezel {
        int val;
        Wezel* left;
        Wezel* right;
};
```

procedurę wypisywania elementów w porządku *inorder* możemy zaimplementować następująco:

```
void inorder(Wezel* w) {
        if(w == NULL) return;
        inorder(w->left);  // wywołanie (rekurencyjne) dla lewego poddrzewa
        cout << w->val;    // przetwarzanie bieżącego węzła
        inorder(w->right); // wywołanie (rekurencyjne) dla prawego poddrzewa
}
```

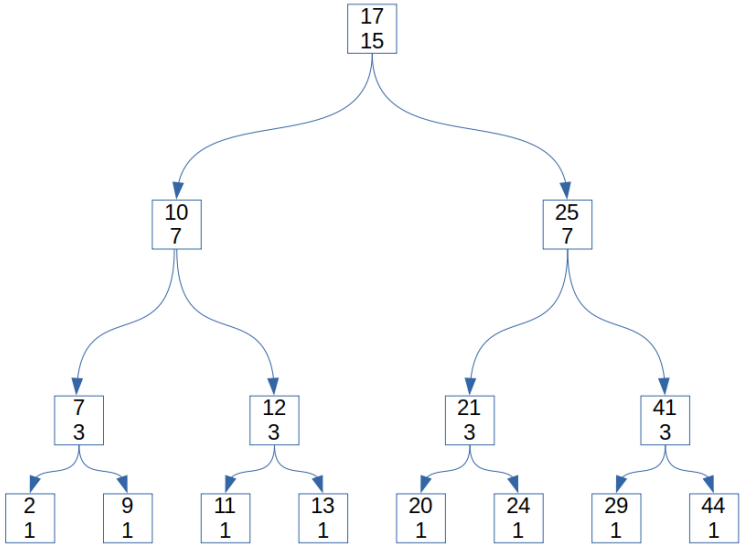
W pewnych zastosowaniach przydatne może być przechodzenie drzewa binarnego w innym porządku, np. *preorder* (najpierw bieżący węzeł, a potem poddrzewa) lub *postorder* (najpierw poddrzewa, a potem bieżący węzeł). Więcej na temat różnych algorytmów odwiedzania węzłów powiemy w dziale VIII, poświęconym grafom.

Wstawianie bądź usuwanie elementów z drzewa BST jest operacją nieco bardziej złożoną, w szczególności z uwagi na dążenie do zachowania zrównoważenia drzewa. Możliwe są tu różne podejścia, a w zasadzie różne rodzaje drzew, takie jak drzewa AVL, czy drzewa *czerwono-czarne*. Zainteresowanych czytelników odsyłamy do literatury [4][22][7], gdyż szczegółowe omówienie tych zagadnień wykracza poza zakres niniejszej książki.

Umiejętność wykorzystania struktur drzewiastych przydaje się do rozwiązywania niektórych trudniejszych problemów algorytmicznych, w szczególności jeśli biblioteki standardowe języka nie oferują gotowej implementacji lub specyfika zadania wymaga jakiejś szczególnej funkcjonalności. Do ważnych typów drzew zaliczmy m.in. *kopce* (ang. *heap*) [2][7][4], charakteryzujące się określoną relacją między każdą parą węzłów połączonych ze sobą bezpośrednio (przykładowo, w kopcu typu „max", wartość danego węzła jest zawsze mniejsza niż wartość węzła znajdującego się bezpośrednio nad nim). Kopiec również może być reprezentowany w postaci drzewa binarnego, ale w ogólnym przypadku liczba poddrzew w każdym węźle może być większa niż dwa. Niektóre rodzaje drzew (np. *B-drzewa* [22][4][7][2]) z założenia mają bardzo dużo poddrzew o stosunkowo niewielkiej wysokości, co w połączeniu z określoną strukturą informacji przechowywanych w węzłach pozwala na skuteczną redukcję czasu przetwarzania danych.

Osobną kwestią jest sposób przechowywania drzew w pamięci operacyjnej. W niektórych przypadkach zamiast struktury „wiązanej”, w której poszczególne węzły przechowują wskaźniki do innych węzłów, całą zawartość drzewa można zapisać w zwykłej tablicy, co upraszcza dostęp do elementów za pomocą odpowiednio skonstruowanego sposobu indeksowania. Podejście to wykorzystuje się typowo w niektórych rodzajach drzew (np. w kopcach binarnych), zwłaszcza w sytuacji, gdy wiemy, że zawartość drzewa nie będzie się często zmieniać.

Rozwiązanie zadania „Kto stoi przede mną” wymaga, jak pamiętamy, szybkiej metody wstawiania elementów na określone pozycje. Możemy w tym celu wykorzystać tzw. *drzewo statystyk pozycyjnych*, czyli np. zwykłe drzewo binarne przechowujące dane dowolnego typu, ale wyposażone w dodatkową informację pozwalającą na określenie pozycji danego elementu [4][17]. Informacją tą jest rozmiar (liczba węzłów) każdego poddrzewa. W przypadku naszego przykładowego drzewa, wyglądałoby to tak:



Każdy węzeł zawiera tu dodatkową liczbę (tę „dolną” na rysunku) określającą rozmiar poddrzewa, którego jest korzeniem. W praktyce, do jej przechowywania w pokazanej wyżej definicji klasy `Wezel` wystarczy uwzględnić dodatkowe pole `size`. Dzięki tej informacji możemy np. szybko[3] podać, jaki element znajduje się na dowolnej zadanej pozycji `poz` (przyjmując tę samą kolejność elementów, co poprzednio):

```
1     Wezel* w = root;
2     while (true) {
3         p = w->left->size + 1; // p, to pozycja elementu w bieżącym węźle.
4         if (poz == p)          // Jeśli jest równa zadanej:
5             return w->val;     // koniec - zwróć wynik.
6         if (poz < p)           // Jeśli zadana pozycja jest mniejsza:
7             w = w->left;       // wejdź do lewego poddrzewa.
8         if (poz > p) {
9             w = w->right;      // W przeciwnym razie wejdź do prawego poddrzewa...
10            poz -= p;          // ...pomniejszając zadaną pozycję o rozmiar lewego
11        }                      // (bo wchodząc do prawego, ignorujemy w pewnym
12    }                          // sensie lewe poddrzewo i jego wszystkie elementy).
```

[3] Mówiąc ściślej: w czasie logarytmicznym.

W analogiczny sposób moglibyśmy również *wstawiać* elementy na zadane pozycje, co jest właśnie naszym celem. Jedyny problem polega na konieczności utrzymania zrównoważenia drzewa. Zauważmy, że przy wyjątkowo niefortunnej kolejności wstawiania elementów, nasze drzewo mogłoby w ekstremalnym przypadku przybrać postać listy (gdyby każdy węzeł miał tylko jedno – np. lewe – poddrzewo), co pogorszyłoby złożoność operacji na nim z logarytmicznej do liniowej. Aby temu zapobiec, możemy zastosować wspomniane wyżej drzewo AVL, wyposażone w odpowiednie mechanizmy równoważenia drzewa. Przykładowy kod tego rozwiązania znajdziesz tutaj:

./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/drzewo_avl.py,

./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/drzewo_avl.cpp.

Nie jest to jedyne możliwe podejście. Analogiczne rozwiązanie można uzyskać wykorzystując np. B-drzewa:

./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/b_drzewo.py,

./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/b_drzewo.cpp.

Trzeba jednak zauważyć, że implementacja takich drzew (AVL, drzew czerwono-czarnych, czy B-drzew) w „czasie rzeczywistym” (np. podczas zawodów, czy konkursu algorytmicznego) nastręcza pewnych problemów. Są to skomplikowane struktury, w których łatwo o błąd w implementacji.

Prostszym drzewem do zaimplementowania (przechowywanym w tablicy, a nie w strukturze „wiązanej”) jest *drzewo przedziałowe*, które pojawia się w rozwiązaniach wzorcowych wielu zadań konkursowych. Struktura drzewa przedziałowego jest kompleksowo opisana w [5]. Żeby jednak użyć tej struktury danych trzeba nieco inaczej podejść do rozwiązania naszego zadania. W omówionym wcześniej rozwiązaniu (dla Gnomów) przetwarzaliśmy listę uczniów od lewej do prawej strony. Zmieńmy teraz kierunek – od tej pory będziemy przetwarzać tę listę od strony prawej do lewej. Nadal naszym celem będzie przetworzenie każdej i-tej wartości tak, aby mieć i dzieci ustawionych we właściwej kolejności. Można zauważyć, że jeśli pierwszy uczeń (najbardziej po prawej stronie) widzi s osób większych od siebie, to jego numer jest równy $n - s$ (gdzie n jest liczbą wszystkich dzieci). Następni uczniowie z listy mają numery podobnie uzyskiwane, jedynie trzeba uwzględniać „usuwanie” poprzednich uczniów z listy. Najprostsza implementacja tego podejścia (bez uwzględniania odpowiedzi `NIE`), może wyglądać następująco:

```
nn = int(input())
infoOdUczniow = list(map(int, input().split()))
wynik, numeracja = [], list(range(1, nn+1))
for i in range(len(infoOdUczniow)-1, -1, -1):
    poz = i-infoOdUczniow[i]
    wynik.append(numeracja[poz])
    del numeracja[poz]
wynik.reverse()
print(' '.join(map(str, wynik)))
```

To rozwiązanie ma czas działania $O(n^2)$, więc jest tak „dobre”, jak rozwiązanie prezentowane dla Gnomów. Możemy jednak zredukować złożoność obliczeniową, wykorzystując w implementacji drzewo przedziałowe, jak zaprezentowano na następnych stronach.

28.2.1 Python

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/solution.py.

```python
def solve():
  nn = int(input())
  dane = [int(i) for i in input().split()]
  for i,v in enumerate(dane):
    if v>i:
      print("NIE")
      return
  agr = [] # drzewo przedziałowe (dane zagregowane)
  n,val = nn,1
  while n>1:
    agr.append([val]*n)
    n = (n+1)//2
    val *= 2
  agr.reverse()
  odp = [None] * nn
  for i,v in enumerate(reversed(dane)):
    val = nn-v
    nn -= 1
    pos = 0
    for a in agr: #wyszukanie i modyfikacja drzewa przedziałowego
      if a[pos]<val:
        val -= a[pos]
        pos += 1
      a[pos] -= 1
      pos *= 2
    odp[i] = str(pos//2+1)
  print(' '.join(reversed(odp)))

if __name__=="__main__":
  solve()
```

28.2.2 C++

Źródło: ./zadania/03_Wczytywanie_wypisywanie/Kto_stoi_przede_mna_Gremliny/solution.cpp.

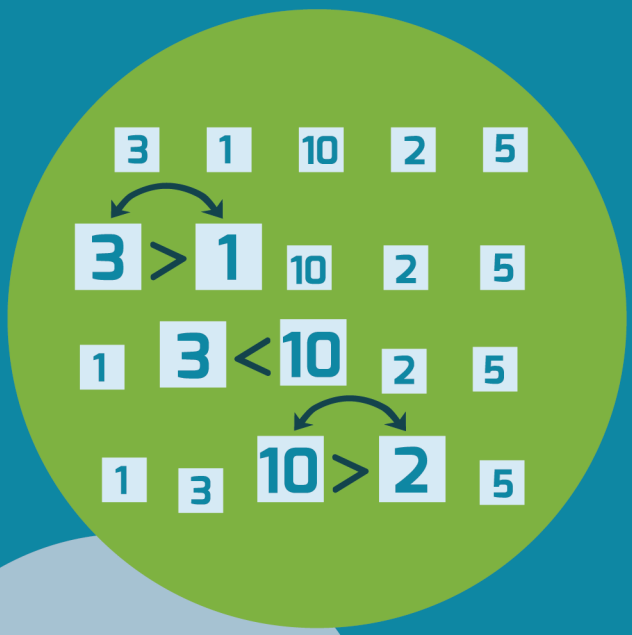
```cpp
#include <iostream>
#include <vector>
using namespace std;

void solve() {
  int nn;
  cin >> nn;
  vector<int> dane(nn);
  for(int i=0; i<nn; i++) {
    cin >> dane[i];
    if(dane[i] > i) {
      cout << "NIE\n";
      return;
    }
  }
  vector<vector<int>> agr;
  int n = nn;
  int val = 1;
  while(n>1) {
    agr.push_back(vector<int>(n, val));
    n = (n+1)/2;
    val *= 2;
  }
  vector<int> odp(nn, 0);
  for(int i=nn-1; i>=0; i--) {
    val = i - dane[i] + 1;
    int pos = 0;
    for(int ia = agr.size()-1; ia>=0; ia--) {
      vector<int> &a = agr[ia];
      if(a[pos]<val) {
        val -= a[pos];
        pos += 1;
      }
      a[pos] -= 1;
      pos *= 2;
    }
    odp[i] = pos/2+1;
  }
  int cnt = 0;
  for(int i : odp) {
    if (cnt == odp.size() - 1)
      cout << i << "\n";
    else
      cout << i << " ";
    cnt++;
  }
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  solve();
}
```

Centrum Mistrzostwa Informatycznego

Dział IV

Sortowanie



29 (IV) – Wprowadzenie

Autor: Łukasz Skonieczny, Politechnika Warszawska

Sortowanie, czyli układanie elementów w określonym porządku, jest jednym z podstawowych, a zarazem najważniejszych problemów obliczeniowych [4][2][8][22]. Nawet poza kontekstem informatycznym możemy zauważyć, że ludzie instynktownie dążą do porządkowania rzeczy w swoim otoczeniu. Warto się zastanowić, czemu tak lubimy porządek. Jest to zrozumiałe nie tylko z punktu widzenia estetyki. Dlaczego numery domów stojących przy jednej ulicy są uszeregowane w porządku rosnącym? Dlaczego przyciski pięter w windzie są ułożone w kolejności? Czemu hasła w encyklopedii lub słowniku są podane alfabetycznie? Listy obecności, książki w bibliotece, rozkłady jazdy, rankingi, karty do gry układane na ręku i tak dalej...? Dążymy do tego, ponieważ znakomicie ułatwia nam to przetwarzanie informacji. W podanych przykładach przede wszystkim chodzi o łatwe wyszukiwanie, które w uporządkowanych danych możemy zrealizować wydajnie, ale to nie jedyne zastosowanie sortowania. Mając uporządkowane dane, wiele problemów obliczeniowych staje się łatwiejszych do rozwiązania, np.: identyfikacja duplikatów (po posortowaniu duplikaty występują obok siebie), szukanie pary najbardziej podobnych obiektów (w wersji jednowymiarowej po uporządkowaniu taka para będzie wśród sąsiadujących elementów), wyznaczanie mediany, częstości wystąpień (patrz zadanie „Czapki i rękawiczki" z poprzedniego działu). Z tego powodu sortowanie jest często jednym z bloków konstrukcyjnych innych algorytmów. Przeglądanie danych w określonym porządku pozwala na uzyskanie rozwiązania prawie bezpośrednio (patrz zadanie „Bridge Club" – dalej w tym dziale) lub uniknięcie konieczności sprawdzania niepotrzebnych przypadków. Można to uzyskać np. za pomocą pewnej metody stosowanej często w kontekście geometrii obliczeniowej (p. dział V – omówienie zadania „Ogrodzenie"), zwanej „techniką zamiatania płaszczyzny" [4] (ang. *plane sweep algorithm*). Jej istota polega na przetwarzaniu tylko elementów znajdujących się przed tzw. „miotłą", czyli elementów o indeksach większych od indeksu „miotły" (w geometrii obliczeniowej, rolę „miotły" pełni często linia prosta przemieszczająca się w określony sposób po płaszczyźnie), a do elementów „zamiecionych" już się nie wraca. Technikę zamiatania wykorzystamy w tym dziale do rozwiązania zadań „Omlety" oraz „Vururuk".

Sztandarowym przykładem wykorzystania uporządkowania elementów w wydajnym przetwarzaniu jest wyszukiwanie według *klucza*. Jeśli przeszukiwane elementy są uporządkowane wg klucza, to nie musimy oglądać ich jeden po drugim, aby znaleźć ten, który nas interesuje lub stwierdzić jego brak po dobrnięciu do końca. Gdy jesteśmy proszeni o otworzenie książki na stronie 214, nie kartkujemy książki po jednej stronie od początku, tylko otwieramy mniej więcej w środku i w zależności od strony, którą zobaczymy, szukamy 214 przed lub za tą stroną za pomocą tej samej techniki, czyli kolejnego „strzału" w środek zakresu, który pozostał. Tę technikę nazywamy wyszukiwaniem binarnym i daje ona gwarancję znalezienia elementu – lub stwierdzenia jego braku – w czasie logarytmicznym (por. problem wstawiania elementu do posortowanej listy, poruszony w omówieniu zadania „Czapki i rękawiczki" dla Skrzatów, w dziale III). W realnym życiu zwykle stosujemy usprawnienie tej metody, czyli wyszukiwanie *interpolacyjne*, pozwalające na dokładniejsze „strzały" i potencjalnie szybsze odnalezienie poszukiwanego elementu (szukając w encyklopedii definicji słowa „sortowanie" otwieramy ją bardziej w końcówce niż dokładnie na środku, gdyż „s" jest pod koniec alfabetu). W programach komputerowych raczej tego nie robimy, gdyż skuteczne wyszukiwanie interpolacyjne wymaga dobrej znajomości rozkładu danych, a pomyłka w założeniach może doprowadzić do drastycznego spadku wydajności. Zwykłe

wyszukiwanie binarne jest logarytmiczne także w przypadku pesymistycznym. Wyszukiwanie interpolacyjne jest pesymistycznie liniowe.

Korzystając z sortowania do rozwiązywania zadań algorytmicznych trzeba pamiętać o kilku kwestiach. Po pierwsze, sortowanie ma zasadniczo złożoność $O(n \log n)$ i poza bardzo specyficznymi przypadkami nie da się zejść poniżej tej granicy. Z tego powodu, nie warto korzystać z sortowania, gdy istnieje szybsze rozwiązanie problemu. Należy więc uważać na przypadki, w których owszem – sortowanie znakomicie ułatwia rozwiązanie zadania, ale istnieje rozwiązanie szybsze nie wymagające wstępnego sortowania, jak na przykład znalezienie elementów największego, najmniejszego, czy nawet medianowego lub k-tego, o określonych własnościach, bądź utworzenie rankingu w stylu „top 10". Należy też uważnie czytać specyfikację danych wejściowych, gdyż zdarza się, że odczytywane na wejściu dane od razu są uporządkowane i nie wymagają dodatkowego sortowania.

Po drugie, w każdym dojrzałym języku programowania istnieją gotowe implementacje algorytmów sortowania i należy z nich korzystać, zamiast implementować je samodzielnie. Znajomość sposobu działania różnych algorytmów sortowania jest oczywiście pouczająca i warto w ramach treningu i studiów nad algorytmiką je przeanalizować i zakodować, gdyż korzystają one z wielu technik algorytmicznych przydatnych także w innych zastosowaniach. Konieczność własnej implementacji w czasie zawodów zdarza się jednak rzadko, nawet gdy wymagane jest ustalenie bardzo specyficznego porządku (jak np. w zadaniu „Porównywanie liczb") lub porządkowanie wg różnych kryteriów. Dostępne w językach programowania metody sortowania są mocno konfigurowalne i pozwalają w wygodny sposób określać kryteria sortowania. Warto więc znać dostępne narzędzia i z nich korzystać (co zresztą dotyczy także innych algorytmów i struktur danych).

Po trzecie, dobrze jest mieć świadomość, co można sortować, jak konstruować własne typy, które można sortować oraz jak definiować nietypowe, ale prawidłowe kryteria sortowania. Algorytmy sortowania bazują na możliwości porównania ze sobą dwóch dowolnych elementów w celu oceny, który z nich jest mniejszy lub czy są sobie równe. Podstawowe typy danych, takie jak liczby, teksty i daty mają oczywiście operatory porównania określone w naturalny sposób i zdefiniowane w ramach języka programowania. Dla własnych typów mamy możliwość samodzielnie określić sposób tego porównywania. W zależności od języka robi się to na różne sposoby, trzeba tylko pamiętać, że samo istnienie *komparatora* (bo tak nazywamy funkcję porównującą ze sobą dwa elementy) nie wystarczy. Rozpatrzmy dla przykładu sortowanie elementów w grze „papier-kamień-nożyce". Paper wygrywa z kamieniem, kamień z nożycami, a nożyce z papierem. Każdą parę elementów można jednoznacznie porównać (papier > kamień, kamień > nożyce, nożyce > papier), ale nie ma możliwości jednoznacznego uporządkowania tych elementów od najmniejszego do największego. Sortowanie wymaga, by komparator definiował tzw. *liniowy porządek*. W przeciwnym przypadku efekt sortowania będzie niedeterministyczny lub zakończy się błędem.

Osobnym problemem jest także utrzymanie danych w porządku w przypadku, gdy zmieniają się one w trakcie działania programu, dochodzą nowe elementy, a część jest usuwana. Ponowne sortowanie danych po każdej takiej operacji jest nieopłacalne, gdyż – przypomnijmy – każde sortowanie ma koszt przynajmniej $O(n \log n)$. Dodawanie elementów do posortowanej kolekcji (p. omówienie zadania „Czapki i rękawiczki" z poprzedniego działu), nazywane często *wsortowaniem* (ang. *insort*, czyli *insert* + *sort*), jest koncepcyjnie proste, gdyż polega na znalezieniu miejsca wstawienia za pomocą wyszukiwania binarnego $O(\log n)$, ale w większości przypadków także jest zbyt kosztowne, gdyż wstawianie w środek tablicy jest liniowe (p. omówienie ostatniego zadania: „Kto stoi przede mną" w wersji dla Gremlinów). Jeśli spodziewamy się, że nasze uporządkowane dane będą „żyły", a chcemy zachować je w uporządkowanej formie należy przechowywać je w strukturze do tego przeznaczonej, czyli na przykład w binarnym drzewie wyszukiwań, czyli drzewie BST [4][5][17][22] (ang. *binary search*

tree – p. zadanie „Vururuk”). W przypadku dynamicznie zmieniających się danych warto też zastanowić się, czy w każdym momencie potrzebujemy pełnego uporządkowania danych. Może interesuje nas tylko element w danej chwili największy lub najmniejszy (jak np. w zadaniu „Skarbce Franka Cenciaka”)? W takim przypadku znakomicie sprawdza się *kolejka priorytetowa* (najczęściej implementowana w postaci *kopca* [4][7][2] i dostępna w standardzie języka programowania).

Podsumowując, sortowanie jest kluczowym elementem w dziedzinie algorytmiki, z szerokim zakresem zastosowań i technik dopasowanych do różnych wymagań i sytuacji. Zademonstrujemy to na przykładzie zadań zebranych w niniejszym dziale, w którym – oprócz wykorzystania gotowych narzędzi – pokażemy również własną implementację niektórych metod sortowania, takich jak *sortowanie bąbelkowe* [7].

30 (IV) – Bridge Club – Gnomy *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

30.1 Task description

Bridge is a card game played by four players sitting around a table. The four players are divided into two competing teams. Pupils of the 0th Secondary School in Byteland decided to organize a bridge club and meet together in one of the classrooms to play the game.
The whole idea is supported by a teacher, Mr. Byteslaw, who wants to make sure the participants have a good time. For this purpose, he first evaluated the skill level of every pupil. Then, he decided that the skill level of a two-person team should be defined as the lower of the skill levels of these two persons. Now he wants to assign the pupils to teams and the teams to tables, in such a way, that the skill levels of both teams sitting at the same table are equal, for as many tables as possible. The problem is that Mr. Byteslaw is a history teacher and he has no idea how best to do it. Help him and create a program which will calculate how many such *fair tables* can be created.

Input description

The first line of standard input contains one natural number n, denoting the number of bridge club members ($4 \leq n \leq 100\,000$). In the second line there is a sequence of n natural numbers $z_1, z_2, ..., z_n$, separated by single spaces ($1 \leq z_i \leq 1\,000\,000\,000$). They denote the skill levels of the bridge club members.

Output description

A single integer, equal to the largest number of *fair tables* that may be formed, should be printed to standard output.

Example

For input data:

```
9
1 3 3 1 9 10 1 9 4
```

the correct result is:

```
2
```

Explanation of the example

There are many ways in which two *fair tables* may be formed here. For example, we can create teams: $(1, 1), (1, 9), (3, 4), (3, 10)$. Then we have one pair of teams with skill level 1 and another pair of teams with skill level 3.

30.2 Rozwiązanie

Stolik brydżowy składa się z czterech osób. Z warunków zadania łatwo zauważyć, że stolik jest uczciwy, gdy co najmniej dwie z tych osób mają równy poziom umiejętności i jednocześnie jest on nie większy niż poziom umiejętności pozostałych osób przy stoliku. Czyli pisząc trochę dokładniej: uczciwy stolik ma postać (A, A, B, C), gdzie $A \leq C$ i $A \leq D$. Taki stolik dzielimy wtedy na dwa zespoły (A, B) oraz (A, C) o równych umiejętnościach.

Zauważmy, że dwie największe liczby w tablicy wejściowej (czyli dwie osoby o najwyższych umiejętnościach) nie nadadzą się w miejsce A, za to na pewno nadadzą się do dowolnego uczciwego stolika jako B i C. To sugeruje, aby tablicę przeglądać w kolejności malejącej i odkładać napotkane osoby do zbioru kandydatów na B i C. Dopiero gdy napotkamy na dwie osoby o tych samych umiejętnościach (w posortowanej malejąco tablicy będą obok siebie), używamy ich jako A, pod warunkiem, że na liście kandydatów dla B i C są co najmniej dwie osoby. Tak naprawdę nie interesują nas składy stolików, a jedynie ich liczba, więc zamiast zbioru kandydatów wystarczy nam licznik.

Dominującym składnikiem w rozwiązaniu będzie sortowanie, skąd otrzymujemy złożoność naszego programu $O(n \log n)$. Pozostała część algorytmu jest liniowa, co wynika z jednokrotnego przejrzenia tablicy.

30.2.1 Python

Źródło: ./zadania/04_Sortowanie/Bridge_Club_Gnomy/solution.py.

```
n = int(input())
z = [int(x) for x in input().split()]
z.sort(reverse = True)
available = 0
previous = 0
result = 0
for skill in z:
    available += 1
    if skill == previous and available >= 4:
        result += 1
        available -= 4
        previous = 0
    else:
        previous = skill
print(result)
```

30.2.2 C++

Źródło: ./zadania/04_Sortowanie/Bridge_Club_Gnomy/solution.cpp.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  std::vector<int> z(n);
  for (int i = 0; i < n; i++) {
    std::cin >> z[i];
  }
  std::sort(z.begin(), z.end());
  int available = 0;
  int previous = 0;
  int result = 0;
  for (int i = n - 1; i >= 0; i--) {
    available++;
    if (z[i] == previous && available >= 4) {
      result++;
      available -= 4;
      previous = 0;
    } else {
      previous = z[i];
    }
  }
  std::cout << result << '\n';
  return 0;
}
```

31 (IV) – Bridge Club – Skrzaty *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Skrzaty (I edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

31.1 Task description

Bridge is a card game played by four players sitting around a table. The four players are divided into two competing teams. Pupils of the 0th Secondary School in Byteland decided to organize a bridge club and meet together in one of the classrooms to play the game.
The whole idea is supported by a teacher, Mr. Byteslaw, who wants to make sure the participants have a good time. For this purpose, he first evaluated the skill level of every pupil. Then, he decided that the skill level of a two-person team should be defined as the lower of the skill levels of these two persons. Now he wants to assign the pupils to teams and the teams to tables, in such a way, that the skill levels of both teams sitting at the same table are equal, for as many tables as possible. The problem is that Mr. Byteslaw is a history teacher and he has no idea how best to do it. Help him and create a program which will calculate how many such *fair tables* can be created.

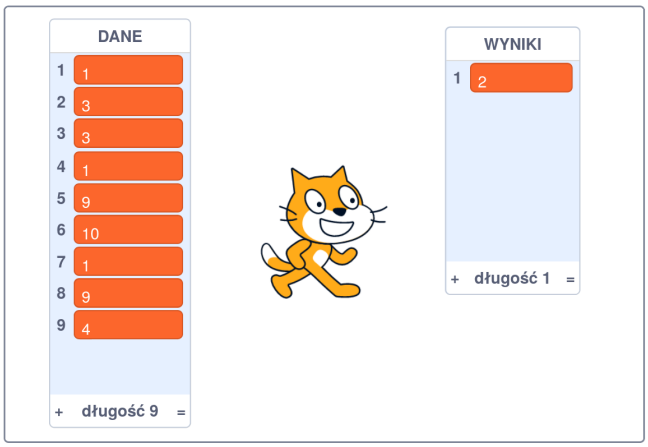
Task

Start from an empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `WYNIKI` should be initially left empty. The list `DANE` should be (manually) filled with natural numbers, with values from the range [1, 1 000 000 000], defining the skill levels of the bridge club members. The aim of the task is to create a program (a script), which will put into the list `WYNIKI` a single integer equal to the largest number of *fair tables* that may be formed.

Example

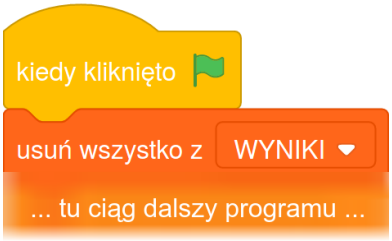


Explanation of the example

There are many ways in which two *fair tables* may be formed here. For example, we can create teams: $(1, 1), (1, 9), (3, 4), (3, 10)$. Then we have one pair of teams with skill level 1 and another pair of teams with skill level 3.

Remarks

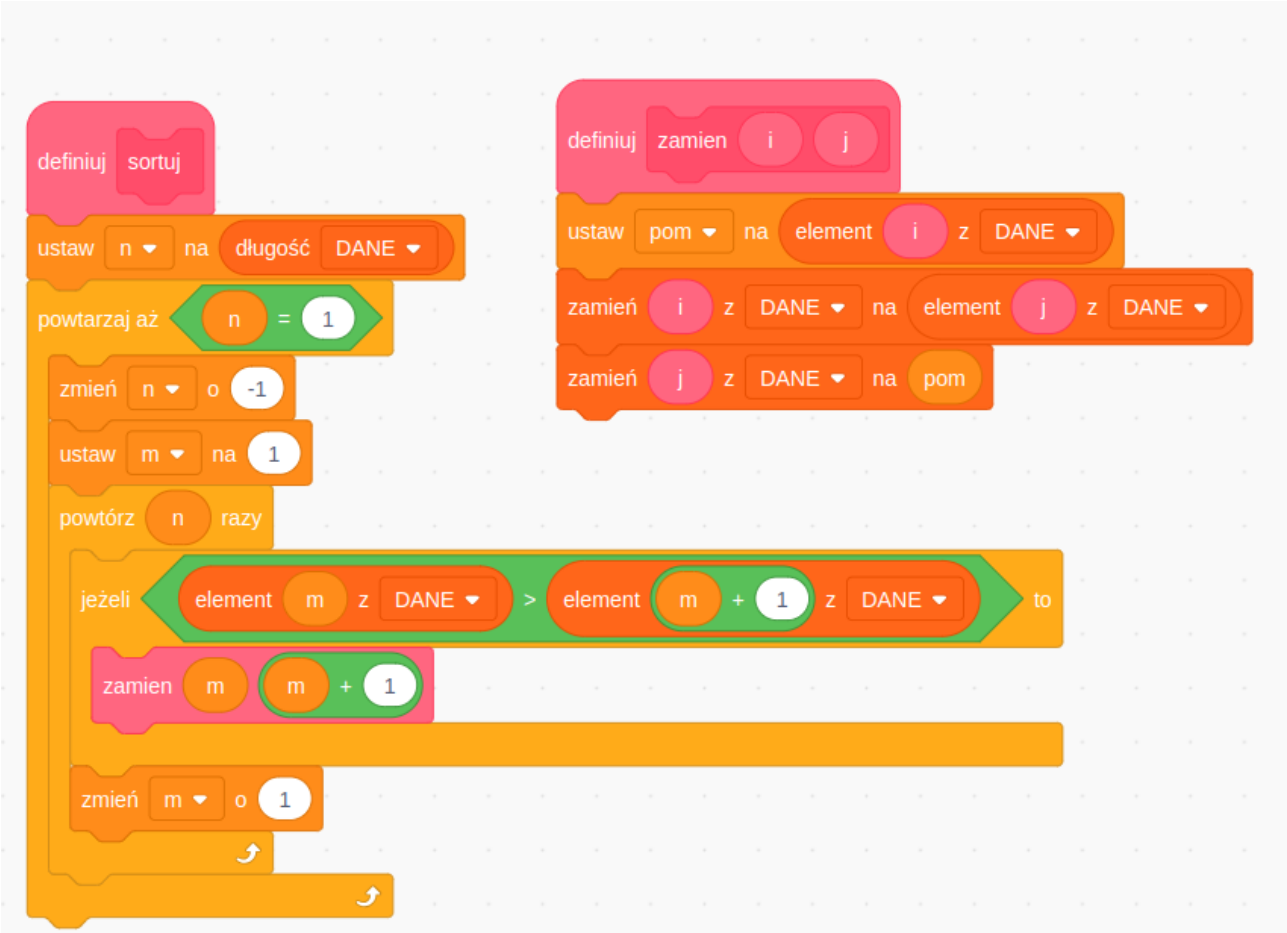
Remember, that the list `DANE` should be filled in manually, by typing in the data with the keyboard. It is worth to test the results of your program also for input values different than those in the example above. The content of the list `WYNIKI` should be always initially removed. The beginning of your program should therefore look like this:



31.2 Rozwiązanie

Rozwiązanie w wersji dla Skrzatów jest analogiczne do przedstawionej wyżej implementacji w C++ i w Pythonie... z jedną „drobną" różnicą – w Scratchu nie ma gotowej funkcji `sort`! Musimy zatem sami zaimplementować algorytm sortujący nasze dane.

Istnieje wiele znanych algorytmów sortowania, z których najlepsze mają złożoność $O(n \log n)$, jak zauważyliśmy we wprowadzeniu do tego działu. Tu jednak skorzystamy z prostszego w implementacji algorytmu *sortowania bąbelkowego* (ang. *bubble-sort*) [7], który ma złożoność $O(n^2)$ – co wystarczy na potrzeby niniejszego zadania:



Sortowanie bąbelkowe opiera się – w dużym skrócie – na sprawdzeniu czy dwa sąsiednie elementy są ustawione we właściwej kolejności i – jeśli nie – na zamianie ich miejscami. Takie sprawdzanie i ew. zamianę wykonujemy najpierw na pierwszej parze elementów z listy (element nr 1 i nr 2), potem na następnej (element 2 i 3) i jeszcze następnej i... po dojściu do ostatniej pary (element przedostatni z ostatnim) mamy gwarancję, że na ostatnim miejscu listy znalazł się element największy. Nawiązując do nazwy algorytmu – to jest ten „najcięższy bąbelek", który „opadł na dno" naszej listy[1]. W powyższym kodzie, w wewnętrznej pętli porównywane są elementy m-ty z m+pierwszym, przy czym m zmienia się od 1 do $n-1$, gdzie n jest (początkowo) ustawione na liczbę wszystkich elementów.

[1] Ktoś powie, że bąbelki powietrza w wodzie płyną w górę... Można i tak – wystarczy odwrócić kolejność sprawdzania i zamieniania elementów (najpierw przedostatni z ostatnim, potem przed-przedostatni z przedostatnim, itd.).

Niestety, wykonanie naszej wewnętrznej pętli zapewni nam tylko prawidłową pozycję tego jednego, ostatniego („najcięższego") elementu. Musimy więc powtórzyć całą procedurę, oczywiście z pominięciem ostatniego elementu, aby przedostatni element też znalazł się na swoim miejscu. W ten sposób – po odpowiedniej liczbie powtórzeń – cała lista zostanie wreszcie posortowana. Tym powtarzaniem zajmuje się zewnętrzna pętla w naszym kodzie (`powtarzaj aż n = 1`). Dwie zagnieżdżone pętle oznaczają, że złożoność sortowania jest kwadratowa względem liczby elementów.

Na koniec spytajmy jeszcze, jak właściwie zamienić miejscami dwa elementy na liście? Sprawa jest prosta, ale pamiętajmy, że jeśli np. trzymamy jeden przedmiot prawą ręką, a drugi lewą, to chcąc je zamienić potrzebujemy jeszcze półki lub stołu, na który odłożymy na chwilę jeden z nich. W naszej funkcji zamieniającej elementy (blok `zamien`), tę rolę pełni zmienna pomocnicza `pom`.

Przykładowe rozwiązanie zadania „Bridge Club" dla Skrzatów, wykorzystujące przedstawiony wyżej algorytm sortowania bąbelkowego, znajdziesz w pliku:

./zadania/04_Sortowanie/Bridge_Club_Skrzaty/solution.sb3.

32 (IV) – Omlety – Gnomy

Autor: Łukasz Skonieczny, Politechnika Warszawska
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

32.1 Treść zadania

Mieszkańcy Procesłowacji bardzo lubią jeść omlety na śniadanie. Działająca od dekad narodowa fabryka omletów produkuje masowo idealnie okrągłe omlety o znormalizowanych rozmiarach. Produkcja polega na wypuszczaniu z dyszy kropli ciasta na poruszającą się pod nią podgrzewaną taśmę. Dzięki dobraniu odpowiedniej prędkości taśmy, czasu między wypuszczeniami kropel oraz ich rozmiarów, krople spadają na taśmę w odległości 30 cm od siebie, co sprawia, że na taśmie powstają okrągłe omlety o promieniu 14 cm, oddzielone od siebie o 2 cm. Działa to bardzo dobrze, ale monotonne śniadania zaczynają Procesłowaków nużyć. Aby urozmaicić dietę, postanowiono zróżnicować wielkość produkowanych omletów. Uznano, że najłatwiej będzie to zrobić, wprowadzając pewną losowość w parametrach procesu taśmowego, a dokładniej w rozmiarach kropel oraz czasów między ich wypuszczeniami. W efekcie, krople spadają na taśmę w różnej odległości i tworzą koła o różnej wielkości. Nie przewidziano jednak, że zupełna losowość może spowodować, że kolejne omlety będą na siebie nachodziły. Takich wadliwych omletów Procesłowacy nie mogą zaakceptować, więc idą one na zmarnowanie.

Zadanie

Przy zadanych odległościach między kolejnymi kroplami ciasta (czyli środkach omletów) oraz promieniach powstałych omletów, należy znaleźć liczbę niewadliwych omletów. Omlet jest wadliwy, gdy przykrywa inny omlet (w całości lub częściowo), jest przykryty innym omletem (w całości lub częściowo), a nawet wtedy, gdy tylko dotyka innego omletu. Taśma jest na tyle długa i szeroka, że pomieści wszystkie omlety.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita: $1 \leq n \leq 1\,000\,000$, oznaczająca liczbę omletów znajdujących się na taśmie. W kolejnych n wierszach znajdują się pary liczb całkowitych $0 \leq x \leq 100\ 000\ 000$, $1 \leq r \leq 100\ 000\ 000$, oznaczające odpowiednio położenia na taśmie i promienie kolejnych omletów. Omlety są podane w kolejności, w jakiej krople spadły na taśmę, czyli według rosnących wartości x.

Opis wyjścia

Na standardowe wyjście należy wypisać liczbę niewadliwych omletów.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
3
1 3
3 2
7 1
```

prawidłową odpowiedzią jest:

```
1
```

Z kolei dla innego wejścia:

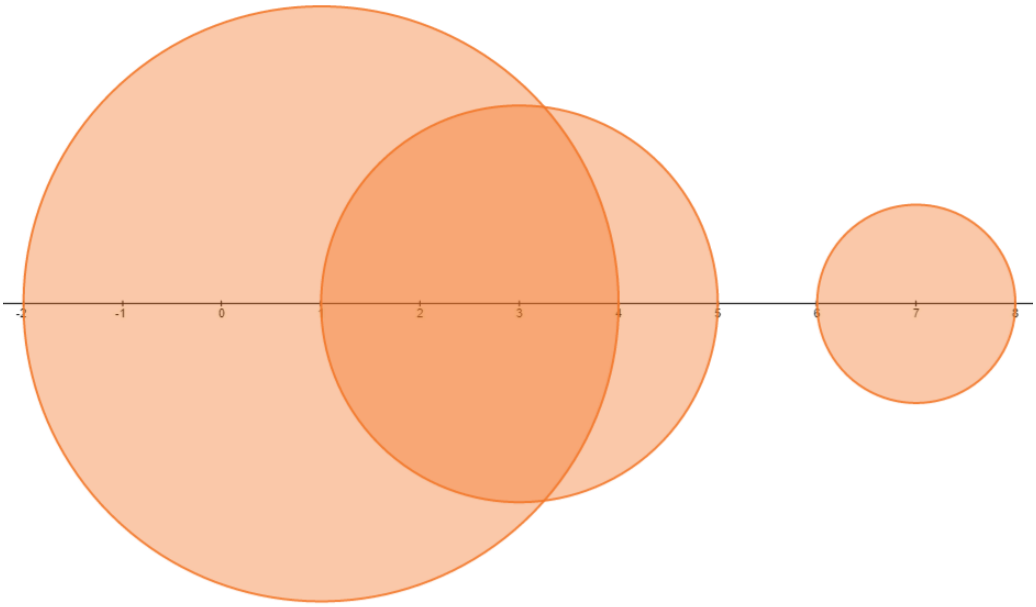
```
7
1 1
5 2
6 1
10 1
12 4
17 1
20 1
```

prawidłową odpowiedzią jest:

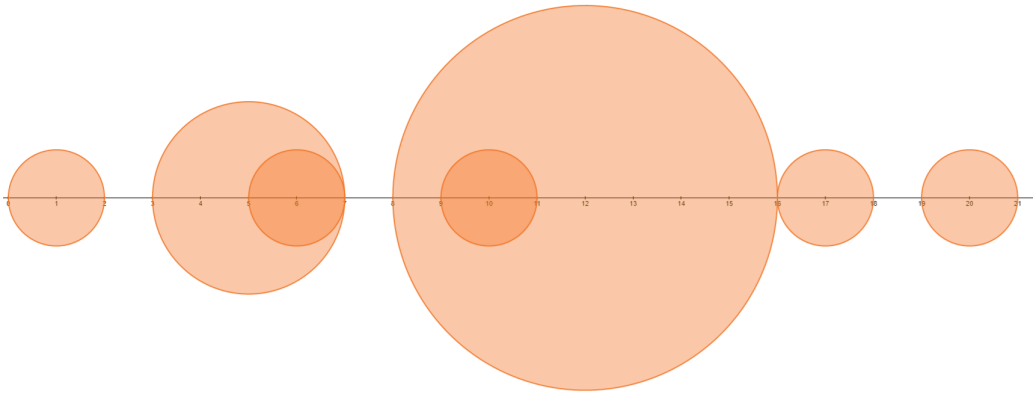
```
2
```

Wyjaśnienie przykładów

Oba przykłady są zilustrowane na rysunkach poniżej. W pierwszym przypadku mamy 3 omlety – dwa pierwsze są wadliwe, gdyż na siebie nachodzą. Trzeci nie jest wadliwy. W drugim przypadku jest 7 omletów. Tylko pierwszy i ostatni są niewadliwe. Pozostałe nachodzą na siebie lub się dotykają.



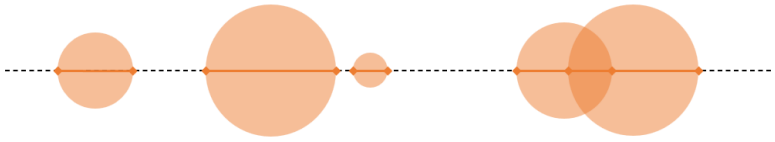
Rysunek 32.1: Przykład 1 – jeden niewadliwy omlet



Rysunek 32.2: Przykład 2 – dwa niewadliwe omlety

32.2 Rozwiązanie

W zadaniu należało znaleźć liczbę niewadliwych omletów, czyli takich, które na siebie nie nachodzą. Ponieważ omlety leżą na jednej prostej, wystarczy myśleć o omletach jak o odcinkach.



Rysunek 32.3: Omlety jako problem nakładania się współliniowych odcinków

Omletowi o środku w punkcie x oraz promieniu r odpowiada odcinek o początku w punkcie $L = x - r$ oraz końcu w punkcie $R = x + r$. Dwa odcinki $[L_1, R_1]$ oraz $[L_2, R_2]$ nie nachodzą na siebie wtedy i tylko wtedy, gdy $R_1 < L_2$ (początek drugiego odcinka leży za końcem pierwszego odcinka) lub $R_2 < L_1$ (początek pierwszego odcinka leży za końcem drugiego odcinka).

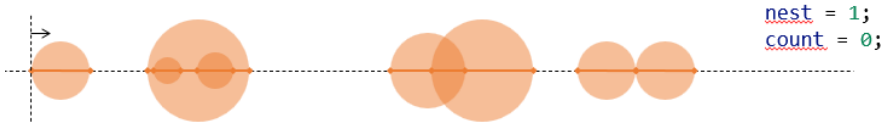
W rozwiązaniu naiwnym wystarczy rozpatrzeć każdy odcinek (czyli omlet) i dla każdego z nich sprawdzić powyższy warunek z wszystkimi pozostałymi odcinkami. Prowadzi to do rozwiązania o złożoności $O(n^2)$.

Łatwo jednak uzyskać lepsze rozwiązanie przeglądając omlety we właściwej kolejności. Zastosujemy technikę *omiatania*. Będziemy przeglądać prostą, na której leżą odcinki według rosnącej wartości x. Będziemy zatrzymywali się w każdym punkcie będącym początkiem lub końcem odcinka. Napotykając na punkt otwierający odcinek, zwiększymy o jeden *poziom zagnieżdżenia*, mówiący ile od tego momentu jest omletów na taśmie. Na początku ta wartość jest równa zero. Napotykając na punkt zamykający odcinek zmniejszamy tę wartość o jeden, gdyż w tym momencie omlet się na taśmie się skończył. Co więcej, jeżeli w tym momencie poziom zagnieżdżenia wynosi 0, a poprzednio rozpatrywany punkt należał do tego samego odcinka, mamy gwarancję, że zakończony w tym miejscu omlet jest niewadliwy.

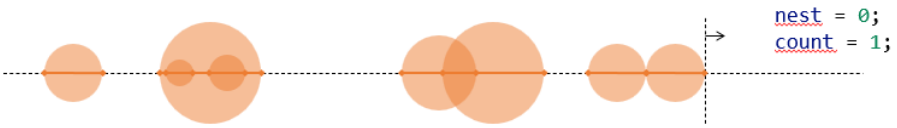
Schemat algorytmu jest następujący:

Omiataj oś X w kierunku rosnącym i zatrzymuj się na końcach odcinków. Jeśli są dwa punkty o tym samym x, to rozpatruj najpierw punkt rozpoczynający odcinek. Przy każdym zatrzymaniu naszej „miotły” w danym punkcie:

- Jeśli to punkt otwierający: *zwiększ* poziom zagnieżdżenia (`nest`) o jeden;
- Jeśli to punkt zamykający: *zmniejsz* poziom zagnieżdżenia (`nest`) o jeden;
- Jeśli punkt należy do tego samego odcinka co poprzedni oraz gdy aktualny poziom zagnieżdżenia jest równy 0: zwiększ licznik niewadliwych omletów (`count`).



Rysunek 32.4: Początek przetwarzania. Punkt otwierający omlet zwiększył aktualny poziom zagnieżdżenia o 1



Rysunek 32.5: Koniec przetwarzania. Na całej taśmie znaleziono tylko jeden niewadliwy omlet

Rozwiązanie będzie miało złożoność $O(n \log n)$, gdyż w celu uzyskania odpowiedniej kolejność przeglądania punktów musimy na początku wykonać sortowanie n punktów. Nie możemy wykorzystać kolejności podanej w danych wejściowych, gdyż tam dane były – owszem – posortowane, ale dotyczyło to środków odcinków, a nie ich początków i końców. Podczas implementacji musimy zadbać o to, by razem z każdym punktem była przechowywana informacja o odcinku, do którego należy oraz czy jest punktem otwierającym czy zamykającym odcinek. W przypadku, gdy dwa punkty leżą w tym samym miejscu (jeden odcinek kończy się dokładnie w tym samym miejscu, w którym zaczyna się kolejny) „miotła" musi w pierwszej kolejności zatrzymać się w punkcie otwierającym odcinek, gdyż w przeciwnym przypadku omlet kończący się w tym punkcie mógłby zostać uznany za niewadliwy. W rozwiązaniu wzorcowym te informacje są przechowywane w jednym polu (druga wartość pary), w postaci numeru omleta, który – jeśli dany punkt jest punktem otwierającym – zostaje zapisany ze znakiem „-" (zanegowany).

32.2.1 Python

Źródło: ./zadania/04_Sortowanie/Omlety/solution.py.

```
def main():
    n = int(input().strip())
    points = []

    for i in range(0, n):
        xs, rs = input().strip().split()
        x, r = int(xs), int(rs)
        points.append((x-r, -i-1))
        points.append((x+r, i+1))

    points.sort()
    count = 0
    prev = 0
    nest = 0
    for _, c in points:
        if c == -prev and nest == 1:
            count += 1
        if c<0:
            nest += 1
        else:
            nest -=1
        prev = c

    print(count)

if __name__ == '__main__':
    main()
```

32.2.2 C++

Źródło: ./zadania/04_Sortowanie/Omlety/solution.cpp.

```cpp
#include <algorithm>
#include <iostream>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
    int n;
    cin >>n;

    pair<int,int> *points = new pair<int,int>[2*n];

    for (int i=0; i<n; i++) {
        int x,r;
        cin >>x>>r;
        points[2*i] = make_pair(x-r, -i-1);
        points[2*i+1] = make_pair(x+r, i+1);
    }

    sort(points, points+2*n);
    int count = 0;
    int prev = 0;
    int nest = 0;
    for (int i=0; i<2*n; i++) {
        auto p = points[i];
        if (p.second == -prev && nest == 1) count++;
        if (p.second<0) nest++;
        if (p.second>0) nest--;
        prev = p.second;
    }

    cout <<count<<"\n";
    delete [] points;
}
```

33 (IV) – Skarbce Franka Cenciaka – Gnomy

Autor: Łukasz Skonieczny, Politechnika Warszawska
Kategoria: Gnomy (III edycja – liga algorytmiczna)
Język programowania: C++/Python

33.1 Treść zadania

Szmalcaria jest krainą bogatą i pełną dobrobytu. Sami mieszkańcy nazywają ją *krajem frankiem i centem płynącym*, co oczywiście ma związek z walutą używaną w Szmalcarii – są to franki podzielone na centy. Co ciekawe, waluta istnieje jedynie w postaci monet o jednostkowych nominałach – lśniących, złotych krążków w przypadku franków oraz ślicznych miedzianych jednocentówek.

Władca tej krainy – Franek Cenciak – jest właścicielem skarbców, w których trzymane są wszystkie oszczędności obywateli Szmalcarii. Skarbce są rozsiane po całym kraju i przechowują tylko franki. Skarbce są bardzo dobrze zarządzane, ale Franek Cenciak, powodowany złym snem, który ostatnio mu się przytrafił, postanowił dokładnie przeliczyć wszystkie monety w nich zawarte. W tym celu chce najpierw zgromadzić wszystkie monety w jednym skarbcu i przez ostrożność będzie to robił etapami. Każdy etap składa się z dwóch faz:

1. Przewiezienie wszystkich monet z jednego niepustego skarbca do drugiego niepustego skarbca (tak – niektóre skarbce mogą być puste).
2. Kontrolne przeliczenie wszystkich monet w połączonym skarbcu.

Aby to kontrolne przeliczenie monet w skarbcu było wykonane należycie, Franek Cenciak płaci swoim skarbnikom jednego centa za każdego przeliczonego franka. Powyższy dwufazowy etap jest powtarzany aż do momentu, w którym wszystkie franki znajdą się w jednym skarbcu. Uwaga: w przypadku, gdy w kraju znajduje się tylko jeden niepusty skarbiec, etap nie jest uruchamiany.

Zadanie

Należy pomóc Frankowi Cenciakowi zorganizować przenoszenie monet jak najmniejszym kosztem. Franek chciałby też wykazać, że nie marnotrawi bezmyślnie bogactw Szmalcarii, dlatego prosi Cię o wyznaczenie najtańszego i najdroższego sposobu na przenoszenie monet do jednego skarbca. W ten sposób będzie mógł pochwalić się potencjalnie zaoszczędzonymi pieniędzmi.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje liczba całkowita: $1 \leq n \leq 100\,000$, oznaczająca liczbę skarbców w Szmalcarii. W kolejnych n liniach znajdują liczby całkowite z zakresu $[0, 100\,000]$, oznaczające liczbę franków znajdujących się w kolejnych skarbcach.

Opis wejścia

Na standardowe wyjście należy wypisać dwie liczby zapisane w jednym wierszu i oddzielone spacją. Liczby mają oznaczać koszt w centach dla odpowiednio najtańszego i najdroższego sposobu przenoszenia monet.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
3
1
2
3
```

prawidłową odpowiedzią jest:

```
9 11
```

Z kolei dla innego wejścia:

```
4
1
1
1
1
```

prawidłową odpowiedzią jest:

```
8 9
```

Wyjaśnienie przykładów

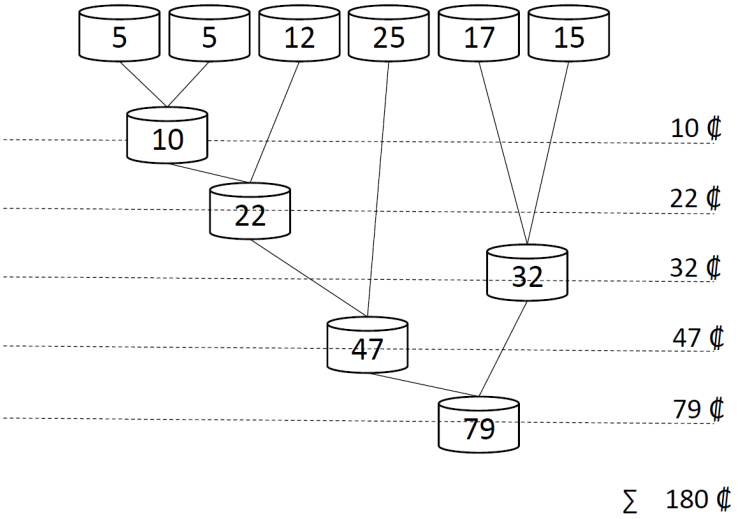
W pierwszym przypadku mamy 3 skarbce zawierające odpowiednio 1, 2 i 3 franki. Najkorzystniej będzie najpierw przenieść zawartość pierwszego skarbca do drugiego, przez co w drugim skarbcu znajdą się 3 franki, a ich przeliczenie będzie kosztowało 3 centy. W drugim kroku, trzy franki z drugiego skarbca zostaną przeniesione do trzeciego skarbca (lub odwrotnie), a przeliczenie wszystkich franków w tym skarbcu kosztować będzie 6 centów. Łącznie zostanie poniesiony koszt 3 + 6 = 9 centów. W najmniej korzystnym sposobie natomiast, przeniesiemy najpierw monety z drugiego do trzeciego skarbca (koszt 5 centów), a następnie z pierwszego do trzeciego (koszt 6 centów), co łącznie kosztować będzie 11 centów.

W drugim przypadku mamy 4 skarbce zawierające po 1 franku każdy. Najkorzystniejsze rozwiązanie polega na utworzeniu w dwóch pierwszych etapach skarbców dwu-frankowych (koszt 2 + 2 centy), a na końcu przeniesieniu 2 franków z jednego skarbca do drugiego (koszt 4 centy). Łącznie dostaniemy koszt 8 centów. W najmniej korzystnym rozwiązaniu będziemy przenosić po jednym franku do skarbca docelowego, co wygeneruje koszt 2 + 3 + 4 = 9 centów.

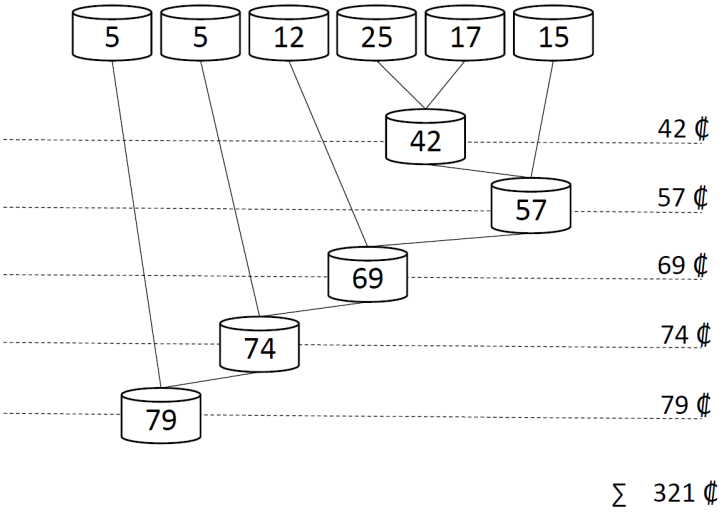
33.2 Rozwiązanie

W każdym etapie przenoszenia monet, liczba skarbców zawierających monety zmniejsza się o jeden. Jeśli na początku mamy n niepustych skarbców, to trzeba przeprowadzić $n-1$ etapów i na każdym wybrać dwa skarbce do „złączenia”. Nie ma znaczenia, czy monety przewieziemy ze skarbca A do skarbca B, czy odwrotnie. Ważne, że po złączeniu monety z obu skarbców będą w jednym z nich i że koszt łączenia tych skarbców jest sumą zawartości obu skarbców (w centach, nie frankach, ale to ma znaczenie tylko fabularne).

Na szczęście, okazuje się, że decyzje można tu podejmować według tzw. strategii *zachłannej* [4][7], o czym łatwo się przekonać wykonując kilka przykładów. Chcąc zminimalizować łączny koszt, należy w każdym etapie wybrać po prostu dwa skarbce o najmniejszej zawartości (chcąc zmaksymalizować – o największej).



Rysunek 33.1: Zachłanna procedura minimalizująca koszt. W każdym kroku do łączenia wybierane są 2 skarbce o najmniejszej wartości



Rysunek 33.2: Zachłanna procedura maksymalizująca koszt. W każdym kroku do łączenia wybierane są 2 skarbce o największej wartości

Ta strategia jest dość intuicyjna – za przeliczenie zawartości skarbców dołączanych na początku będziemy

zapewne musieli płacić wielokrotnie, więc warto, żeby były jak najmniejsze. Z drugiej strony, za przeliczenie skarbca dołączonego na końcu zapłacimy tylko raz, więc najlepiej, żeby był to skarbiec o największej wartości.

W algorytmach zachłannych podejmujemy serię decyzji, za każdym razem wybierając tę, która w danym momencie jest najlepsza według łatwego do określenia kryterium (decyzja „lokalnie optymalna"). Dlatego często się zdarza, że pierwszym krokiem algorytmu zachłannego jest posortowanie dostępnych decyzji, a następnie kolejne wybieranie ich od najlepszej do najgorszej lub ewentualne pomijanie tych, które straciły zasadność.

W naszym przypadku sytuacja jest jednak dynamiczna. Po połączeniu skarbców, z puli skarbców usuwane są dwa połączone, a dodawany jeden nowy. Możemy sortować dostępne niepuste skarbce w każdym etapie, ale byłoby to nadmiarowe. Zauważmy, że na każdym etapie musimy wybrać tylko dwa najlepsze skarbce i nie potrzebujemy pełnego posortowania pozostałych. Znalezienie dwóch elementów minimalnych (lub maksymalnych) realizujemy liniowo, co da nam łączny czas działania $O(n^2)$. Wciąż jest to jednak rozwiązanie nadmiarowe. Szukanie elementu najlepszego nie musi być wykonywane w każdym etapie od początku, gdyż nasz zbiór skarbców po każdym etapie zmienia się tylko nieznacznie.

Najwygodniej jest tu zastosować *kolejkę priorytetową*, która „na wierzchu" przechowuje element najlepszy (minimalny lub maksymalny). Większość języków programowania dostarcza tego typu kolejkę w bibliotece standardowej. Koszt stworzenia kolejki z n elementów jest liniowy, koszt pobrania elementu najlepszego (minimalnego lub maksymalnego, w zależności od rodzaju kolejki) jest logarytmiczny, a koszt dodania nowego elementu – również logarytmiczny. Łącznie (biorąc pod uwagę, że mamy $n-1$ etapów) uzyskamy rozwiązanie o złożoności $O(n \log n)$.

Na koniec, warto zauważyć, że zamiast kolejki priorytetowej można tu również zastosować drzewo BST. Będzie to rozwiązanie wolniejsze (bo na każdym etapie wymuszamy pełne posortowanie skarbców), ale wciąż pozostaniemy w złożoności $O(n \log n)$.

33.2.1 Python

Źródło: ./zadania/04_Sortowanie/Skarbce_Franka_Gnomy/solution.py.

```
from queue import PriorityQueue

def pair_join(q, n):
    r=0
    while n>1:
        a = q.get()
        b = q.get()
        q.put(a+b)
        r += a+b
        n -= 1
    return r

def main():
    n = int(input())
    size = 0
    minq = PriorityQueue()
    maxq = PriorityQueue()
    for i in range(0,n):
        v = int(input())
        if v>0:
            size += 1
            minq.put(v)
            maxq.put(-v)

    print(f'{pair_join(minq, size)} {-pair_join(maxq, size)}')

if __name__ == '__main__':
    main()
```

33.2.2 C++

Źródło: ./zadania/04_Sortowanie/Skarbce_Franka_Gnomy/solution.cpp.

```cpp
#include <functional>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

template<class T>
unsigned long long pair_join(priority_queue<unsigned long long, vector<unsigned long long>, T> &q) {
        unsigned long long r=0;
        while (q.size()>1) {
            unsigned long long a = q.top();
            q.pop();
            unsigned long long b = q.top();
            q.pop();
            q.push(a+b);
            r += a+b;
        }
        return r;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >>n;
    vector<unsigned long long> data;
    for (int i=0; i<n; i++) {
        unsigned long long v;
        cin >>v;
        if (v>0)
            data.push_back(v);
    }
    priority_queue<unsigned long long, vector<unsigned long long>,
        greater<unsigned long long>> minq(data.begin(), data.end());
    priority_queue<unsigned long long, vector<unsigned long long>,
        less<unsigned long long>> maxq(data.begin(), data.end());

    cout <<pair_join(minq)<<" "<<pair_join(maxq)<<endl;
}
```

34 (IV) – Skarbce Franka Cenciaka – Skrzaty

Autorzy: ***Łukasz Skonieczny***, ***Politechnika Warszawska***, ***Bartłomiej Stasiak***, ***Politechnika Łódzka***
Kategoria: ***Skrzaty (III edycja – liga algorytmiczna)***
Język programowania: ***Scratch***

34.1 Treść zadania

Szmalcaria jest krainą bogatą i pełną dobrobytu. Sami mieszkańcy nazywają ją *krajem frankiem i centem płynącym*, co oczywiście ma związek z walutą używaną w Szmalcarii – są to franki podzielone na centy. Co ciekawe, waluta istnieje jedynie w postaci monet o jednostkowych nominałach – lśniących, złotych krążków w przypadku franków oraz ślicznych miedzianych jednocentówek.

Władca tej krainy – Franek Cenciak – jest właścicielem skarbców, w których trzymane są wszystkie oszczędności obywateli Szmalcarii. Skarbce są rozsiane po całym kraju i przechowują tylko franki. Skarbce są bardzo dobrze zarządzane, ale Franek Cenciak, powodowany złym snem, który ostatnio mu się przytrafił, postanowił dokładnie przeliczyć wszystkie monety w nich zawarte. W tym celu chce najpierw zgromadzić wszystkie monety w jednym skarbcu i przez ostrożność będzie to robił etapami. Każdy etap składa się z dwóch faz:

1. Przewiezienie wszystkich monet z jednego niepustego skarbca do drugiego niepustego skarbca (tak – niektóre skarbce mogą być puste).
2. Kontrolne przeliczenie wszystkich monet w połączonym skarbcu.

Aby to kontrolne przeliczenie monet w skarbcu było wykonane należycie, Franek Cenciak płaci swoim skarbnikom jednego centa za każdego przeliczonego franka. Powyższy dwufazowy etap jest powtarzany aż do momentu, w którym wszystkie franki znajdą się w jednym skarbcu. Uwaga: w przypadku, gdy w kraju znajduje się tylko jeden niepusty skarbiec, etap nie jest uruchamiany.

Zadanie

W nowym, pustym projekcie Scratch, utwórz dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania.

Celem zadania jest udzielenie pomocy Frankowi Cenciakowi w organizacji przenoszenia monet jak najmniejszym kosztem. Franek chciałby też wykazać, że nie marnotrawi bezmyślnie bogactw Szmalcarii, dlatego prosi Cię o wyznaczenie najtańszego i najdroższego sposobu na przenoszenie monet do jednego skarbca (w ten sposób będzie mógł pochwalić się potencjalnie zaoszczędzonymi pieniędzmi).

Opis wejścia

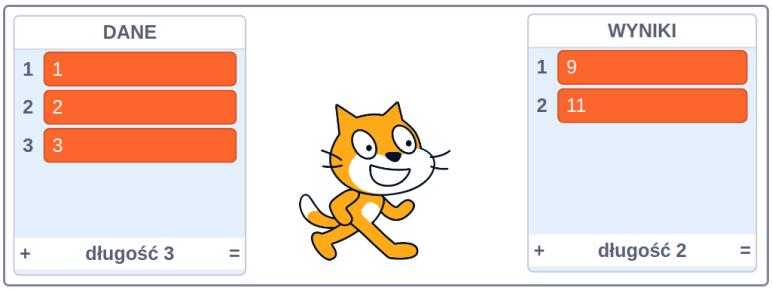
Lista `DANE` zawiera n liczb całkowitych z zakresu [0, 100 000], oznaczających liczbę franków znajdujących się w kolejnych skarbcach. Liczba skarbców n spełnia warunek: $1 \leq n \leq 10\,000$.

Opis wyjścia

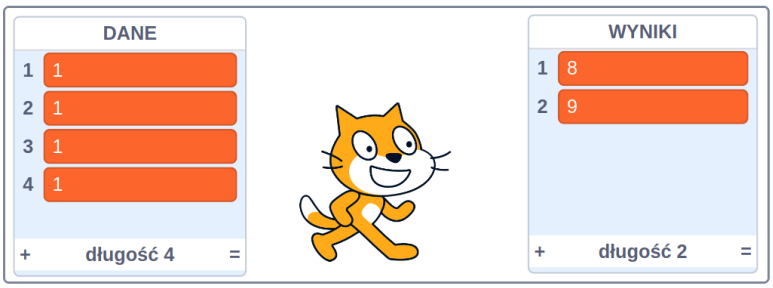
Twój program po uruchomieniu (za pomocą zielonej flagi) powinien wypisać na liście `WYNIKI` dwie liczby, oznaczające koszt w centach dla odpowiednio najtańszego i najdroższego sposobu przenoszenia monet.

Przykłady

Na poniższych rysunkach pokazano przykładowe wyniki działania programu.



Rysunek 34.1: Przykład 1: Dane wejściowe i oczekiwane wyniki działania programu



Rysunek 34.2: Przykład 2: Dane wejściowe i oczekiwane wyniki działania programu

Wyjaśnienie przykładów

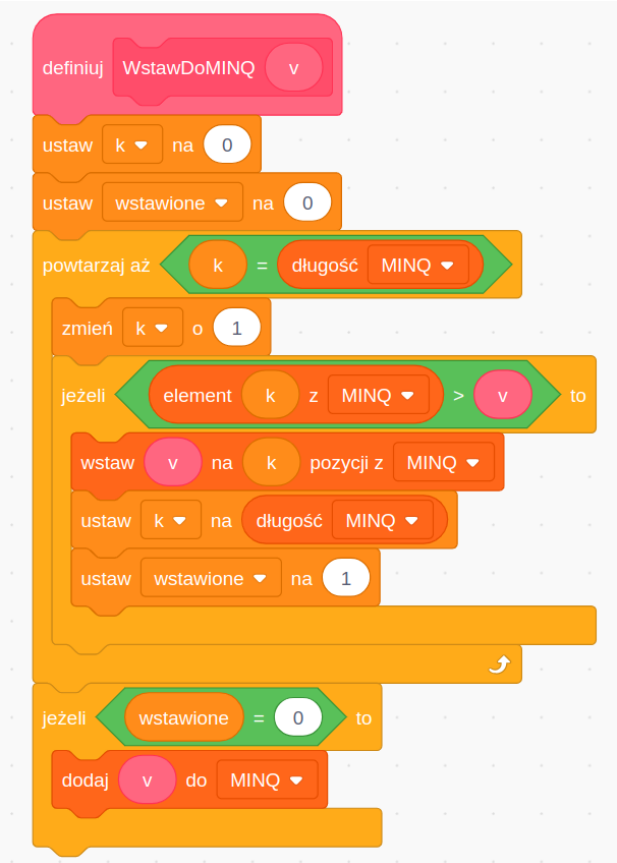
W pierwszym przypadku mamy 3 skarbce zawierające odpowiednio 1, 2 i 3 franki. Najkorzystniej będzie najpierw przenieść zawartość pierwszego skarbca do drugiego, przez co w drugim skarbcu znajdą się 3 franki, a ich przeliczenie będzie kosztowało 3 centy. W drugim kroku, trzy franki z drugiego skarbca zostaną przeniesione do trzeciego skarbca (lub odwrotnie), a przeliczenie wszystkich franków w tym skarbcu kosztować będzie 6 centów. Łącznie zostanie poniesiony koszt 3 + 6 = 9 centów. W najmniej korzystnym sposobie natomiast, przeniesiemy najpierw monety z drugiego do trzeciego skarbca (koszt 5 centów), a następnie z pierwszego do trzeciego (koszt 6 centów), co łącznie kosztować będzie 11 centów.

W drugim przypadku mamy 4 skarbce zawierające po 1 franku każdy. Najkorzystniejsze rozwiązanie polega na utworzeniu w dwóch pierwszych etapach skarbców dwu-frankowych (koszt 2 + 2 centy), a na końcu przeniesieniu 2 franków z jednego skarbca do drugiego (koszt 4 centy). Łącznie dostaniemy koszt 8 centów. W najmniej korzystnym rozwiązaniu będziemy przenosić po jednym franku do skarbca docelowego, co wygeneruje koszt 2 + 3 + 4 = 9 centów.

34.2 Rozwiązanie

Rozwiązanie zadania „Skarbce Franka Cenciaka” możemy zaimplementować w Scratchu według tego samego algorytmu, co w przypadku wersji dla Gnomów. Podstawowy problem stanowi oczywiście brak gotowej kolejki priorytetowej, którą musimy zasymulować w naszym własnym kodzie.

W tym celu tworzymy listę `MINQ` (i – analogicznie – `MAXQ`), która będzie przechowywać posortowane dane oraz specjalną funkcję `WstawDoMINQ` (oraz `WstawDoMAXQ`), która zapewni, że wstawiany element znajdzie się na właściwym miejscu (czyli – że po wstawieniu elementu lista nadal będzie posortowana). Możemy zrobić to w prosty sposób – przy każdym wstawieniu porównując wstawiany element kolejno ze wszystkimi elementami wstawionymi wcześniej, tak aby znaleźć dla niego właściwe miejsce:



Rysunek 34.3: Wstawianie elementów do MINQ – implementacja naiwna

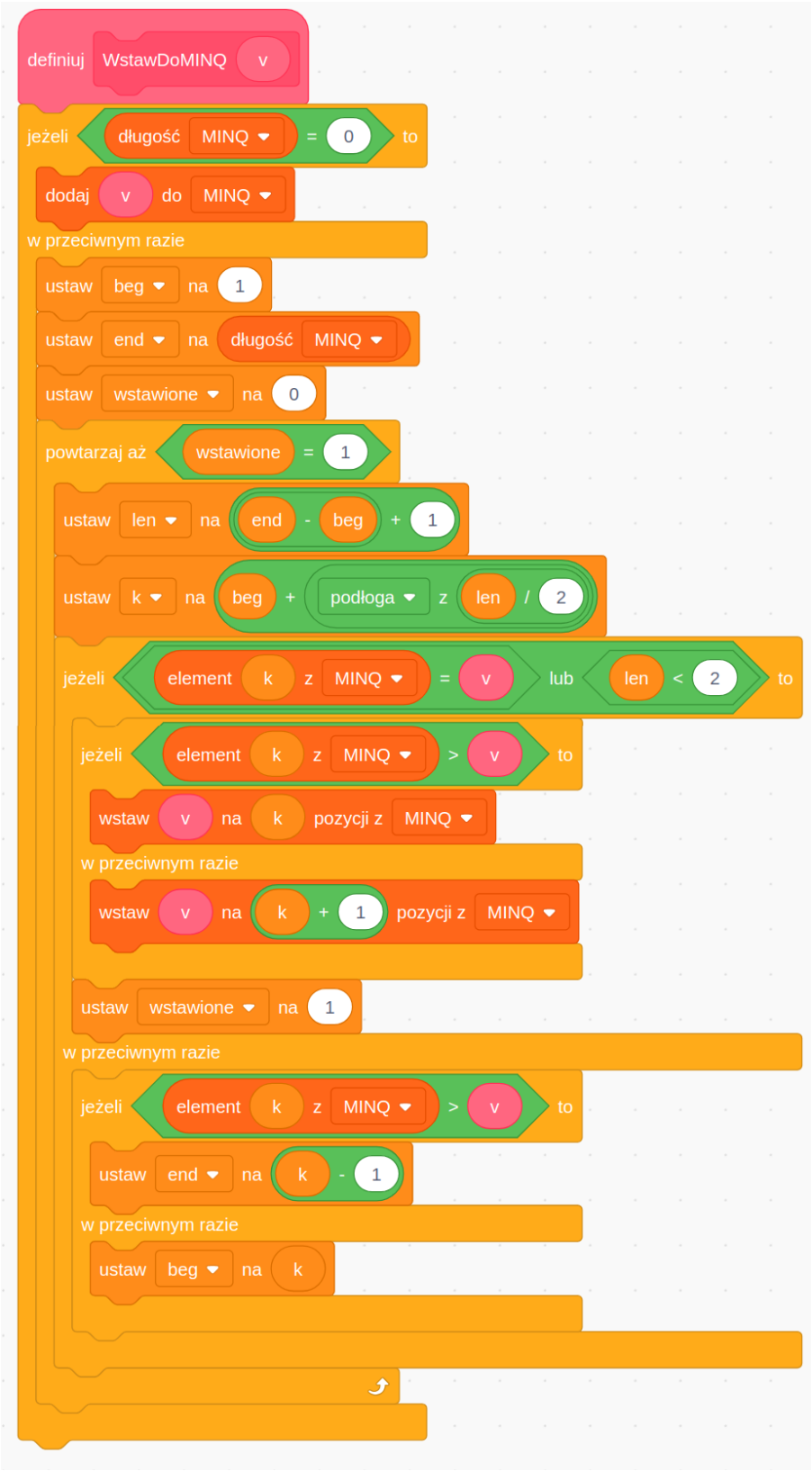
Jest to jednak podejście mało wydajne, bo wstawienie pojedynczego elementu jest liniowe ($O(n)$), a całe rozwiązanie:

./zadania/04_Sortowanie/Skarbce_Franka_Skrzaty/solutionnaive.sb3

będzie mieć złożoność $O(n^2)$ względem długości danych.

Znacząco szybszy algorytm uzyskamy, wykorzystując fakt, że lista `MINQ` jest zawsze posortowana, a więc wstawiając nowy element możemy porównać go od razu z elementem w połowie tej listy. Jeśli wstawiany element jest od niego mniejszy, to wiemy na pewno, że trzeba go wstawić gdzieś w pierwszej połowie `MINQ` (a więc drugą część listy możemy zignorować, czyli redukujemy problem o połowę). Jeśli jest większy – wstawimy go w drugiej połowie (ignorując pierwszą).

Po wybraniu właściwej połowy robimy znów to samo, czyli porównujemy wstawiany element z elementem w połowie (wybranej połowy), co redukuje nam zakres elementów pozostałych do rozważenia do $1/4$ oryginalnej długości danych i... powtarzamy tę procedurę, aż dojdziemy do zakresu jednoelementowego.



Ten sposób wstawiania jest oczywiście bardziej kłopotliwy, bo przy każdym powtórzeniu musimy pamiętać początek (`beg`) i koniec (`end`) bieżącego zakresu i odpowiednio go modyfikować, w zależności od tego czy element w połowie tego zakresu (czyli `k`-ty element listy `MINQ`) jest mniejszy, czy większy od elementu wstawianego (`v`). Jeśli jest większy – skracamy o połowę nasz zakres od prawej strony (przesuwając `end` na środek). Jeśli mniejszy – skracamy go od lewej (przesuwając `beg` na środek). Odpowiada za to ostatni fragment powyższego skryptu. Podstawową zaletą tego algorytmu jest natomiast redukcja złożoności obliczeniowej. Złożoność operacji wyszukiwania miejsca do wstawienia pojedynczego elementu – pod względem liczby porównań – wynosi $O(\log n)$, a całe rozwiązanie:

./zadania/04_Sortowanie/Skarbce_Franka_Skrzaty/solution.sb3

ma pod tym względem złożoność liniowo-logarytmiczną. Oczywiście, na ostateczną złożoność wpływa także czas samego wstawiania elementu do listy, ale na to nie mamy już wpływu (to zależy od wewnętrznej implementacji list i operacji na nich w Scratchu). Niemniej jednak – jak łatwo się przekonać – dla maksymalnego rozmiaru danych określonego przez treść zadania (w przypadku Skrzatów mamy do $10\ 000$ skarbców), czas działania naszego drugiego rozwiązania jest przynajmniej kilkunastokrotnie krótszy niż pierwszego.

Na koniec zauważmy, że – niejako przy okazji – zaimplementowaliśmy tu wydajny algorytm sortowania, znacząco szybszy w stosunku do zaprezentowanego wcześniej algorytmu *bubble-sort* (p. zadanie „Bridge Club”). Istotnie, mając dane do posortowania, wystarczy dodać je po kolei za pomocą bloku `WstawDoMINQ` do listy `MINQ`, która będzie automatycznie, na bieżąco sortowana.

35 (IV) – Szalony matematyk – Gnomy

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – liga algorytmiczna)
Język programowania: C++/Python

35.1 Treść zadania

Twój ulubiony nauczyciel matematyki niefortunnie złamał nogę, dlatego szkoła musiała zatrudnić tymczasowo kogoś nowego na zastępstwo. Lekcje z nowym nauczycielem okazują się wprawdzie ciekawe, ale zadaje on zdecydowanie zbyt skomplikowane prace domowe. Czasami wręcz wydaje się, że powinien je rozwiązywać program komputerowy, a nie uczniowie.

Ostatnia lekcja była na temat mnożenia liczb całkowitych. Nauczyciel zadał jako pracę domową następujące ćwiczenie:

„Dany jest zbiór zawierający N liczb całkowitych. Wybierz z nich k liczb tak, żeby ich iloczyn był jak największy. Jeżeli da się to zrobić na kilka sposobów, wybierz ten o największej sumie.”

Zadanie

Napisz program, który rozwiąże to zadanie za Ciebie.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajdują się dwie opisane w treści zadania liczby naturalne N, k ($2 \leq k < N \leq 500\,000$) oddzielone spacją. W drugim, ostatnim wierszu, znajduje się N liczb całkowitych $a_1, a_2, ..., a_N$ oddzielonych pojedynczymi spacjami – są to elementy zadanego zbioru w losowej kolejności. Każdy element a_i mieści się w przedziale $[-10\,000\,000, 10\,000\,000]$ i liczby nie powtarzają się.

Opis wyjścia

Na standardowe wyjście należy wypisać jeden wiersz, zawierający k liczb całkowitych wybranych z zadanego zbioru zgodnie z treścią zadania, oddzielonych pojedynczymi spacjami. Liczby te muszą być uporządkowane rosnąco.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
5 3
2 -4 6 -3 5
```

prawidłową odpowiedzią jest:

```
-4 -3 6
```

natomiast dla wejścia:

```
5 2
-1 0 4 2 -8
```

prawidłową odpowiedzią jest:

```
2 4
```

Wyjaśnienie

W pierwszym przypadku największym możliwym iloczynem trzech liczb jest $72 = (-4) \cdot (-3) \cdot 6$. Zgodnie z opisem wyjścia, liczby należy wypisać w porządku rosnącym.

W drugim przypadku największym iloczynem jest 8, ale wartość tę można osiągnąć na dwa sposoby: jako $2 \cdot 4$ lub $(-8) \cdot (-1)$. Według treści zadania, kryterium rozstrzygającym remisy ma być suma liczb, a $2+4 > (-8)+(-1)$, dlatego wybór powinien paść na liczby $2, 4$.

35.2 Rozwiązanie

W zadaniu należało z N-elementowego zbioru liczb całkowitych wybrać k ($k < N$) elementów takich, żeby ich iloczyn był jak największy. W przypadku kilku możliwych odpowiedzi, należało wybrać tę z nich o największej sumie.

Warto zacząć od spostrzeżenia, że liczb mogło być bardzo dużo, stąd ich łączny iloczyn mógłby być olbrzymi – mógłby na przykład okazać się liczbą mającą milion cyfr. Taka liczba wykraczałaby daleko poza zakres jakiegokolwiek typu wbudowanego w języku C++, natomiast w języku Python wykonywanie na niej obliczeń byłoby bardzo powolne. Należy zatem wykluczyć jakiekolwiek rozwiązanie oparte na obliczaniu tego iloczynu.

Zauważmy też, że gdyby nie liczby ujemne, to zadanie byłoby trywialne – wystarczyłoby wybrać k największych elementów zbioru. Może się wydawać, że problem pomoże rozwiązać wybranie k elementów o największej wartości bezwzględnej, tj. ignorując znaki minus, ale ten pomysł ma dużą wadę – iloczyn tak wybranych liczb może nieraz okazać się ujemny. Na przykład, dla zbioru $\{-4, -3, 2, 5, 6, 10\}$ i $k = 4$, dałoby nam to $(-4) \cdot 5 \cdot 6 \cdot 10 = -1200$ zamiast optymalnego wyboru $(-4) \cdot (-3) \cdot 6 \cdot 10 = 720$.

Ten problem da się łatwo obejść: wystarczy, abyśmy w każdym kroku wybierali dwa elementy naraz – oba albo ujemne albo dodatnie. W ten sposób gwarantujemy, że nasz iloczyn pozostanie dodatni. Wyboru dokonujemy na podstawie ich iloczynu. W powyższym przypadku, taki algorytm zadziałałby w następujący sposób:

1. Skrajne pary w zbiorze $\{-4, -3, 2, 5, 6, 10\}$ to $(-4, -3)$ oraz $(6, 10)$. Ta druga ma większy iloczyn, więc dołączamy ją do rozwiązania i wyłączamy z dalszych rozważań.
2. Skrajne pary w zbiorze $\{-4, -3, 2, 5\}$ to $(-4, -3)$ oraz $(2, 5)$. Tym razem para o ujemnych znakach ma większy iloczyn, dlatego to ją dołączamy do rozwiązania.
3. Otrzymujemy optymalne rozwiązanie $\{-4, -3, 6, 10\}$.

W przypadku, gdy para ujemna i dodatnia mają równe iloczyny, powinniśmy wybrać dodatnią, bo ta pozwoli nam osiągnąć wyższą sumę.

Taki algorytm jest szybki i na pozór skuteczny, ale ma kilka braków. Po pierwsze, jak powinien zadziałać, jeśli k jest liczbą nieparzystą? Wówczas jeden element musimy dołączyć do rozwiązania bez pary. W tym celu najlepiej na początku działania algorytmu wybrać największą liczbę dodatnią ze zbioru.

Taka odpowiedź wydaje się satysfakcjonująca, ale pojawia się tu przypadek brzegowy – co jeśli zbiór nie zawiera żadnej liczby dodatniej? Otóż jeśli k jest liczbą nieparzystą i jednocześnie zbiór zawiera tylko liczby niedodatnie, to w wyniku nie da się osiągnąć w ogóle iloczynu dodatniego i naszym zadaniem staje się znaleźć wynik „jak najmniej ujemny". Możemy wykrywać takie przypadki i odpowiadać na nie wybierając po prostu k największych liczb ze zbioru, np. jeśli zbiór to $\{-4, -3, -2, -1\}$ a $k = 3$, to optymalną odpowiedzią jest $\{-3, -2, -1\}$.

Wreszcie, rozważmy przypadek $\{-3, -2, -1, 0, 1\}$ i $k = 4$. W pierwszym kroku dołączymy do rozwiązania parę $(-3, -2)$ i pozostaje nam zbiór $\{-1, 0, 1\}$, z którego wciąż musimy wybrać 2 liczby – lecz przez obecność zera w środku, żadna para nie jest już ani dodatnia ani ujemna. Samo w sobie nie stanowi to dużego problemu – możemy wciąż dobrać parę $(0, 1)$. Otrzymujemy w ten sposób optymalny iloczyn równy 0, ale nieoptymalną sumę – mogliśmy wszak wybrać liczby $\{-2, -1, 0, 1\}$ i również otrzymać iloczyn 0, ale wyższą sumę. Oznaczałoby to, że już w pierwszym kroku dokonaliśmy błędnego wyboru – ale jak mogliśmy to przewidzieć?

Do tej sytuacji dochodzi tylko wówczas, gdy optymalnym iloczynem jest 0 i dobrym rozwiązaniem jest nie

przewidywać tego wcale – zamiast tego, jeżeli nasz domniemany wynik końcowy zawiera 0, powinniśmy odrzucić go i – podobnie jak w poprzednim przypadku szczególnym – wypisać po prostu k największych liczb ze zbioru. Wśród tych liczb na pewno znajdzie się 0 (inaczej wszystkie z nich musiałyby większe od zera, więc ich iloczyn byłby dodatni, a wówczas nasz algorytm wybrałby je od początku), więc iloczyn pozostanie bez zmian, a suma może wzrosnąć.

Pod względem implementacji, w zadaniu najlepiej jest operować na uporządkowanej zaraz po wczytaniu tablicy. Pozwala to szybko w dowolnym momencie znaleźć jej największe i najmniejsze elementy – są one odpowiednio na jej końcu i początku. Należy przy tym pamiętać, że faktyczne usuwanie elementów z początku dużej tablicy jest bardzo czasochłonne, więc lepiej zamiast tego przechowywać indeksy pierwszego i ostatniego elementu zbioru i wraz z „usuwaniem" elementów tylko je odpowiednio zwiększać lub zmniejszać. Alternatywnie, można wykorzystać dostępny w obu językach programowania typ `deque`, podobny do tablicy, lecz pozwalający na szybkie usuwanie elementów z dowolnego końca.

W przypadku wyboru języka C++, istotne jest również zachowanie ostrożności podczas obliczania iloczynów – te mogą przekraczać zakres typu 32-bitowego `int`, więc należy wykorzystać typ 64-bitowy.

35.2.1 Python

Źródło: ./zadania/04_Sortowanie/Szalony_matematyk_Gnomy/solution.py.

```python
def wypisz(tablica):
    print(' '.join([str(m) for m in tablica]))

def rozwiaz():
    n, k = [int(m) for m in input().split()]
    zbior = sorted([int(m) for m in input().split()])
    wynik = []
    poczatek = 0
    koniec = n - 1
    niedobor = k
    if k % 2 == 1:
        if zbior[koniec] <= 0:
            wypisz(zbior[-k:n])
            return
        wynik.append(zbior[koniec])
        koniec -= 1
        niedobor -= 1
    while niedobor > 0:
        if zbior[poczatek] * zbior[poczatek + 1] > zbior[koniec] * zbior[koniec - 1]:
            wynik.append(zbior[poczatek])
            wynik.append(zbior[poczatek + 1])
            poczatek += 2
        else:
            wynik.append(zbior[koniec])
            wynik.append(zbior[koniec - 1])
            koniec -= 2
        niedobor -= 2
    if 0 in wynik:
        wypisz(zbior[-k:n])
        return
    wynik.sort()
    wypisz(wynik)

rozwiaz()
```

35.2.2 C++

Źródło: ./zadania/04_Sortowanie/Szalony_matematyk_Gnomy/solution.cpp.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

void wypisz_ostatnie_k_elementow(const std::vector<long long>& v, int k) {
  for (auto i = v.size() - k; i < v.size(); i++) {
```

```
    std::cout << v[i];
    if (i + 1 != v.size())
      std::cout << ' ';
  }
  std::cout << '\n';
}

int main() {
  int n, k;
  std::cin >> n >> k;
  std::vector<long long> zbior(n);
  for (auto&& a : zbior) {
    std::cin >> a;
  }
  std::sort(zbior.begin(), zbior.end());
  std::vector<long long> wynik;
  int poczatek = 0;
  int koniec = n - 1;
  int niedobor = k;
  if (k % 2 == 1) {
    if (zbior[koniec] <= 0) {
      wypisz_ostatnie_k_elementow(zbior, k);
      return 0;
    }
    wynik.push_back(zbior[koniec]);
    koniec--;
    niedobor--;
  }
  while (niedobor > 0) {
    if (zbior[poczatek] * zbior[poczatek + 1] > zbior[koniec] * zbior[koniec - 1]) {
      wynik.push_back(zbior[poczatek]);
      wynik.push_back(zbior[poczatek + 1]);
      poczatek += 2;
    } else {
      wynik.push_back(zbior[koniec]);
      wynik.push_back(zbior[koniec - 1]);
      koniec -= 2;
    }
    niedobor -= 2;
  }
  if (std::count(wynik.begin(), wynik.end(), 0)) {
    wypisz_ostatnie_k_elementow(zbior, k);
    return 0;
  }
  std::sort(wynik.begin(), wynik.end());
  wypisz_ostatnie_k_elementow(wynik, k);
  return 0;
}
```

36 (IV) – Szalony matematyk – Skrzaty

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Skrzaty (III edycja – liga algorytmiczna)
Język programowania: Scratch

36.1 Treść zadania

Twój ulubiony nauczyciel matematyki niefortunnie złamał nogę, dlatego szkoła musiała zatrudnić tymczasowo kogoś nowego na zastępstwo. Lekcje z nowym nauczycielem okazują się wprawdzie ciekawe, ale zadaje on zdecydowanie zbyt skomplikowane prace domowe. Czasami wręcz wydaje się, że powinien je rozwiązywać program komputerowy, a nie uczniowie.

Ostatnia lekcja była na temat mnożenia liczb całkowitych. Nauczyciel zadał jako pracę domową następujące ćwiczenie:

„Dany jest zbiór zawierający N liczb całkowitych. Wybierz z nich k liczb tak, żeby ich iloczyn był jak największy. Jeżeli da się to zrobić na kilka sposobów, wybierz ten o największej sumie.”

Napisz program, który rozwiąże to zadanie za Ciebie.

Zadanie

W nowym, pustym projekcie Scratch, utwórz dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania.

Celem zadania jest napisanie programu, który po naciśnięciu zielonej flagi, wczyta z listy wejściowej `DANE` liczby zadane przez nauczyciela, a następnie wybierze i wpisze do listy wyjściowej `WYNIKI` liczby spełniające postawione warunki.

Opis wejścia

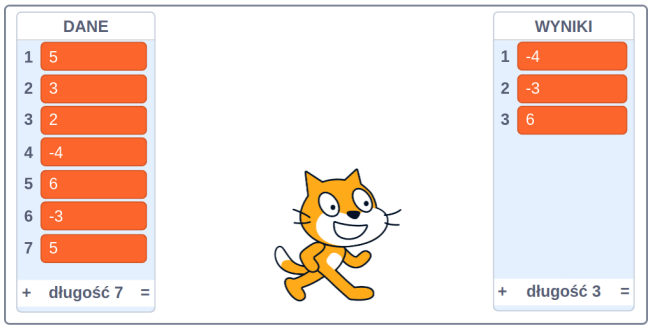
Pierwsze dwa elementy listy `DANE` to liczby naturalne N oraz k (gdzie $2 \leq k < N \leq 1\,000$), opisane w treści zadania. Kolejne N elementów, to liczby całkowite $a_1, a_2, ..., a_N$ – są to elementy zadanego zbioru w losowej kolejności. Każdy element a_i mieści się w przedziale $[-10\,000\,000, 10\,000\,000]$ i liczby nie powtarzają się.

Opis wyjścia

Do listy `WYNIKI` należy wpisać k liczb całkowitych wybranych z zadanego zbioru zgodnie z treścią zadania. Liczby te muszą być uporządkowane rosnąco.

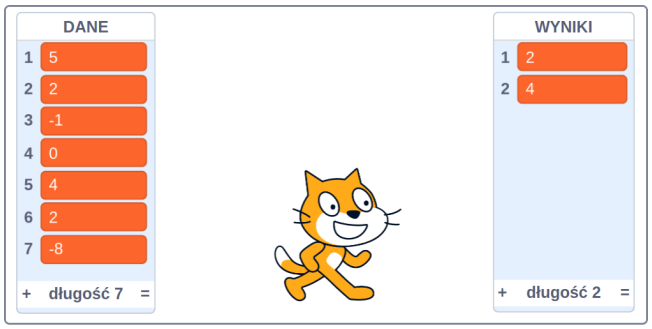
Example

Poniższy rysunek przedstawia przykładowe dane wejściowe i poprawne wyniki:



Rysunek 36.1: Przykład 1

Inny przypadek przedstawiono poniżej:



Rysunek 36.2: Przykład 2

Wyjaśnienie

W pierwszym przypadku, ze zbioru zawierającego 5 wartości należy wybrać 3 liczby, przy czym sam zbiór złożony jest z elementów $\{2, -4, 6, -3, 5\}$. Największym możliwym iloczynem trzech liczb jest w tym wypadku $72 = (-4) \cdot (-3) \cdot 6$. Zgodnie z opisem wyjścia, liczby należy wypisać w porządku rosnącym.

W drugim przypadku największym iloczynem jest 8, ale wartość tę można osiągnąć na dwa sposoby: jako $2 \cdot 4$ lub $(-8) \cdot (-1)$. Według treści zadania, kryterium rozstrzygającym remisy ma być suma liczb, a $2+4 > (-8)+(-1)$, dlatego wybór powinien paść na liczby $2, 4$.

36.2 Rozwiązanie

Rozwiązanie tego zadania w wersji dla Skrzatów jest zgodne z przedstawionym wyżej omówieniem dla Gnomów. Podobnie jak w rozwiązaniu zadania „Bridge Club" wykorzystujemy tu algorytm sortowania bąbelkowego, który – z uwagi na ograniczenia Scratcha (brak możliwości przekazania listy do funkcji) – zapisaliśmy tu osobno do sortowania danych wejściowych oraz wyjściowych.

Rozwiązanie to znajdziesz w pliku:

./zadania/04_Sortowanie/Szalony_matematyk_Skrzaty/solution.sb3.

37 (IV) – Porównywanie liczb – Gremliny

Autor: Andrzej Jastrzębski, Politechnika Gdańska

Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap lokalny)

Język programowania: C++/Python

37.1 Treść zadania

W matematyce liczby naturalne są definiowane przy pomocy aksjomatyki Peana lub przy pomocy aksjomatyki von Neumanna. Obie definicje wprowadzają **klasyczne porównywanie** liczb. Jest to porównanie, które doskonale znamy: $1 < 2 < 3 < 4 < 5 < 6 < \ldots$

Klasyczny porządek liczb nie jest jednak jedynym możliwym. Na potrzeby niniejszego zadania wykorzystamy nieco inny porządek, zdefiniowany w XX wieku przez ukraińskiego matematyka Aleksandra Szarkowskiego.

Porządek Szarkowskiego (który będziemy oznaczali przez $\lhd$) liczb naturalnych jest związany z teorią chaosu. Niech $n = 2^{k_n} \cdot p_n, m = 2^{k_m} \cdot p_m$, gdzie p_n, p_m są liczbami nieparzystymi. Porządek Szarkowskiego definiujemy następująco:

- jeśli $p_n \neq 1$ i $p_m \neq 1$, to $n \lhd m$, gdy $k_n < k_m$ lub $k_n = k_m$ i $p_n < p_m$
- jeśli $p_n \neq 1$ i $p_m = 1$ to $n \lhd m$
- jeśli $p_n = p_m = 1$ to $n \lhd m$, gdy $k_n > k_m$

W tym porządku kolejność liczb jest następująca:

$$\begin{array}{ccccccccc}
3 & \lhd & 5 & \lhd & 7 & \lhd & 9 & \lhd & \ldots \\
3 \cdot 2 & \lhd & 5 \cdot 2 & \lhd & 7 \cdot 2 & \lhd & 9 \cdot 2 & \lhd & \ldots \\
3 \cdot 2^2 & \lhd & 5 \cdot 2^2 & \lhd & 7 \cdot 2^2 & \lhd & 9 \cdot 2^2 & \lhd & \ldots \\
3 \cdot 2^3 & \lhd & 5 \cdot 2^3 & \lhd & 7 \cdot 2^3 & \lhd & 9 \cdot 2^3 & \lhd & \ldots \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\ldots & \lhd & 2^3 & \lhd & 2^2 & \lhd & 2 & \lhd & 1
\end{array}$$

Zadanie

Napisz program, który sortuje liczby w sposób klasyczny i według porządku Szarkowskiego.

Opis wejścia

W pierwszym wierszu znajduje się liczba n – jest to liczba liczb do posortowania (nie więcej niż $50\,000$).

W kolejnej linii znajduje się n różnych liczb naturalnych przedzielonych pojedynczą spacją. Wartości tych liczb nie przekraczają $1\,000\,000\,000\,000\,000\,000$. Na potrzeby zadania zakładamy, że 0 nie jest liczbą naturalną.

Opis wyjścia

W dwóch liniach należy wpisać posortowany ciąg liczb wejściowych. W pierwszej linii w porządku klasycznym, w drugiej w porządku Szarkowskiego.

Przykład

Dla danych wejściowych:

```
20
18 5 17 12 6 10 3 1 11 16 9 14 2 15 13 4 19 20 7 8
```

prawidłową odpowiedzią jest:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
3 5 7 9 11 13 15 17 19 6 10 14 18 12 20 16 8 4 2 1
```

Wyjaśnienie przykładu

Danymi wejściowymi są kolejne liczby naturalne od 1 do 20.

Liczby te możemy zapisać następująco: $1 = 2^0 \cdot 1$, $2 = 2^1 \cdot 1$, $3 = 2^0 \cdot 3$, $4 = 2^2 \cdot 1$, $5 = 2^0 \cdot 5$, $6 = 2^1 \cdot 3$, $7 = 2^0 \cdot 7$, $8 = 2^3 \cdot 1$, $9 = 2^0 \cdot 9$, $10 = 2^1 \cdot 5$, $11 = 2^0 \cdot 11$, $12 = 2^2 \cdot 3$, $13 = 2^0 \cdot 13$, $14 = 2^1 \cdot 7$, $15 = 2^0 \cdot 5$, $16 = 2^4 \cdot 1$, $17 = 2^0 \cdot 17$, $18 = 2^1 \cdot 9$, $19 = 2^0 \cdot 19$, $20 = 2^2 \cdot 5$.

Zgodnie z porządkiem Szarkowskiego będziemy mieli zatem:

$$\begin{array}{ccccccccccccccccccc}
 & 2^0 \cdot 3 & \lhd & 2^0 \cdot 5 & \lhd & 2^0 \cdot 7 & \lhd & 2^0 \cdot 9 & \lhd & 2^0 \cdot 11 & \lhd \\
\lhd & 2^0 \cdot 13 & \lhd & 2^0 \cdot 15 & \lhd & 2^0 \cdot 17 & \lhd & 2^0 \cdot 19 & \lhd & 2^1 \cdot 3 & \lhd \\
\lhd & 2^1 \cdot 5 & \lhd & 2^1 \cdot 7 & \lhd & 2^1 \cdot 9 & \lhd & 2^2 \cdot 3 & \lhd & 2^2 \cdot 5 & \lhd \\
\lhd & 2^4 & \lhd & 2^3 & \lhd & 2^2 & \lhd & 2^1 & \lhd & 2^0 \, , &
\end{array}$$

skąd ostatecznie:

$$\begin{array}{ccccccccccc}
 & 3 & \lhd & 5 & \lhd & 7 & \lhd & 9 & \lhd & 11 & \lhd \\
\lhd & 13 & \lhd & 15 & \lhd & 17 & \lhd & 19 & \lhd & 6 & \lhd \\
\lhd & 10 & \lhd & 14 & \lhd & 18 & \lhd & 12 & \lhd & 20 & \lhd \\
\lhd & 16 & \lhd & 8 & \lhd & 4 & \lhd & 2 & \lhd & 1 \, . &
\end{array}$$

37.2 Rozwiązanie

Zadanie jest związane z sortowaniem tablicy liczb. Użycie sortowania wbudowanego w język jest tu dobrym podejściem do rozwiązania. Jedynym problemem jest niestandardowy sposób porównywania liczb.

Są dwa sposoby na zmierzenie się z wyznaczaniem porządku Szarkowskiego [16]. Pierwszy sposób to obliczenie dla wszystkich wartości z tablicy (dane wejściowe) współczynników k_n, p_n takich, że $n = 2^{k_n} \cdot p_n$, a następnie stworzenie nowej tablicy z parami (k_n, p_n) – lub trójką (k_n, p_n, n) – oraz porównywanie ich.

Drugi sposób, to wyznaczanie par (k_n, p_n) za każdym razem, gdy porównujemy dwie liczby.

Kolejnym problemem jest skomplikowany warunek wykorzystywany przy porównaniu.

Rozwiązania wzorcowe w językach C++ i Python podchodzą do tego zagadnienia w różny sposób. W rozwiązaniu wzorcowym w C++ wykorzystano warunek z zadania, natomiast w rozwiązaniu wzorcowym w Pythonie użyto spostrzeżenia, że porównywanie dzieli liczby naturalne na:

1. liczby postaci $n = 2^{k_n} \cdot p_n$, gdzie $p_n \neq 1$,
2. liczby postaci $n = 2^{k_n}$.

Dla liczb w pierwszej postaci porównywanie jest tożsame z porównywaniem leksykograficznym ciągów (k_n, p_n) tj. najpierw klasycznie porównujemy pierwszą współrzędną, a jeśli wartości pierwszej współrzędnej są równe, to porównujemy drugą współrzędną.

Dla liczb w drugiej postaci porównywanie jest tożsame z odwrotnym porównywaniem leksykograficznym ciągów (k_n, p_n).

Można zmienić reprezentację ciągową liczb w drugiej postaci tak, aby zadanie sprowadziło się do porównywania leksykograficznego ciągów. Wystarczy, że liczby w drugiej postaci zamieni się na ciąg $(M - k_n, 1)$ gdzie M jest odpowiednio dużą liczbą. Ograniczenia w zadaniu powodują, że $k_n < 60$, a zatem wystarczy przyjąć $M = 120$. W rozwiązaniu wzorcowym została użyta liczba 200.

37.2.1 Python

Źródło: ./zadania/04_Sortowanie/Porownywanie_liczb/solution.py.

```
def sort_szarkowski(x):
  i=0
  while x%2==0:
    x//=2
    i+=1
  if x==1: #wytrych, gdy mamy do czynienia z potęgami 2
    return (200-i,1)
  return (i, x)


def test():
  linia = input().split()
  n = int(linia[0])
  linia = input()
  dane = [int(i) for i in linia.split()]

  dane.sort()
  print(' '.join(str(i) for i in dane)+' ')

  dane.sort(key=sort_szarkowski)
  print(' '.join(str(i) for i in dane)+' ')

test()
```

37.2.2 C++

Źródło: ./zadania/04_Sortowanie/Porownywanie_liczb/solution.cpp.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

void test(vector<long long> &dane) {
  //sortowanie klasyczne
  sort(dane.begin(), dane.end());
  for(auto x : dane) {
    cout << x << " ";
  }
  cout << endl;
  //sortowanie Szarkowskiego
  sort(dane.begin(), dane.end(),
    [](auto v1, auto v2) {
      int k1=0, k2=0;
      while(v1%2==0) {
```

```
        v1/=2;
        k1+=1;
      }
      while(v2%2==0) {
        v2/=2;
        k2+=1;
      }
      //porównywanie
      if(v1==1 && v2==1)
        return k1>k2;
      if(v1==1 || v2==1)
        return v2==1;
      //v1!=1 && v2!=1
      if(k1!=k2) return k1<k2;
      //k1==k2
      return v1<v2;
    }
  );
  for(auto x : dane) {
    cout << x << " ";
  }
  cout << endl;
}

int main(int argc, char* argv[]) {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int n;
  vector<long long> dane;
  cin >> n;
  dane.resize(n);
  for(int i=0; i<n; i++) {
    cin >> dane[i];
  }
  test(dane);
}
```

38 (IV) – Vururuk – Gremliny

Autor: Łukasz Skonieczny, Politechnika Warszawska
Kategoria: Gremliny (II edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

38.1 Treść zadania

W wiosce plemienia Vururuk co 10 lat odbywają się masowe śluby małżeńskie. Wszystkie pełnoletnie panny i pełnoletni kawalerowie stawiają się przed radą starszych, która dokonuje swatania w sposób zgodny z wielowiekową tradycją plemienia. Po pierwsze, mąż musi mieć większy wzrost i większy obwód głowy niż żona. Po drugie, każda panna i każdy kawaler musi znaleźć małżonka. W przypadku niemożności spełnienia tych warunków, proces swatania kończy się niepowodzeniem i jest odraczany na następne 10 lat.

Najbliższe śluby odbędą się już w tym tygodniu! Pomóż radzie starszych przeprowadzić swatanie.

Zadanie

Określ, czy w danym zbiorze kawalerów i panien możliwe jest swatanie spełniające oba warunki.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita: $1 \leq n \leq 500\,000$ oznaczająca liczbę kawalerów oraz liczbę panien (szczęśliwie zbiory kawalerów i panien są równoliczne). W kolejnych n liniach znajdują się pary liczb opisujące kawalerów, a w kolejnych n liniach podano pary liczb opisujące panny. Pierwsza liczba z pary oznacza wzrost, a druga – obwód głowy. Są to liczby z zakresu $[1, 10^7]$ – oczywiście całkowite, bo starsi normalizują długości przed swataniem i dobierają jednostkę w zależności od potrzeb.

Opis wyjścia

Na standardowe wyjście należy wyprowadzić ciąg znaków `NIE`, jeśli nie jest możliwe przeprowadzenie swatania zakończonego sukcesem. W przeciwnym przypadku należy podać liczbę oznaczającą łączną sumę różnic wzrostów i obwodów głów wszystkich zeswatanych par.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
2
180 100
170 90
150 70
160 80
```

prawidłową odpowiedzią jest:

```
80
```

Dla innego, podanego poniżej wejścia:

```
3
180 100
170 80
190 80
160 90
150 70
170 80
```

prawidłową odpowiedzią jest:

```
NIE
```

Wyjaśnienie przykładu

W pierwszym przypadku do konkursu staje 2 kawalerów i 2 panny. Ponieważ obaj kawalerowie są wyżsi od panien, a także mają od nich większe obwody głów, wszystkie możliwe zeswatania są dozwolone: (1) pierwszy kawaler z pierwszą panną, drugi kawaler z drugą panną, oraz (2) pierwszy z drugą i drugi z pierwszą. Bez względu na wybór zeswatania, łączna suma różnic wzrostów i obwodów głów będzie taka sama. Sprawdźmy:

1. $(180 - 150) + (100 - 70) + (170 - 160) + (90 - 80) = 80$
2. $(180 - 160) + (100 - 80) + (170 - 150) + (90 - 70) = 80$

W drugim przypadku do konkursu staje 3 kawalerów i 3 panny. Można sprawdzić, że żadne z 6 możliwych zeswatań nie spełnia wymaganych warunków.

38.2 Rozwiązanie

Treść zadania sugeruje, że mamy tu do czynienia z problemem kojarzenia małżeństw[1]. W istocie, możemy uzyskać rozwiązanie modelując problem w ten sposób. Tworzymy graf dwudzielny (p. dział VIII), zawierający n wierzchołków reprezentujących kawalerów, n wierzchołków reprezentujących panny i generujemy odpowiednią liczbę krawędzi łączących wierzchołki obu grup zgodnie z warunkami zadania, tzn. krawędź istnieje tylko wtedy, gdy kawaler jest wyższy i ma większy obwód głowy niż panna. Taki graf można wygenerować w czasie $O(n^2)$. Następnie szukamy w tym grafie pełnego skojarzenia (p. dział VIII), np. za pomocą algorytmu Edmondsa-Karpa lub Hopcrofta-Karpa [4][17][23].

Na podstawie rozmiarów danych wejściowych (graf o milionie wierzchołków) dochodzimy jednak do wniosku, że złożoność takiego rozwiązania może okazać się zbyt duża (w zależności od liczby krawędzi grafu – kwadratowa lub większa). W rozwiązaniu wzorcowym zaproponowano metodę z „omiataniem" [4]. Dla łatwiejszej wizualizacji reprezentujmy kawalerów i panny jako punkty na płaszczyzny, których współrzędna x oznacza wzrost, a y – obwód głowy. Przeglądając punkty w kolejności malejących „iksów" możemy dokonywać skojarzeń „na bieżąco". Możemy tak robić, gdyż (jak przekonuje przykład) wszystkie skojarzenia są tak samo dobre – mają tę samą łączną sumę różnic wzrostów i obwodów głów zeswatanych par.

W czasie omiatania płaszczyzny, będziemy wrzucać napotkanych kawalerów do zbioru kandydatów. Jeżeli napotkamy pannę, to musimy znaleźć jej męża w tym zbiorze. Jeśli go tam nie ma, to nie ma sensu szukać dalej, gdyż kolejni kandydaci będą niżsi, więc żaden nie będzie mógł zostać jej mężem. Wszyscy kawalerowie w zbiorze kandydatów mają wyższy wzrost (to wynika z kolejności przeglądania punktów), więc wystarczy znaleźć w nim osoby o większym obwodzie głowy. Spośród nich wybieramy tego o najmniejszym obwodzie głowy, aby kolejne panny miały potencjalnie większy wybór. Struktura, do której wrzucamy kandydatów, powinna więc umożliwiać szybkie wykonywanie następujących operacji:

- szybkie znajdowanie elementu z najmniejszą wartością y większą od danej (tzw. operacja *upper-bound*);
- szybkie wstawianie i usuwanie.

Znakomicie nadają się do tego zrównoważone drzewa BST (z kluczem opartym na y), które realizują wszystkie wymagane operacje w czasie $O(\log n)$. W bibliotece standardowej C++ taką funkcjonalność zapewnia `set`.

Niestety, standardowe zbiory w Pythonie są zrealizowane w postaci tablic mieszających (odpowiednik struktury `unordered_set` z C++), więc nie możemy użyć ich do realizacji zbioru kandydujących kawalerów. Python nie posiada też żadnych wbudowanych implementacji drzew BST. Można pokusić o własną implementację, ale – dla uproszczenia – w rozwiązaniu wzorcowym dla Pythona użyto zwykłej posortowanej listy. Wyszukiwanie najmniejszego y większego od podanego można tu zrealizować w czasie logarytmicznym za pomocą funkcji `bisect_right`. Dodawanie nowych elementów z utrzymaniem posortowania można wykonać za pomocą metody `insort`. Niestety ma ona złożoność liniową, gdyż wstawianie w środek listy (która – jak pamiętamy z wprowadzenia do poprzedniego działu – nie jest implementowana w Pythonie jako prawdziwa *lista wiązana*) wymaga przesunięcia elementów stojących za miejscem wstawienia.

Rozwiązanie w C++ ma złożoność $O(n \log n)$ (sortowania oraz pesymistycznie n-krotne $\log n$ przy każdym zatrzymaniu „miotły"), a rozwiązanie w Pythonie jest pesymistycznie kwadratowe. W praktyce będzie działać szybciej, gdyż rzadko zdarzy się, że zbiór kandydatów będzie duży (bliski n).

[1]Problem ten zostanie omówiony we wprowadzeniu do działu VIII o grafach.

38.2.1 Python

Źródło: ./zadania/04_Sortowanie/Vururuk/solution.py.

```
from operator import attrgetter
from bisect import insort
from bisect import bisect_right

class Person:
    height = 0
    head = 0
    female = False
    id = 0

    def __lt__(self,other):
        if self.head < other.head:
            return True
        if self.head > other.head:
            return False
        if self.height > other.height:
            return True
        if self.height < other.height:
            return False
        if self.id < other.id:
            return True
        return False

def main():
    n = int(input())
    people = []
    men = []
    for i in range(0,2*n):
        p = Person()
        p.height, p.head =  [int(x) for x in input().split()]
        p.id = i
        p.female = (i>=n)
        people.append(p)
    people.sort(key=attrgetter("head"), reverse=True)
    people.sort(key=attrgetter("height"), reverse=True)
    result = 0
    for p in people:
        if p.female:
            matching_man = bisect_right(men, p)
            if matching_man!=len(men):
                result += (men[matching_man].height - p.height) + (men[matching_man].head - p.head)
                del(men[matching_man])  # O(n) sadly
            else:
                print("NIE")
                return
        else:
            insort(men, p)  # O(n) sadly
    print(result)
```

```
if __name__ == '__main__':
    main()
```

38.2.2 C++

Źródło: ./zadania/04_Sortowanie/Vururuk/solution.cpp.

```
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>

using namespace std;

struct Person {
    unsigned height;
    unsigned head;
    unsigned id;
    bool female = true;
};

bool heightCompRev(const Person& a, const Person& b) {
    if (a.height > b.height) return true;
    if (a.height < b.height) return false;
    if (a.head > b.head) return true;
    if (a.head < b.head) return false;
    if (a.id < b.id) return true;
    return false;
}

bool headComp(const Person& a, const Person& b) {
    if (a.head < b.head) return true;
    if (a.head > b.head) return false;
    if (a.height > b.height) return true;
    if (a.height < b.height) return false;
    if (a.id < b.id) return true;
    return false;
}

int main() {
    int n;
    cin>>n;
    vector<Person> people;
    people.resize(2*n);
    set<Person,bool(*)(const Person&,const Person&)> men(headComp);

    for (int i=0; i<2*n; i++) {
        cin>>people[i].height;
        cin>>people[i].head;
```

```
        people[i].id = i;
        people[i].female = (i>=n);
    }

    sort(people.begin(),people.end(),heightCompRev);
    unsigned long long result = 0;
    for (const Person &p : people) {
        if (p.female) {
            auto matching_man = men.upper_bound(p);
            if (matching_man!=men.end()) {
                result += (matching_man->height - p.height) + (matching_man->head - p.head);
                men.erase(matching_man);
            } else {
                cout<<"NIE"<<endl;
                return 0;
            }
        } else {
            men.insert(p);
        }
    }
    cout <<result<<endl;
}
```

Centrum Mistrzostwa Informatycznego

Dział V

Dwuwymiarowe struktury danych i elementy geometrii obliczeniowej

39 (V) – Wprowadzenie

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka

W wielu zadaniach, które omawialiśmy w poprzednich działach pojawiały się listy, tablice i inne *kontenery* służące do przechowywania i przetwarzania ciągów danych. Naturalnym rozszerzeniem tych struktur danych są struktury dwuwymiarowe, pozwalające np. na odwoływanie się do lokalizacji obiektów na płaszczyźnie za pomocą współrzędnych x-y, czy modelowanie bardziej złożonych relacji między nimi. Struktury takie są bardzo przydatne w wielu problemach algorytmicznych, m.in. z zakresu programowania dynamicznego, czy grafów – o czym będziemy mówić w dalszych działach – a także w problemach związanych z szeroko pojętą geometrią obliczeniową, które pojawią się w tym dziale. Na początku jednak omówimy pokrótce sposoby konstruowania i użycia struktur 2D w językach C++ i Python (Scratch nie udostępnia takich struktur, jeśli pominąć *planszę*, która jest wprawdzie obiektem dwuwymiarowym, ale nie służy zasadniczo do przechowywania danych).

Prostym sposobem utworzenia dwuwymiarowej struktury danych jest zagnieżdżenie deklaracji struktur jednowymiarowych, czyli utworzenie np. „listy list", czy „wektora wektorów". Rozważmy poniższy przykład w języku Python:

```
wiersz1 = [1, 11, 111]
wiersz2 = [2, 22, 222]
tab_2D = [wiersz1, wiersz2]
```

Zmienna `tab_2D` reprezentuje listę wierszy, z których każdy jest również listą (wartości całkowitych). Dokładnie ten sam efekt możemy uzyskać zapisując bezpośrednio:

```
tab_2D = [[1, 11, 111], [2, 22, 222]]
```

Zawartość zmiennej `tab_2D` odpowiada następującej tablicy dwuwymiarowej (którą w kontekście matematycznym moglibyśmy utożsamić z *macierzą* o dwóch wierszach i trzech kolumnach):

1	11	111
2	22	222

Do poszczególnych elementów odwołujemy się za pomocą podwójnego indeksu, przykładowo:

```
print(tab_2D[0][2]) # wynik: 111
```

Możemy to zinterpretować jako „element o indeksie 2 z zerowego wiersza".
Warto też wiedzieć, że tablicę dowolnych rozmiarów $N \times M$, wypełnioną np. zerami możemy łatwo skonstruować w Pythonie za pomocą składni:

```
tab_2D = [[0] * M for x in range(N)]
```

W zapisie tym wyrażenie `[0]*M` odpowiada `M`-elementowej liście samych zer.
Analogiczną tablicę o wymiarach 2×3 możemy utworzyć w języku C++, na przykład z wykorzystaniem struktury `vector`:

```
vector<vector<int>> tab_2D = vector<vector<int>>(2, vector<int>(3));
tab_2D[0][2] = 111;
```

Warto pamiętać, że struktury dwuwymiarowe możemy także konstruować w oparciu o podstawowy, wbudowany typ tablicowy języka C++:

```
int tab[2][3]
```

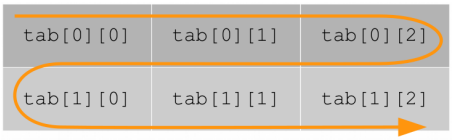
Podobnie jak we wcześniejszych przykładach, pierwszy indeks określa liczbę wierszy, a drugi – liczbę kolumn i oczywiście tej samej składni używamy do odwoływania się do poszczególnych elementów tablicy:

tab[0][0]	tab[0][1]	tab[0][2]
tab[1][0]	tab[1][1]	tab[1][2]

Niezależnie od języka i wybranej struktury danych, do przetwarzania wszystkich elementów tablicy dwuwymiarowej stosujemy typowo dwie pętle zagnieżdżone jedna w drugiej. Przykładowo, kod wypełniający tabliczkę mnożenia właściwymi wartościami może mieć postać:

```
int tabliczkaMnozenia[10][10];
for (int w = 0; w < 10; w++) {       // w: numer wiersza
    for (int k = 0; k < 10; k++) {   // k: numer kolumny
        tabliczkaMnozenia[w][k] = (w+1) * (k+1);
    }
}
```

W języku C++ dane w tablicy umieszczane są w pamięci wierszami (ang. *row-major*) w jednym ciągłym bloku...



...czyli w gruncie rzeczy po prostu tak:

tab[0][0]
tab[0][1]
tab[0][2]
tab[1][0]
tab[1][1]
tab[1][2]

Moglibyśmy więc zasymulować tablicę dwuwymiarową wykorzystując zwykłą, jednowymiarową strukturę danych. Oczywiście trzeba przy tym pamiętać o alokacji pamięci na odpowiednią liczbę elementów i o odpowiednim przeliczaniu indeksów wierszy i kolumn na adresy komórek:

```
int tab[6]; // 2 wiersze X 3 kolumny = 6 elementów
for (int w = 0; w < 2; w++) {
    for (int k = 0; k < 3; k++) {
        tab[w * 3 + k] = (w + 1) * (k + 1);
    }
}
```

To podejście jest szczególnie użyteczne w sytuacji, w której nie dysponujemy dwuwymiarowymi strukturami danych – np. w Scratchu (p. zadanie: „Baza księżycowa” dla Skrzatów, dalej w tym dziale).

Kontynuując temat tablic dwuwymiarowych w C++, zauważmy iż deklarację:

```
int tab [2][3];
```

można interpretować jako „tablicę tablic”:

```
int (tab [2])[3];
```

Do poszczególnych wierszy można zatem również odwoływać się indywidualnie za pomocą odpowiedniego wskaźnika (adresu):

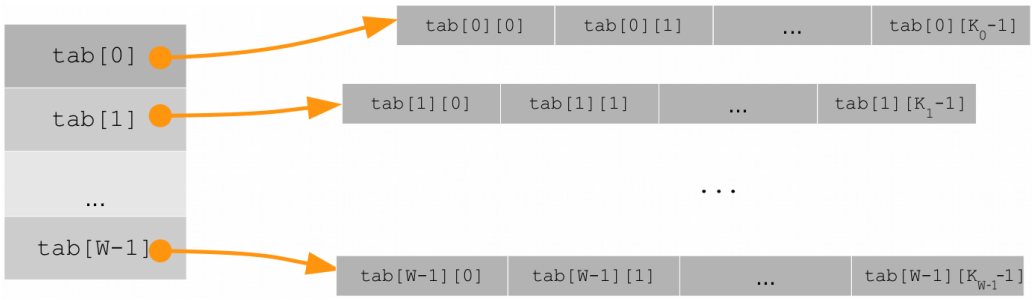
```
int *ptr = tab[1] // ptr wskazuje na 2 wiersz
ptr[2] = 5; //odwołanie do trzeciego elementu
```

tab[0][0]	tab[0][1]	tab[0][2]
tab[1][0]	tab[1][1]	tab[1][2]

Alternatywnym sposobem konstrukcji tablicy dwuwymiarowej jest zastosowanie tablicy wskaźników, z których każdy odnosi się do jednego wiersza. Każdemu wskaźnikowi przypisujemy adres jednowymiarowej tablicy (wiersza), utworzonej operatorem `new`. Nasz przykład z tabliczką mnożenia mógłby zatem wyglądać tak:

```
int ** tabliczkaMnozenia = new int*[10];
for (int w = 0; w < 10; w++) {
    tabliczkaMnozenia[w] = new int[10];
    for (int k = 0; k < 10; k++) {
        tabliczkaMnozenia[w][k] = (w+1) * (k+1);
    }
}
```

Zauważmy, że w tym rodzaju tablicy (ang. *jagged array*) każdy wiersz może mieć inną długość. Można w ten sposób efektywnie tworzyć tablice o kształcie innym niż prostokątny (np. trójkątne).



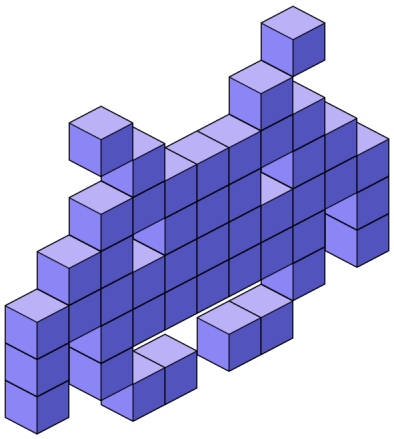
Na koniec warto zauważyć, że w przypadku, gdy przechowywane elementy mogą być reprezentowane w postaci pojedynczego znaku (cyfry, litery, etc.), wówczas rolę struktury dwuwymiarowej może pełnić po prostu lista napisów (czyli lista obiektów typu `string`). Przykład takiego podejścia zobaczymy m.in. w zadaniu „Koneser gier retro” w tym dziale.

40 (V) – Koneser gier retro – Gnomy

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap regionalny)
Język programowania: C++/Python

40.1 Treść zadania

Bajtek chce zrobić prezent swojemu tacie. Tata Bajtka jest amatorem gier komputerowych z ubiegłej epoki – z czasów, gdy dwuwymiarowa rozgrywka w zupełności każdemu wystarczała, a w elementach graficznych dało się na pierwszy rzut oka zliczyć piksele. Dlatego właśnie pomysłem Bajtka jest sklejony z drewnianych sześciennych kostek model przedstawiający rozpoznawalną grafikę 2D z jednej z ulubionych gier jego taty.



Rysunek 40.1: Przykładowy prezent

Bajtek ma już plan co do tego, jak skonstruować model. Pozostała jedynie kwestia jego pomalowania. W tym celu musi zamówić właściwą ilość farby. Pomalowanie jednej ściany kostki wymaga 1 porcji farby – ale oczywiście nie trzeba malować ścian, które nie są widoczne, bo zostały sklejone z innymi ścianami. Ustal, ilu dokładnie porcji farby potrzeba, żeby pomalować cały model.
Zauważ, że model niekoniecznie musi być w całości spójny. Bajtek może planować połączyć niektóre elementy mało widocznymi drutami, aby utrzymać go w całości – ale nie musisz się tym przejmować w obliczeniach.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby naturalne a, h (gdzie $2 \leq a, h \leq 1024$) oddzielone spacją, oznaczające odpowiednio szerokość i wysokość modelu. Każdy z kolejnych h wierszy zawiera ciąg a znaków, z których każdy to kropka („.”) lub kratka („#”) – oznaczają one ułożenie kostek. Kratka na danej pozycji oznacza, że znajduje się na niej kostka, zaś kropka symbolizuje pustą przestrzeń. Możesz założyć, że model zawiera przynajmniej jedną kostkę.

Opis wyjścia

Na standardowe wyjście należy wypisać jedną liczbę całkowitą, oznaczającą ilość farby potrzebną do pomalowania prezentu Bajtka.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
3 3
..#
#..
##.
```

prawidłową odpowiedzią jest:

```
20
```

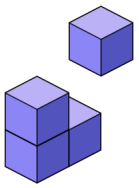
natomiast dla wejścia:

```
11 8
..#.....#..
...#...#...
..#######..
.##.###.##.
###########
#.#######.#
#.#.....#.#
...##.##...
```

prawidłową odpowiedzią jest:

```
166
```

Wyjaśnienie



Rysunek 40.2: Ilustracja przykładu

Pierwszy przykład został zilustrowany powyżej (Rys. 40.2). Składa się z dwóch oddzielnych brył: pojedynczej kostki mającej 6 ścian i bardziej złożonego kształtu, który wymaga 14 porcji farby – łącznie 20.

Drugi przykład odpowiada ilustracji załączonej w treści zadania (Rys. 40.1).

40.2 Rozwiązanie

W zadaniu należało obliczyć ilość farby potrzebną do pomalowania zbioru brył utworzonych z sześcianów jednostkowych sklejonych ścianami – czyli inaczej ich łączne pole powierzchni. Wszystkie sześciany leżały przy tym w tej samej płaszczyźnie.

W tym zadaniu najbardziej intuicyjne rozwiązanie jest zarazem najwydajniejsze – polega ono na prostym zliczeniu ścian, do których nie przylegały inne sześciany.

Przypomnijmy, że wejście w zadaniu było podane w formacie ciągów kropek i kratek, gdzie kratki oznaczały, że na danej pozycji znajduje się sześcian, np.:

```
..#
#..
##.
```

Dobrym pierwszym krokiem w tego typu zadaniach bywa otoczenie wczytanego wejścia „ramką” złożoną z pustej przestrzeni, czyli w tym wypadku kropek, tj.:

```
.....
...#.
.#...
.##..
.....
```

Pozwala nam to uniknąć martwienia się granicami tablic w dalszych krokach. Następnie możemy zagnieżdżoną pętlą `for` przejść po każdym znaku. Zawsze, gdy napotkamy znak „#”, dodajemy 2 do wyniku – ścianę przednią i tylną, które na pewno nie mogą zostać zasłonięte. Dodatkowo dodajemy 1 za każde sąsiadujące pole (powyżej, poniżej, po lewej i po prawej), na którym znajduje się kropka.

Do zliczania można też podejść na kilka równoważnych sposobów – na przykład początkowo dodając 6 do wyniku za każdy sześcian, a potem odejmując 2 za każdą parę sąsiadujących poziomo lub pionowo kratek.

Niezależnie od wyboru podejścia, tak napisany program rozwiązuje zadanie.

40.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Koneser_gier_retro/solution.py.

```
szerokosc, wysokosc = [int(n) for n in input().split()]
dane = [(szerokosc + 2) * '.']
for _ in range(wysokosc):
    dane.append('.' + input() + '.')
dane.append((szerokosc + 2) * '.')
wynik = 0
for i in range(wysokosc + 1):
    for j in range(szerokosc + 1):
        if dane[i][j] == '#':
            wynik += 2
        if dane[i][j] != dane[i][j + 1]:
            wynik += 1
        if dane[i][j] != dane[i + 1][j]:
            wynik += 1
print(wynik)
```

40.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Koneser_gier_retro/solution.cpp.

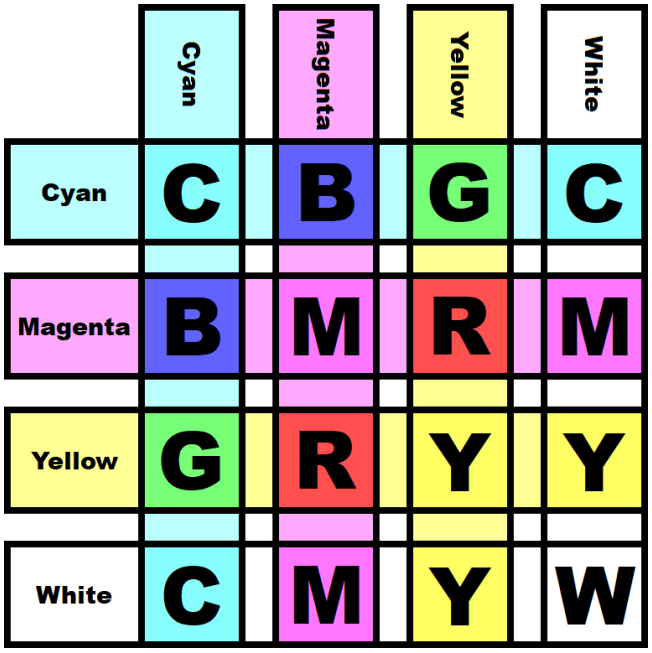
```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

int main() {
  int szerokosc, wysokosc;
  std::cin >> szerokosc >> wysokosc;
  std::vector<std::string> dane;
  dane.emplace_back(szerokosc + 2, '.');
  for (int i = 0; i < wysokosc; i++) {
    std::string wiersz;
    std::cin >> wiersz;
    dane.push_back('.' + wiersz + '.');
  }
  dane.emplace_back(szerokosc + 2, '.');
  int wynik = 0;
  for (int i = 0; i <= wysokosc; i++) {
    for (int j = 0; j <= szerokosc; j++) {
      if (dane[i][j] == '#')
        wynik += 2;
      if (dane[i][j] != dane[i][j + 1])
        wynik++;
      if (dane[i][j] != dane[i + 1][j])
        wynik++;
    }
  }
  std::cout << wynik << '\n';
  return 0;
}
```

41 (V) – Modern Art Analysis – Gnomy *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (II edycja – zawody algorytmiczne – etap regionalny)
Język programowania: C++/Python

41.1 Task description

Your city has erected a new monument in the town square. It is an unusual and abstract piece of art: a square frame containing a large, colorful, translucent grid of small squares. Every small square is one of 7 colors: red, green, blue, cyan, magenta, yellow, or white. After close observation of the monument you noticed that it is not actually built of individual squares, but of thin long strips divided into 2 layers – one vertical and one horizontal. You suspect that every strip is one of 4 colors: cyan, magenta, yellow, or white, and the remaining colors are simply combinations of overlapping strips on the two layers. The image below illustrates all the possible combinations of the colors and the results of mixing them.



Unfortunately the monument's frame covers its borders so you cannot see the individual stripes and directly confirm your suspicions. Instead, you noted down the colors of all the small squares in order to check if your hypothesis is plausible.

Task

Knowing the colors of the small squares, determine whether there exists an arrangement of colored strips that could produce the grid. If yes, find and print that arrangement, otherwise print "IMPOSSIBLE".

Input description

The first line of the standard input contains a single integer n ($2 \leq n \leq 1000$) denoting the side length of the square grid. The following n lines contain text strings of length n each, describing the grid arrangement. Every string consists only of capital letters `R, G, B, C, M, Y, W`, corresponding to appropriate colors as explained above. It is guaranteed that 2 adjacent columns will never be identical, and – similarly – that 2 adjacent rows will never be identical.

Output description

The standard output should contain 2 lines of length n each, consisting only of capital letters `C, M, Y, W`. The first line should denote the colors used by the horizontal strips, and the second one – by the vertical strips. If the provided arrangement is impossible to obtain according to the rules described above, print only 1 line that says `IMPOSSIBLE` instead. If multiple valid solutions exist, print any one of them.

Example

For the following sample input:

```
3
MMM
RYR
BCB
```

the correct output is:

```
MYC
MWM
```

whereas for input:

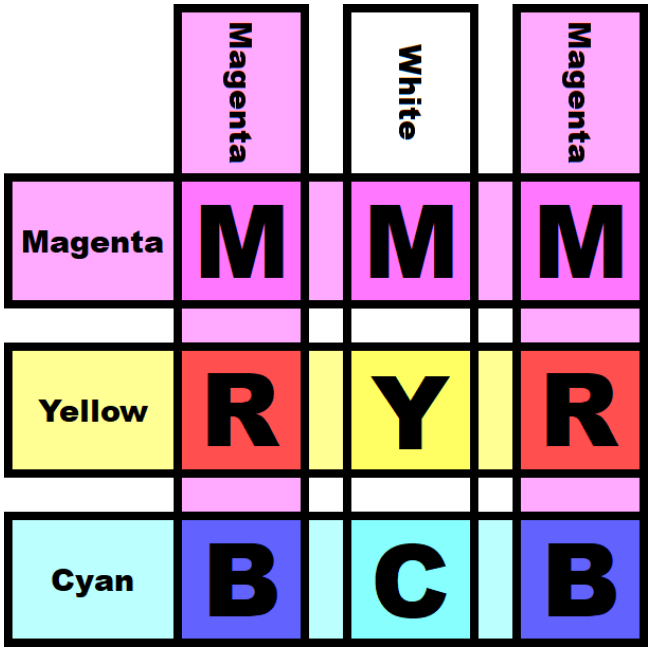
```
3
MMM
RYR
BCC
```

the correct output is:

```
IMPOSSIBLE
```

Explanation

The first test corresponds to the following situation:



The second test is similar but replaces the color in the bottom right corner with cyan. There is actually no color strip that can produce both cyan and red when combined with other colors, so the rightmost column alone makes this arrangement impossible to satisfy.

41.2 Rozwiązanie

Opis zadania:

- Na podstawie kwadratowego diagramu kolorowych kwadratów ustal, czy i w jaki sposób da się go otrzymać jako kombinację podłużnych półprzezroczystych kolorowych płytek ułożonych poziomo i pionowo.
- Żadna para sąsiednich wierszy ani kolumn wejścia nie będzie identyczna, a rozmiar diagramu to co najmniej 2×2.

Rozwiązanie opiera się na spostrzeżeniu, że wystarczy sprawdzić pierwsze dwa kwadraty w każdym wierszu, aby z całą pewnością ustalić kolor, jaki musi mieć odpowiadająca mu płytka (o ile poprawne rozwiązanie w ogóle istnieje). To samo można odpowiednio powiedzieć o kolumnach.

Aby tego dowieść, zauważmy, że:

- żadne dwie sąsiadujące pionowe płytki nie mają tego samego koloru, bo inaczej odpowiadające im kolumny musiałyby być identyczne, a to jest wykluczone przez treść zadania;
- skoro tak, to pierwsze dwa kwadraty wiersza muszą odpowiadać dwóm **różnym** kwadratom powyższego diagramu ze wspólnego wiersza (potencjalnie zawierającym jednakową literę);
- każda taka para liter jednoznacznie identyfikuje jeden z kolorów *cyan*, *magenta*, *yellow* lub *white*.

Na przykład,

- jeśli pierwsze dwa kwadraty wiersza zawierają kolory „`CC`”, to odpowiadającą im płytka musi być koloru *cyan*, bo tylko dla tej płytki diagram zawiera 2 osobne kwadraty o kolorze „`C`”;
- jeżeli są to „`BR`”, to płytka musi mieć kolor *magenta*, bo tylko taka płytka może składać się na oba z tych kolorów;
- jeżeli są to „`CR`”, to już na tym etapie możemy stwierdzić, że rozwiązanie nie istnieje – albo możemy tymczasowo przydzielić płytce dowolny kolor, bo w następnym etapie zostanie to i tak wykryte.

Powyższa metoda pozwala nam ustalić potencjalne kolory każdej z płytek poziomych i pionowych. Po takim przydzieleniu pozostaje nam tylko sprawdzić, czy nasze rozwiązanie faktycznie skutkuje takim samym diagramem, jaki otrzymaliśmy na wejściu. Jeśli tak, to wypisujemy nasze rozwiązanie, a jeśli nie, to rozwiązanie nie istnieje.

Warto wspomnieć, że w treści znajduje się polecenie, by wypisać dowolne rozwiązanie, jeśli istnieje ich wiele – ale jak wykazano powyżej, zawsze istnieje co najwyżej jedno rozwiązanie, więc zdanie to było swego rodzaju „pułapką”.

41.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Modern_art_analysis_Gnomy/solution.py.

```
allowed_in_cyan = "BCG"
allowed_in_magenta = "BMR"
allowed_in_yellow = "GRY"
color_mixes = [
    "WCMY",
    "CCBG",
    "MBMR",
    "YGRY"
]
n = int(input())
grid = [input() for _ in range(n)]
answers = [n * [0], n * [0]]
for k in range(2):
    for i in range(n):
        can_be_cyan = True
        can_be_magenta = True
        can_be_yellow = True
        for j in range(2):
            color = grid[i][j] if k == 0 else grid[j][i]
            can_be_cyan = can_be_cyan and color in allowed_in_cyan
            can_be_magenta = can_be_magenta and color in allowed_in_magenta
            can_be_yellow = can_be_yellow and color in allowed_in_yellow
        if can_be_cyan:
            answers[k][i] = 1
        elif can_be_magenta:
            answers[k][i] = 2
        elif can_be_yellow:
            answers[k][i] = 3
correct = True
for i in range(n):
    for j in range(n):
        if grid[i][j] != color_mixes[answers[0][i]][answers[1][j]]:
            correct = False
if correct:
    for k in range(2):
        result = ""
        for i in range(n):
            result += color_mixes[0][answers[k][i]]
        print(result)
else:
    print("IMPOSSIBLE")
```

41.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Modern_art_analysis_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <string>
#include <vector>

bool contains(const std::string& s, char c) {
  return s.find(c) != std::string::npos;
}

int main() {
  const std::string allowed_in_cyan = "BCG";
  const std::string allowed_in_magenta = "BMR";
  const std::string allowed_in_yellow = "GRY";
  const std::string color_mixes[4] = {
    "WCMY",
    "CCBG",
    "MBMR",
    "YGRY",
  };
  int n;
  std::cin >> n;
  std::vector<std::string> grid(n);
  std::vector<int> answers[2];
  for (int i = 0; i < n; i++) {
    std::cin >> grid[i];
  }
  for (int k = 0; k < 2; k++) {
    answers[k].resize(n);
    for (int i = 0; i < n; i++) {
      bool can_be_cyan = true;
      bool can_be_magenta = true;
      bool can_be_yellow = true;
      for (int j = 0; j < 2; j++) {
        char color = k == 0 ? grid[i][j] : grid[j][i];
        can_be_cyan = can_be_cyan && contains(allowed_in_cyan, color);
        can_be_magenta = can_be_magenta && contains(allowed_in_magenta, color);
        can_be_yellow = can_be_yellow && contains(allowed_in_yellow, color);
      }
      if (can_be_cyan)
        answers[k][i] = 1;
      else if (can_be_magenta)
        answers[k][i] = 2;
      else if (can_be_yellow)
        answers[k][i] = 3;
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (grid[i][j] != color_mixes[answers[0][i]][answers[1][j]]) {
```

```
        std::cout << "IMPOSSIBLE";
        return 0;
      }
    }
  }
  for (int k = 0; k < 2; k++) {
    for (int i = 0; i < n; i++) {
      std::cout << color_mixes[0][answers[k][i]];
    }
    std::cout << '\n';
  }
  return 0;
}
```

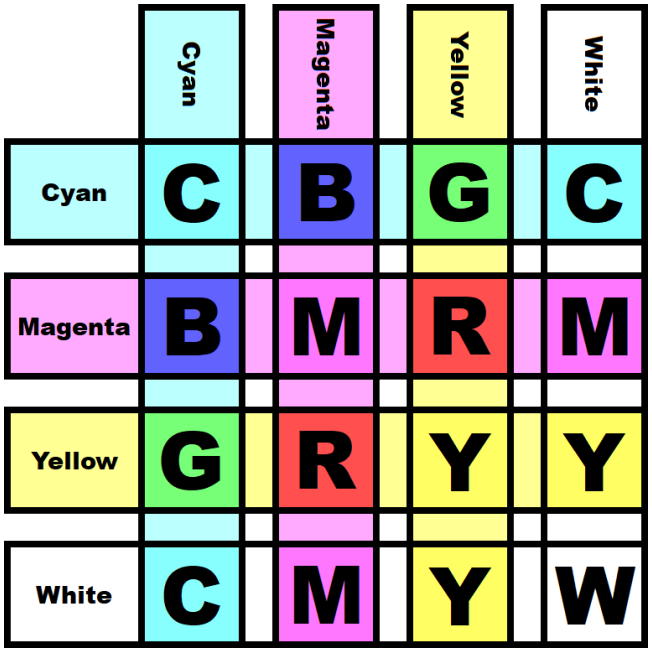
42 (V) – Modern Art Analysis – Skrzaty *(zad. w jęz. angielskim)*

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (II edycja – zawody algorytmiczne – etap regionalny)
Język programowania: Scratch

42.1 Task description

Your city has erected a new monument in the town square. It is an unusual and abstract piece of art: a square frame containing a large, colorful, translucent grid of small squares. Every small square is one of 7 colors: red, green, blue, cyan, magenta, yellow, or white.

After close observation of the monument you noticed that it is not actually built of individual squares, but of thin long strips divided into 2 layers – one vertical and one horizontal. You suspect that every strip is one of 4 colors: cyan, magenta, yellow, or white, and the remaining colors are simply combinations of overlapping strips on the two layers. The image below illustrates all the possible combinations of the colors and the results of mixing them.



Unfortunately the monument's frame covers its borders so you cannot see the individual stripes and directly confirm your suspicions. Instead, you noted down the colors of all the small squares in order to check if your hypothesis is plausible.

Task

Knowing the colors of the small squares, determine whether there exists an arrangement of colored strips that could produce the grid. If yes, find and print that arrangement, otherwise print "IMPOSSIBLE".

Start from a new, empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `DANE` should be manually filled with example input data, while the list `WYNIKI` is a place where your program should output the obtained results.

Input description

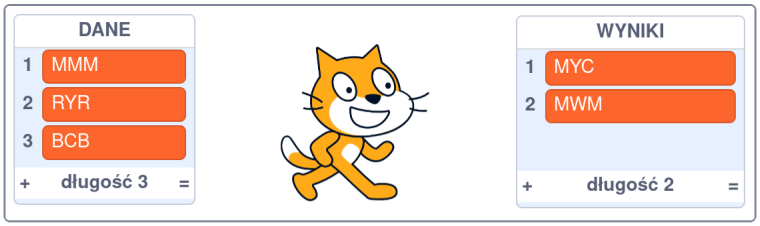
The input list `DANE` contains n elements ($2 \leq n \leq 1000$), describing the grid arrangement. Each element is a text string of length n. Every string consists only of capital letters `R, G, B, C, M, Y, W`, corresponding to appropriate colors as explained above. It is guaranteed that 2 adjacent columns will never be identical, and – similarly – that 2 adjacent rows will never be identical.

Output description

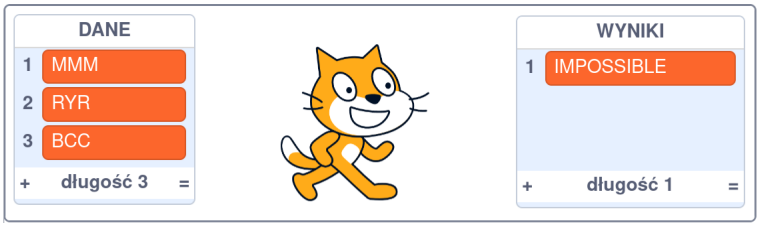
After execution of your program, the list `WYNIKI` should contain 2 elements: text strings of length n, each consisting only of capital letters `C, M, Y, W`. The first string should denote the colors used by the horizontal strips, and the second one – by the vertical strips. If multiple valid solutions exist, you can report any one of them. If the provided arrangement is impossible to obtain according to the rules described above, the list `WYNIKI` should contain only 1 element: the word `IMPOSSIBLE`.

Example

Example 1:

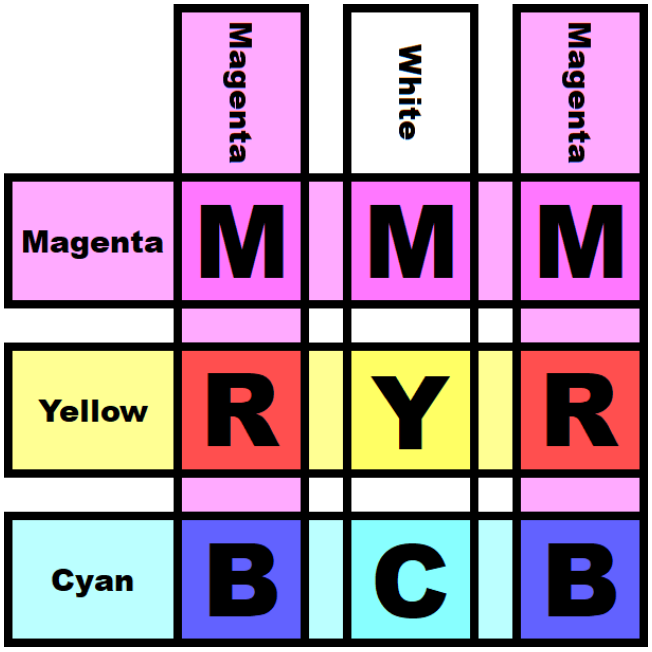


Example 2:



Explanation

The first test corresponds to the following situation:

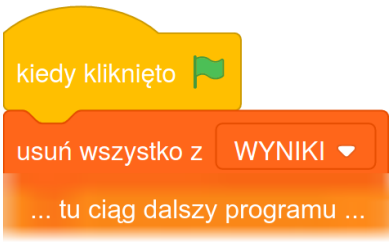


The second test is similar but replaces the color in the bottom right corner with cyan. There is actually no color strip that can produce both cyan and red when combined with other colors, so the rightmost column alone makes this arrangement impossible to satisfy.

Remarks

Remember, that the list `DANE` should be filled in manually, by typing the data with the keyboard. It is worth to test the results of your program also for input values different than those in the examples.

The content of the list `WYNIKI` should always be initially removed. The beginning of your program should therefore look like this:



42.2 Rozwiązanie

Rozwiązanie tego zadania w Scratchu jest zasadniczo zgodne z opisem dla Gnomów. Przykładową implementację znajdziesz w pliku:

./zadania/05_Dwuwymiarowe_struktury/Modern_art_analysis_Skrzaty/solution.sb3.

43 (V) – Projekt drona – Skrzaty

Autorzy: Jacek Starzyński, Politechnika Warszawska, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap finałowy)
Język programowania: Scratch

43.1 Treść zadania

Jasio wraz ze swoim wujkiem postanowili zbudować własnego drona. Wujek ma warsztat, a w nim wiele starych części, wśród których bez większych problemów udało się znaleźć podzespoły na projekt Jasia.

Prace szły pełną parą i były już na ukończeniu, gdy niespodziewanie wujek musiał opuścić Jasia i pomóc cioci. Większość była już gotowa – pozostało w zasadzie tylko wyznaczyć 2 punkty A i B oraz zamontować tam dwa śmigła, dlatego Jasio postanowił dokończyć projekt samodzielnie.

Jednakże – pomimo wielkiego zapału i ambicji młodego konstruktora – istnieje ryzyko, że mogło mu zabraknąć doświadczenia i niezbędnej precyzji, aby zamontować śmigła w poprawny sposób, tzn. tak, by po uruchomieniu maszyny nie zderzyły się ze sobą. Twoim zadaniem jest sprawdzenie, czy dla danej konfiguracji śmigieł, po uruchomieniu drona śmigła się nie zderzą.

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie danymi dotyczącymi konfiguracji śmigieł, zaś do listy `WYNIKI` Twój program powinien wpisać efekty swojego działania, czyli wyniki sprawdzenia, czy dojdzie do kolizji.

Opis wejścia

Pierwszy element na liście wejściowej `DANE` to liczba naturalna T oznaczająca liczbę testów, jakie należy przeprowadzić ($1 \leq T \leq 10$). Każdy test opisany jest kolejnymi ośmioma elementami listy `DANE`, określającymi położenie i rozmiary obu śmigieł. Są to odpowiednio (w tej kolejności):

- x_{s1} – współrzędna x środka pierwszego śmigła;
- y_{s1} – współrzędna y środka pierwszego śmigła;
- x_{k1} – współrzędna x końca pierwszego śmigła;
- y_{k1} – współrzędna y końca pierwszego śmigła;
- x_{s2} – współrzędna x środka drugiego śmigła;
- y_{s2} – współrzędna y środka drugiego śmigła;
- x_{k2} – współrzędna x końca drugiego śmigła;
- y_{k2} – współrzędna y końca drugiego śmigła.

Wszystkie współrzędne są liczbami całkowitymi z zakresu $[-50, 50]$.

Opis wyjścia

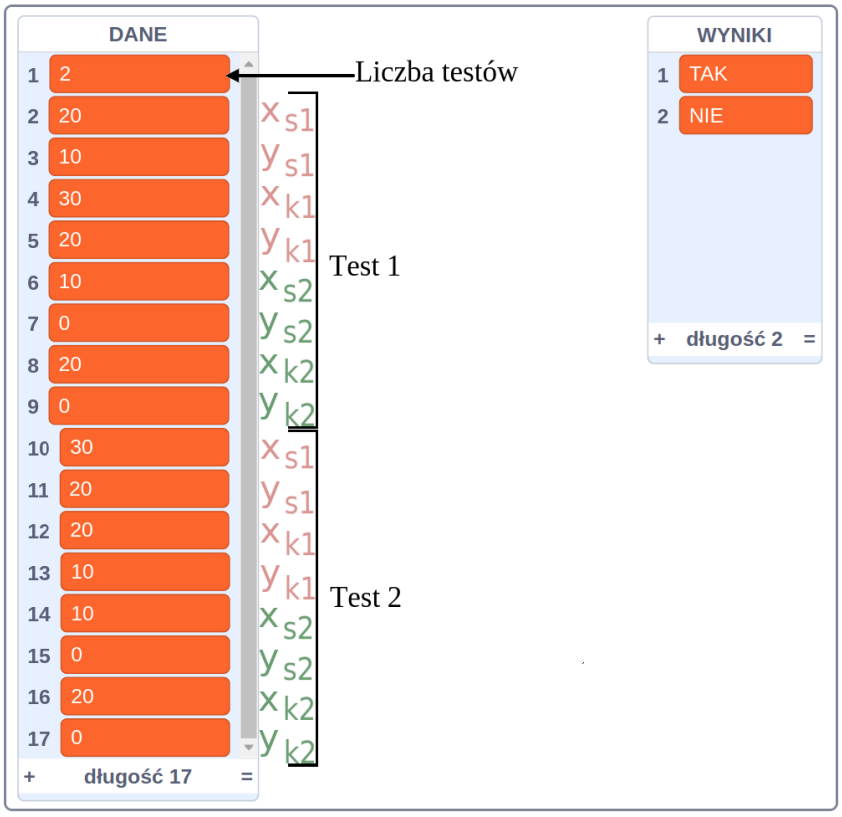
Twój program po uruchomieniu (za pomocą zielonej flagi) powinien dla każdego z T testów wypisać na liście wyjściowej `WYNIKI`, jedno słowo (koniecznie wielkimi literami):

- TAK, jeśli śmigła mogą się zderzyć;
- NIE, jeśli zderzenie śmigieł jest niemożliwe.

Uwaga: możesz założyć, że wynik będzie zawsze jednoznaczny (tzn. że śmigła nigdy nie będą ustawione „na styk").

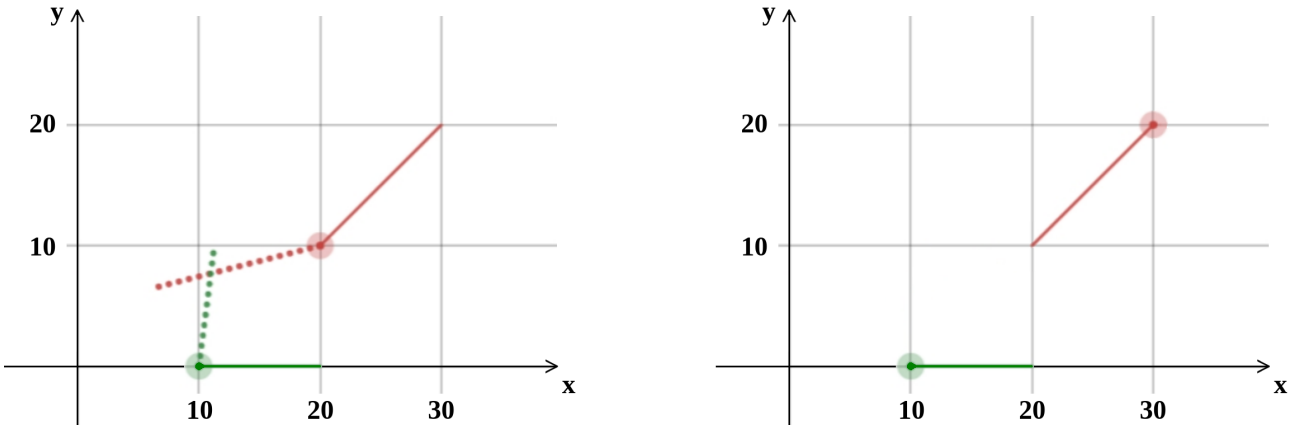
Przykład

Na poniższym rysunku przedstawiono przykładowe dane wejściowe i poprawne odpowiedzi.



Rysunek 43.1: Przykładowe wejście i oczekiwane wyjście programu (objaśnienia po prawej stronie listy DANE są podane tylko dla ilustracji – nie ma potrzeby uwzględniać ich w programie)

Jak widać, mamy tu dwa przypadki testowe do rozważenia. Oba zostały zaprezentowane na rysunku 43.2 (na kolejnej stronie). W obu przypadkach pierwsze śmigło jest narysowane kolorem czerwonym, natomiast drugie – zielonym. Środek śmigła jest zaznaczony kółkiem. W pierwszym przypadku może dojść do kolizji (liniami przerywanymi są zaznaczone pozycje śmigieł w chwili potencjalnego zderzenia) i dlatego na pierwszej pozycji listy `WYNIKI` program wypisał słowo TAK. Na drugiej pozycji zostało wypisane słowo NIE, bo w drugim teście zderzenie jest niemożliwe.



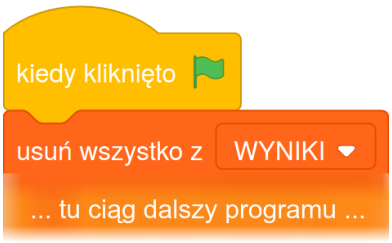
Rysunek 43.2: Graficzna reprezentacja pierwszego przypadku testowego (po lewej) i drugiego (po prawej)

Uwagi

Zadanie może zostać rozwiązane w dowolny sposób. Możesz wykorzystać w tym celu dowolne duszki (np. reprezentujące śmigła) lub też wykonać niezbędne obliczenia bez pomocy duszków. Ważne jest oczywiście, aby finalna zawartość listy `WYNIKI` była prawidłowa, a także aby Twój program wykonał się w możliwie najkrótszym czasie.

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



43.2 Rozwiązanie

To zadanie – jak zauważono w treści – można rozwiązać na różne sposoby. W przypadku starszych uczniów, narzucającą się metodą jest wykorzystanie twierdzenia Pitagorasa i bezpośredni pomiar długości śmigieł oraz odległości między ich środkami. Jeśli odległość między środkami przekracza sumę długości śmigieł, to nigdy nie będą w stanie się zderzyć (odpowiedź „ NIE”). W przeciwnym wypadku – kolizji nie można wykluczyć (odpowiedź „TAK”).

Tę implementację znajdziesz w pliku:

./zadania/05_Dwuwymiarowe_struktury/Projekt_drona/ProjektDronaWZORCOWYcomp.sb3.

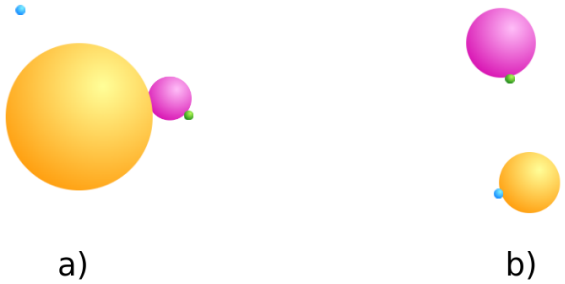
Inną metodę rozwiązania otrzymamy, wykorzystując możliwości Scratcha w zakresie tworzenia animacji za pomocą duszków. Można skonstruować dwa faktycznie obracające się śmigła i sprawdzić czy dojdzie do kolizji. Należy jednak zauważyć, że nawet jeśli kolizja jest możliwa, to mogłoby się tak zdarzyć, że – przy określonych położeniach początkowych i prędkości obrotowej śmigieł – w praktyce nigdy by do niej nie doszło (obracające się „synchronicznie” śmigła mogłyby się zawsze mijać).

Z tego względu, w poniższym rozwiązaniu przykładowym:

./zadania/05_Dwuwymiarowe_struktury/Projekt_drona/ProjektDronaWZORCOWYanim.sb3

zaproponowano inne podejście, w którym korzystamy z dwóch duszków w kształcie koła (`Ball`) o rozmiarach obejmujących cały obszar „omiatany” przez jedno i drugie śmigło, odpowiednio. Ściślej rzecz biorąc, pierwszy duszek umieszczany jest tam, gdzie znajduje się środek pierwszego śmigła i ma początkowo ustawiony minimalny rozmiar. Rozmiar ten stopniowo zwiększamy, przy czym ogranicznikiem wzrostu jest dodatkowy duszek, ustawiony w miejscu podanym jako koniec pierwszego śmigła (dzięki temu unikamy konieczności bezpośredniego obliczania rozmiaru śmigła). Z drugim duszkiem, reprezentującym drugie śmigło, postępujemy tak samo (i tu też potrzebny jest dodatkowy duszek reprezentujący koniec drugiego śmigła). Kiedy oba „śmigła” urosną już do właściwych rozmiarów, wystarczy sprawdzić, czy dotykają się nawzajem. Tak naprawdę, możemy to sprawdzać na bieżąco, jeszcze podczas fazy wzrostu, co pozwoli na wcześniejsze przerwanie symulacji w wielu przypadkach.

Na poniższym rysunku przedstawiono przykłady sytuacji, w których może dojść do kolizji (a) lub kolizję można wykluczyć (b):



Rysunek 43.3: a) Możliwa kolizja śmigieł; b) Kolizja śmigieł wykluczona

Duszek w kolorze żółtym reprezentuje pierwsze śmigło (niebieski duszek, to jego koniec), a duszek w kolorze różowym – drugie (zielony duszek, to koniec drugiego śmigła).

44 (V) – Baza księżycowa – Gnomy

Autor: ***Łukasz Skonieczny***, ***Politechnika Warszawska***
Kategoria: ***Gnomy (IV edycja – liga algorytmiczna)***
Język programowania: ***C++/Python***

44.1 Treść zadania

Królestwo Lunatocji po dwóch wiekach raczkującej eksploracji kosmosu postanowiło postawić pierwszy wielki krok – wybudować własną bazę księżycową! Powołana do tego jednostka o kryptonimie CMI (*Cardinal Moonbase Initiative*) rozpoczęła przygotowania od wyboru optymalnego miejsca na powierzchni księżyca. Teren przeznaczony na budowę bazy musi być płaski, a ponieważ baza ma służyć także celom naukowym – między innymi wykrywaniu fal grawitacyjnych – jej konstrukcja musi się składać z dwóch podłużnych segmentów przecinających się pod kątem prostym. Dzięki różnorakim sztucznym satelitom, od lat krążącym wokół srebrnego globu, uzyskano szereg map zawierających wysokości poziomu powierzchni księżyca. Należy je teraz przeanalizować, by wybrać płaski obszar pozwalający na zbudowanie jak największej bazy.

Zadanie

Mapa regionu księżyca jest kwadratową, ortogonalną siatką zawierającą n^2 kwadratowych komórek. Każda komórka reprezentuje obszar o rozmiarze 5 na 5 metrów. Znane są uśrednione wysokości powierzchni każdej komórki z dokładnością do jednego metra. Planowana baza księżycowa składa się dwóch segmentów o szerokości jednej komórki i długości co najmniej jednej komórki. Segmenty muszą być ułożone na powierzchni o tej samej wysokości, prostopadle do siebie, ortogonalnie do siatki i nachodząc na siebie w jednym miejscu. Na danej mapie należy znaleźć maksymalną wielkość możliwej do zbudowania bazy liczonej jako suma dwóch segmentów.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita n (gdzie: $1 \leq n \leq 2000$), oznaczająca liczbę wierszy i kolumn kwadratowej mapy. W kolejnych n liniach znajdują się ciągi liczb oznaczające wysokości kolejnych komórek. Są to liczby z przedziału [0, 1100]. W każdym wierszu jest n liczb (a zatem łącznie jest n^2 liczb do odczytania).

Opis wyjścia

Na standardowe wyjście należy wypisać rozmiar największej bazy, która może być wybudowana w zadanym regionie. Wielkość bazy liczona jest jako suma długości (czyli liczby zajmowanych komórek) dwóch przecinających się segmentów.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
5
1 2 2 4 5
1 2 2 2 5
3 3 2 2 2
5 5 2 2 5
5 5 3 2 2
```

prawidłową odpowiedzią jest:

```
7
```

Z kolei dla innego wejścia:

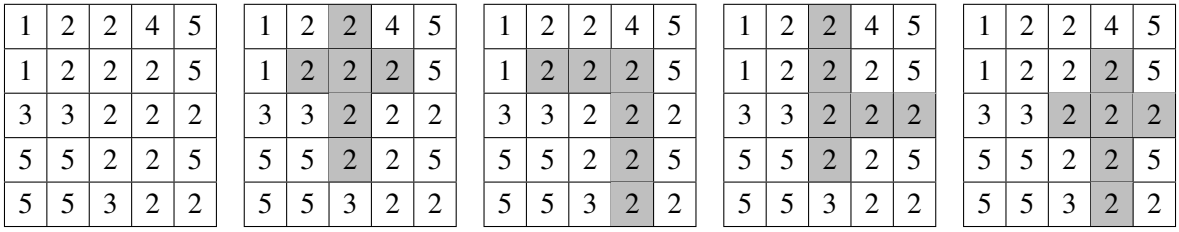
```
4
1 1 2 0
1 3 2 0
2 3 2 0
2 2 1 0
```

prawidłową odpowiedzią jest:

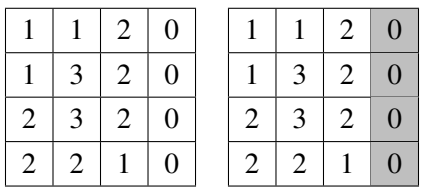
```
5
```

Wyjaśnienie przykładów

Pierwszy przypadek jest przedstawiony na poniższym rysunku. Możliwe są cztery rozmieszczenia maksymalnej bazy. Baza ma wielkość 7, gdyż w każdym przypadku składa się z segmentu pionowego o długości 4 i segmentu poziomego o długości 3.



Drugi przypadek jest przedstawiony na rysunku poniżej. Największa baza składa się z segmentu pionowego o długości 4 oraz segmentu poziomego o długości 1, co łącznie daje bazę o wielkości 5. Segment poziomy może przecinać segment pionowy w dowolnym z czterech możliwych miejsc.



44.2 Rozwiązanie

Wyznaczenie maksymalnej bazy, której „środek” (miejsce przecięcia dwóch segmentów) znajduje się w danej komórce jest względnie proste. Wystarczy znaleźć liczbę komórek na prawo/lewo/górę/dół od niej, które mają tę samą wysokość, co komórka, od której zaczęliśmy. W najgorszym przypadku zrobimy to w czasie liniowym $O(n)$, przy czym „najgorszy przypadek” oznacza, że poziomy i/lub pionowy pas o tej samej wysokości ciągnie się przez prawie cały obszar mapy. Problem polega na tym, że mapa składa się z n^2 komórek, więc sprawdzenie każdej z nich niezależnie w ten sposób, a następnie wybranie największej uzyskanej wielkości, będzie miało złożoność $O(n^3)$. Chcemy to zrobić szybciej.

Zauważmy, że jeżeli dla każdej komórki wyznaczymy niezależnie maksymalną długość segmentu poziomego i segmentu pionowego, to wielkość maksymalnej bazy uzyskamy sumując obie wartości. Ten sposób ma jednak tę zaletę, że policzone wartości (długości segmentów w pionie/poziomie) są wspólne dla wszystkich komórek tej samej wysokości na lewo/prawo oraz w górę/dół od startowej komórki. Zatem za jednym zamachem wyznaczymy te wartości potencjalnie dla wielu komórek, co ostatecznie pozwoli nam uzyskać łączną złożoność $O(n^2)$.

Aby to zrealizować, ponumerujmy wszystkie pionowe i poziome ciągłe obszary o tej samej wysokości oraz utwórzmy słownik wskazujący jaką długość ma obszar o podanym numerze. Innymi słowy, każda komórka mapy będzie dodatkowo (poza wysokością) zawierała numer obszaru pionowego i poziomego, do którego należy. W rozwiązaniu wzorcowym (podanym na następnych stronach) te informacje przechowywane są w tablicach `xflat` oraz `yflat`, które mają taki sam rozmiar jak cała mapa. Ponieważ zakładamy, że obszary poziome oraz pionowe będą numerowane niezależnie (w obu przypadkach od zera) – potrzebne są więc dwa słowniki przyporządkowujące wysokości dla obszarów pionowych i poziomych. Mają one nazwy odpowiednio `xlength` i `ylentgh`. Wartości wszystkich struktur po wypełnieniu (dla pierwszego przykładu z treści zadania) pokazano w poniższych tabelach:

data

1	2	2	4	5
1	2	2	2	5
3	3	2	2	2
5	5	2	2	5
5	5	3	2	2

xflat

0	1	1	2	3
4	5	5	5	6
7	7	8	8	8
9	9	10	10	11
12	12	13	14	14

yflat

0	3	6	8	10
0	3	6	9	10
1	4	6	9	11
2	5	6	9	12
2	5	7	9	13

`xlength:`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	1	1	1	3	1	2	3	2	2	1	2	1	2

`ylength:`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	1	2	2	1	2	4	1	1	4	2	1	1	1

Można stąd szybko odczytać wielkości bazy umieszczone w dowolnej komórce. Na przykład, komórka znajdująca się w samym środku mapy (`data`) ma wysokość 2. Komórka należy do regionu poziomego o numerze 8 (`xflat`) oraz regionu pionowego o numerze 6 (`yflat`). Rozmiary tych regionów wynoszą odpowiednio: 3 (`xlength[8]`) oraz 4 (`ylength[6]`). Baza ze środkiem w tej komórce miałaby więc wielkość $3 + 4 = 7$.
W ten sposób można szybko sprawdzić każdą komórkę i wybrać tę, która zapewni nam największą wartość: `xlength[xflat[i][j]] + ylength[yflat[i][j]]`.

44.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Baza_ksiezycowa_Gnomy/solution.py.

```
def main():
    n = int(input())
    data = [list(map(int,input().strip().split())) for i in range(0, n)]
    xflat = [[0]*n for i in range(0, n)]
    yflat = [[0]*n for i in range(0, n)]
    xlength = {}
    ylength = {}
    # fill x
    flats = 0;
    for i in range(0, n):
        len = 1;
        lev = -1;
        for j in range(0, n):
            if data[i][j] != lev:
                flats += 1
                lev = data[i][j]
                len = 1
            else:
                xflat[i][j] = flats;
                len += 1
            xflat[i][j] = flats
            xlength[flats] = len
    # fill y
    flats = 0;
    for j in range(0, n):
        len = 1;
        lev = -1;
        for i in range(0, n):
            if data[i][j] != lev:
                flats += 1
                lev = data[i][j]
                len = 1
            else:
                yflat[i][j] = flats;
                len += 1
            yflat[i][j] = flats
            ylength[flats] = len
    best = 0;
    for i in range(0, n):
        for j in range(0, n):
            sum = xlength[xflat[i][j]] + ylength[yflat[i][j]]
            if sum > best:
                best = sum;
    print(best)

if __name__ == '__main__':
    main()
```

44.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Baza_ksiezycowa_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <algorithm>
#include <map>

using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >>n;

    int** data = new int*[n];
    int** xflat = new int*[n];
    int** yflat = new int*[n];
    map<int,int> xlength;
    map<int,int> ylength;

    for (int i=0; i<n; i++) {
        data[i] = new int[n];
        xflat[i] = new int[n];
        yflat[i] = new int[n];
    }

    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            cin >>data[i][j];
            xflat[i][j] = 0;
            yflat[i][j] = 0;
        }
    }

    //fill x
    int flats = 0;
    for (int i=0; i<n; i++) {
        int len = 1;
        int lev = -1;
        for (int j=0; j<n; j++) {
            if (data[i][j] != lev) {
                flats++;
                lev = data[i][j];
                len = 1;
            } else {
                xflat[i][j] = flats;
                len++;
            }
            xflat[i][j] = flats;
```

```
            xlength[flats] = len;
        }
    }

    //fill y
    flats = 0;
    for (int j=0; j<n; j++) {
        int len = 1;
        int lev = -1;
        for (int i=0; i<n; i++) {
            if (data[i][j] != lev) {
                flats++;
                lev = data[i][j];
                len = 1;
            } else {
                yflat[i][j] = flats;
                len++;
            }
            yflat[i][j] = flats;
            ylength[flats] = len;
        }
    }

    int best = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            int sum = xlength[xflat[i][j]] + ylength[yflat[i][j]];
            if (sum > best) {
                best = sum;
            }
        }
    }
    cout <<best<<endl;
}
```

45 (V) – Baza księżycowa – Skrzaty

Autorzy: ***Łukasz Skonieczny***, ***Politechnika Warszawska***, ***Bartłomiej Stasiak***, ***Politechnika Łódzka***
Kategoria: ***Skrzaty (IV edycja – liga algorytmiczna)***
Język programowania: ***Scratch***

45.1 Treść zadania

Królestwo Lunatocji po dwóch wiekach raczkującej eksploracji kosmosu postanowiło postawić pierwszy wielki krok – wybudować własną bazę księżycową! Powołana do tego jednostka o kryptonimie CMI (*Cardinal Moonbase Initiative*) rozpoczęła przygotowania od wyboru optymalnego miejsca na powierzchni księżyca. Teren przeznaczony na budowę bazy musi być płaski, a ponieważ baza ma służyć także celom naukowym – między innymi wykrywaniu fal grawitacyjnych – jej konstrukcja musi się składać z dwóch podłużnych segmentów przecinających się pod kątem prostym. Dzięki różnorakim sztucznym satelitom, od lat krążącym wokół srebrnego globu, uzyskano szereg map zawierających wysokości poziomu powierzchni księżyca. Należy je teraz przeanalizować, by wybrać płaski obszar pozwalający na zbudowanie jak największej bazy.

Zadanie

Mapa regionu księżyca jest kwadratową, ortogonalną siatką zawierającą n^2 kwadratowych komórek. Każda komórka reprezentuje obszar o rozmiarze 5 na 5 metrów. Znane są uśrednione wysokości powierzchni każdej komórki z dokładnością do jednego metra. Planowana baza księżycowa składa się dwóch segmentów o szerokości jednej komórki i długości co najmniej jednej komórki. Segmenty muszą być ułożone na powierzchni o tej samej wysokości, prostopadle do siebie, ortogonalnie do siatki i nachodząc na siebie w jednym miejscu. Na danej mapie należy znaleźć maksymalną wielkość możliwej do zbudowania bazy liczonej jako suma dwóch segmentów.

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, wpisując do niej reprezentację mapy regionu księżyca, zaś na liście `WYNIKI` Twój program powinien umieścić rezultat swojego działania, czyli maksymalną wielkość bazy możliwej do zbudowania.

Opis wejścia

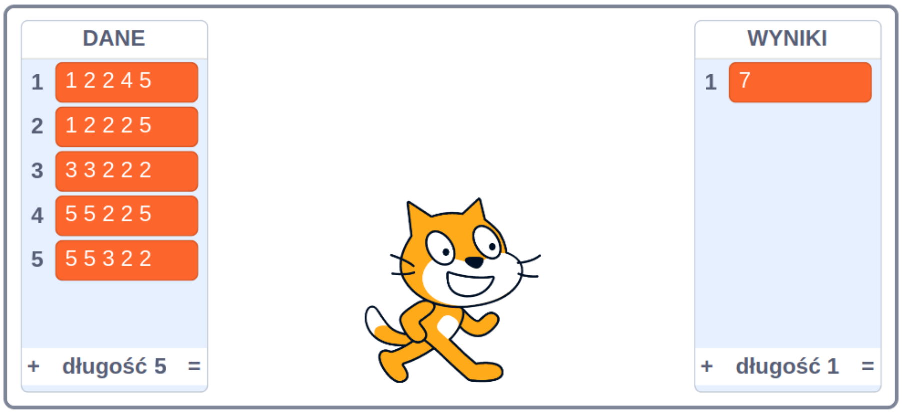
Lista wejściowa `DANE` zawiera n elementów ($1 \leq n \leq 300$), z których każdy stanowi ciąg n liczb (z przedziału od 0 do 1100), oddzielanych pojedynczą spacją. Liczby te oznaczają wysokości poszczególnych komórek kwadratowej mapy.

Opis wyjścia

Na liście wyjściowej `WYNIKI` należy umieścić jeden element: liczbę całkowitą oznaczającą rozmiar największej bazy, która może być wybudowana w zadanym regionie. Wielkość bazy liczona jest jako suma długości (czyli liczby zajmowanych komórek) dwóch przecinających się segmentów.

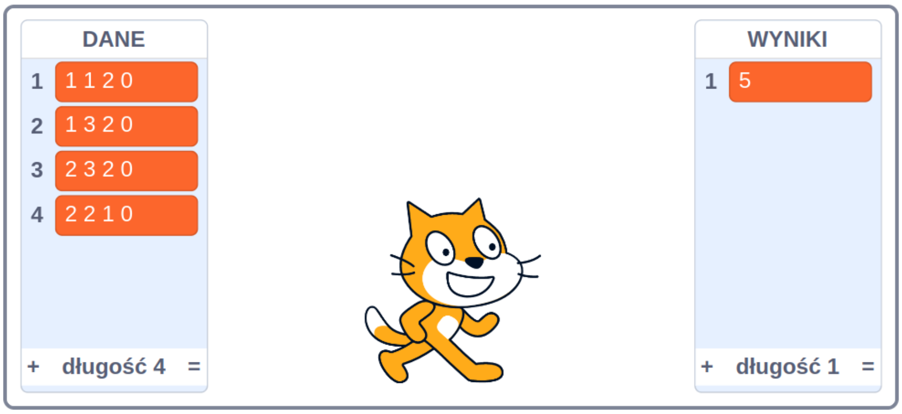
Przykłady

Na poniższym rysunku pokazano przykładowe dane wejściowe i wynik działania programu:



Rysunek 45.1: Przykładowy wygląd planszy po zakończeniu programu

Inne dane wejściowe i wynik działania programu przedstawiono poniżej:



Rysunek 45.2: Przykładowy wygląd planszy po zakończeniu programu

Wyjaśnienie przykładów

Pierwszy przypadek jest przedstawiony na poniższym rysunku. Możliwe są cztery rozmieszczenia maksymalnej bazy. Baza ma wielkość 7, gdyż w każdym przypadku składa się z segmentu pionowego o długości 4 i segmentu poziomego o długości 3.

1	2	2	4	5
1	2	2	2	5
3	3	2	2	2
5	5	2	2	5
5	5	3	2	2

1	2	**2**	4	5
1	**2**	**2**	**2**	5
3	3	**2**	2	2
5	5	**2**	2	5
5	5	3	2	2

1	2	2	4	5
1	**2**	**2**	**2**	5
3	3	2	**2**	2
5	5	2	**2**	5
5	5	3	**2**	2

1	2	**2**	4	5
1	2	**2**	2	5
3	3	**2**	**2**	**2**
5	5	**2**	2	5
5	5	3	2	2

1	2	2	4	5
1	2	2	**2**	5
3	3	**2**	**2**	**2**
5	5	2	**2**	5
5	5	3	**2**	2

Drugi przypadek jest przedstawiony na rysunku poniżej. Największa baza składa się z segmentu pionowego o długości 4 oraz segmentu poziomego o długości 1, co łącznie daje bazę o wielkości 5. Segment poziomy może przecinać segment pionowy w dowolnym z czterech możliwych miejsc.

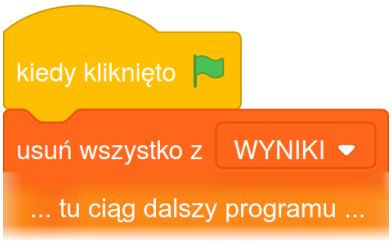
1	1	2	0
1	3	2	0
2	3	2	0
2	2	1	0

1	1	2	0
1	3	2	0
2	3	2	0
2	2	1	0

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:

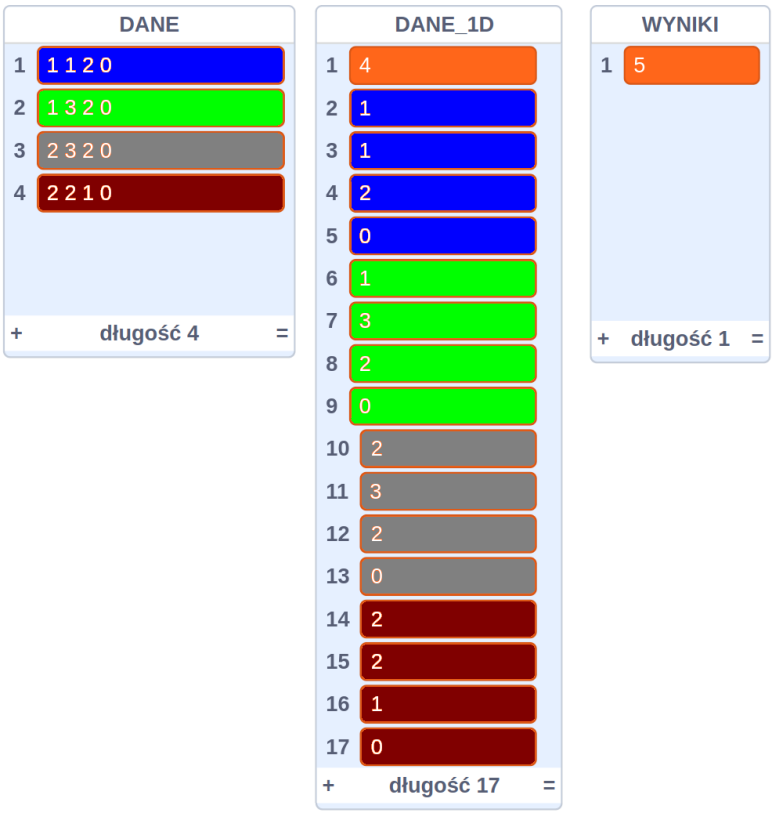


Rysunek 45.3: Początek programu

45.2 Rozwiązanie

Rozwiązanie tego zadania opiera się na tym samym podejściu, co w przypadku wersji dla Gnomów, chociaż implementacja w Scratchu jest znacząco trudniejsza, z uwagi na brak struktur do przechowywania tablic dwuwymiarowych.

Dane wejściowe podawane są co prawda w liście `DANE` w sposób analogiczny, jak w wersji dla Gnomów, ale ta postać jest bardzo niewygodna do przetwarzania. Każdy element listy `DANE` reprezentuje jeden wiersz mapy, ale jego poszczególne komórki, oddzielane spacjami, nie są traktowane jako liczby, tylko fragmenty napisu[1]. Z tego względu w pierwszym kroku przekształcamy listę `DANE` do postaci jednowymiarowej listy `DANE_1D`, której każdy element jest pojedynczą liczbą, reprezentującą pojedynczą komórkę mapy. Na poniższym rysunku przedstawiono przykładowe dane z treści zadania (przypadek 2) przed i po konwersji do postaci jednowymiarowej (1D). Dodatkowy, pierwszy element listy `DANE_1D` przechowuje liczbę wierszy/kolumn mapy (stąd długość listy `DANE_1D` wynosi $1 + 4 \times 4 = 17$).



Rysunek 45.4: Reprezentacja mapy w postaci tablicy jednowymiarowej (osobnymi kolorami zaznaczono osobne wiersze mapy)

Ta reprezentacja jest już prostsza do przetwarzania. Jedyne czego potrzebujemy, to sposób przeliczania współrzędnych komórki na mapie (numer wiersza, numer kolumny) na numer tej komórki na liście `DANE_1D`. Wygodnie jest tu przyjąć, że wiersze i kolumny numerowane są od zera, podobnie jak w implementacji w C++ i w Pythonie. Gdyby nasza jednowymiarowa lista przechowująca kolejne komórki była także indeksowana od zera, wówczas numer komórki o współrzędnych (w, k) na tej liście byłby równy $N \cdot w + k$ (gdzie N jest długością wiersza).

[1] Zauważmy też, że każdy taki fragment może mieć różną liczbę znaków (w zależności od liczby cyfr w liczbie, którą reprezentuje), co dodatkowo komplikuje sprawę.

Ponieważ jednak listy w Scratchu indeksowane są od 1, a ponadto dołożyliśmy dodatkowo na pierwszej pozycji liczbę wierszy/kolumn mapy, więc ostatecznie numer komórki obliczamy jako:

Przykładowo, druga komórka ostatniego wiersza („czerwonego”), czyli komórka o współrzędnych:

$(w, k) = (3, 1)$,

znajduje się w tablicy `DANE_1D` na pozycji $4 \cdot 3 + 1 + 2 = 15$.

W analogiczny, jednowymiarowy sposób reprezentowane (i indeksowane) są również tablice `xflat` i `yflat`.

Kompletne rozwiązanie znajdziesz w pliku:

./zadania/05_Dwuwymiarowe_struktury/Baza_ksiezycowa_Skrzaty/solution.sb3.

46 (V) – Pizza Picnic Picle – Gremliny *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (II edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

46.1 Task description

A small town is organizing a public picnic in the forest. The highlight of the event is supposed to be a gigantic pizza made as a collaborative effort by local chefs and bakers. One of the organizers noticed a problem with this plan: the pizza must be transported from the outdoor kitchen to the picnic area, which is surrounded by trees. If it is too big, it might be impossible to carry it between the trees in one piece. For this reason, it has become essential to determine the maximum possible size of the pizza.

During delivery, the pizza will be inside a square cardboard box with sides equal to the pizza diameter. It will have to be carried by multiple people, so the delivery should not require any complex maneuvers such as rotating the box – instead, it should just be a simple sequence of movements in straight lines.

Task

Given a detailed map of the forest area with all the trees marked, determine the maximum diameter of a pizza that can be transported from the kitchen in the bottom left corner to the picnic site in the top right. The path, including the initial and final positions of the pizza box, cannot contain any trees and cannot even partially leave the boundaries of the map.

Input description

The first line of standard input contains two integers w, h ($2 \leq w, h \leq 2000$) separated by a single space – the width and the height of the map, respectively. The following h lines contain strings of characters of length w each. Each string consists only of characters "`X`" symbolizing trees and "`.`" symbolizing empty space. Assume that every tree completely occupies the 1×1 square it is growing in. It is guaranteed that the map contains at least 1 tree, and that a pizza of diameter 1 can be delivered to the picnic site.

Output description

Your program should print one integer to standard output – the maximum diameter of a pizza that can be transported to the picnic site.

Example

For the following sample input:

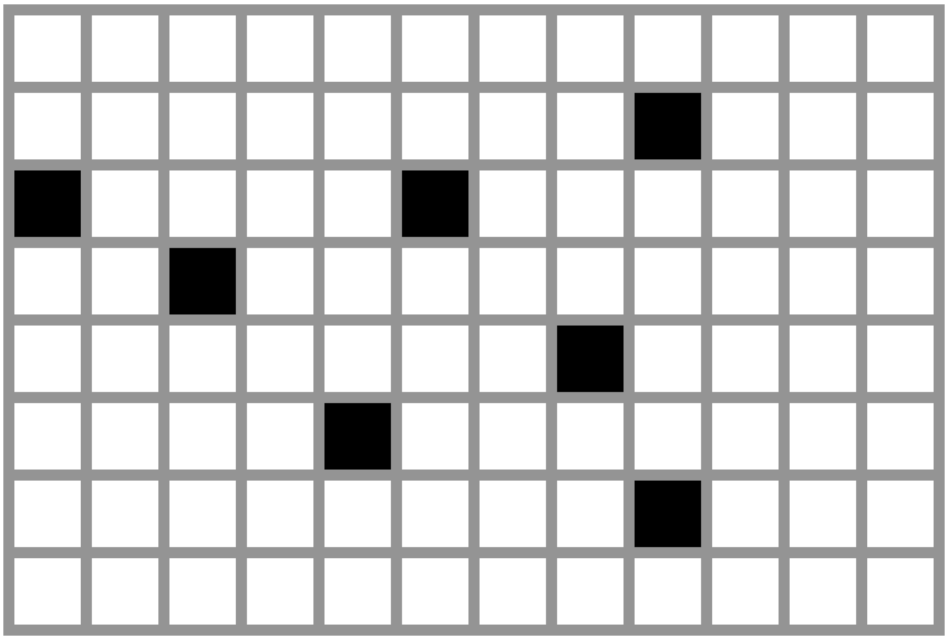
```
12 8
............
........X...
X....X......
..X.........
.......X....
....X.......
........X...
............
```

the correct output is:

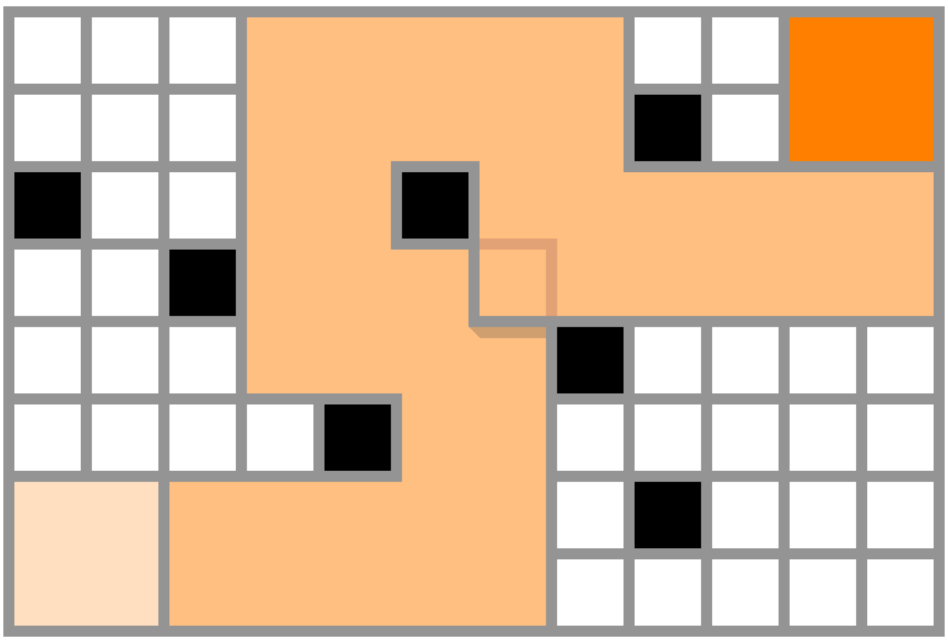
```
2
```

Explanation

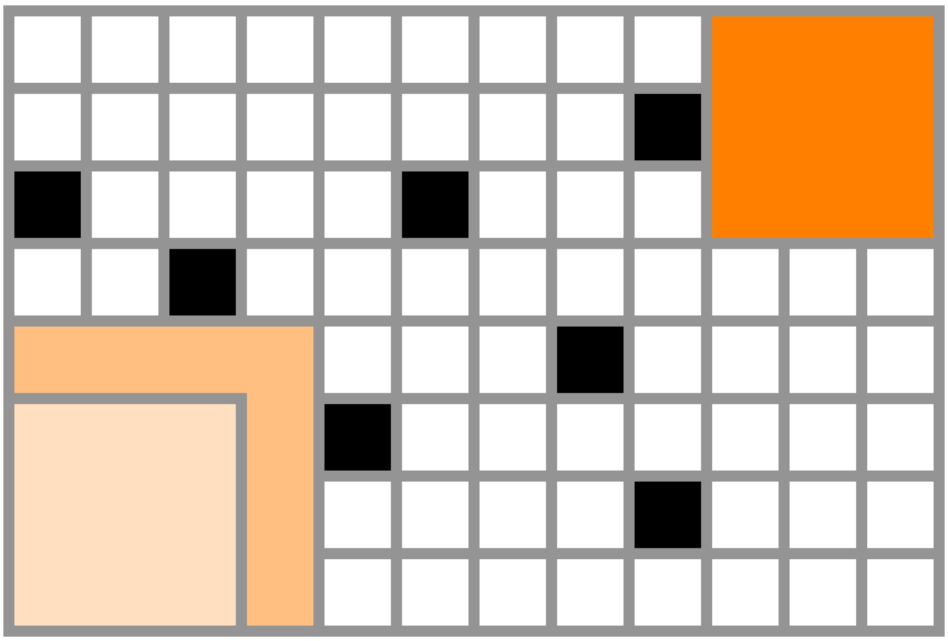
The input corresponds to the following map, with trees marked in black:



One of the possible paths for a pizza of diameter 2 is shown in orange below.



On the other hand, a pizza of diameter 3 could not even be carried outside of the 4×4 square near the kitchen before getting stuck between trees.



Therefore, the correct answer is 2.

46.2 Rozwiązanie

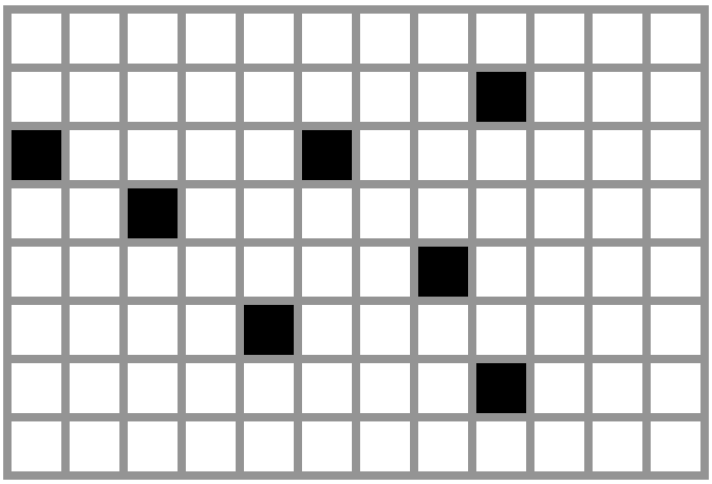
Opis zadania:

Podana jest prostokątna mapa składająca się z kwadratów jednostkowych, z których niektóre zawierają przeszkody. Znajdź maksymalny bok kwadratu, którym da się przejść z lewego dolnego do prawego górnego narożnika mapy.

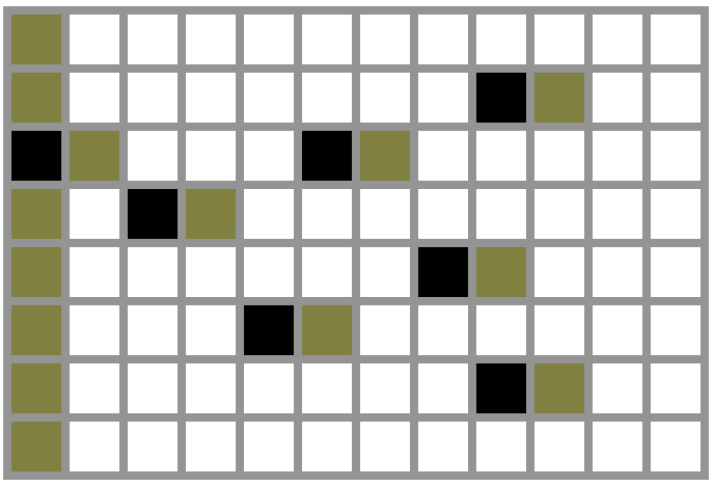
Opis rozwiązania:

Dla ustalonego boku kwadratu możemy łatwo ustalić, czy przejście mapy jest możliwe – kluczowe jest tutaj spostrzeżenie, że zamiast rozważać transport dużego kwadratu wokół małych przeszkód, możemy powiększyć przeszkody, a kwadrat pozostawić jednostkowym.

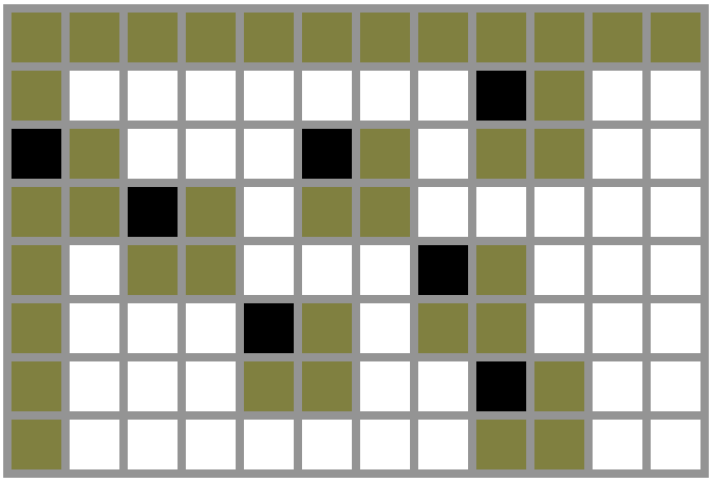
Rozważmy przykładową mapę i kwadrat o wymiarach 2×2:



Możemy w pierwszym kroku wydłużyć wszystkie przeszkody w poziomie:



a następnie w pionie:



Przejście po takiej mapie kwadratem jednostkowym jest równoważne przejściu po oryginalnej mapie kwadratem 2×2, a da się je wykonać szybko, tj. w czasie $O(wh)$. Wykorzystamy w tym celu prosty algorytm typu *flood-fill*, używany m.in. w programach graficznych do automatycznego wypełniania kolorem obszarów na rysunku[1]. Idea jest prosta – jeśli rozpoczniemy zamalowywanie mapy w jednym narożniku i uda się nam dotrzeć do przeciwległego narożnika, to znaczy, że przejście istnieje (oczywiście zakładamy, że mapa została wstępnie zainicjalizowana przez zamalowanie wszystkich pozycji przeszkód).

Algorytm *flood-fill* sprowadza się do sprawdzenia czterech sąsiadów[2] danego piksela obrazu (w naszym przypadku: danego kwadratu jednostkowego mapy). Jeśli dany sąsiad nie jest jeszcze zamalowany, wówczas go zamalowujemy i przeprowadzamy sprawdzanie z kolei *jego* czterech sąsiadów. Algorytm ten jest więc jak widać algorytmem z natury rekurencyjnym. Z drugiej strony, zasadniczo każdą rekurencję możemy zastąpić rozwiązaniem iteracyjnym, unikając potencjalnych problemów z przepełnieniem stosu systemowego podczas wielokrotnie zagnieżdżonych wywołań funkcji. W tym celu należy wykorzystać odpowiednią strukturę danych (np. stos lub kolejkę), w której będą umieszczani sąsiedzi bieżącego kwadratu – do późniejszego sprawdzenia. W przedstawionych dalej przykładowych implementacjach użyto stosu (p. ostatni fragment funkcji `da_sie_przejechac`), na którym początkowo kładziemy tylko współrzędne narożnika startowego, a współrzędne kolejnych kwadratów dodawane są podczas sprawdzania sąsiadów (i ich sąsiadów, sąsiadów ich sąsiadów, itd.).

Na koniec zauważmy, że przedstawiony algorytm będziemy musieli wykonać wielokrotnie – dla różnych rozmiarów pizzy (a właściwie – równoważnie – dla różnych rozmiarów przeszkód, jak zauważyliśmy wyżej). Aby przyspieszyć znalezienie maksymalnego dopuszczalnego rozmiaru, zastosujemy poznane już wcześniej podejście oparte na wielokrotnej redukcji obszaru poszukiwań o połowę (p. omówienie rozwiązań zadań „Czapki i rękawiczki” oraz „Skarbce Franka Cenciaka” dla Skrzatów). Wykonamy zatem przeszukiwanie binarne od 1 do $\min(w, h)$, dla każdej wartości sprawdzając, czy da się dla kwadratu o danym boku przejść po mapie i odpowiednio zawężając zakres poszukiwań. Jak można zauważyć, ostateczną złożoność obliczeniową naszego rozwiązania będzie wówczas można oszacować jako $O(wh \log(w + h))$.

[1] Mowa tu o popularnym narzędziu typu „kubełek z farbą” (ang. *bucket-fill*).

[2] W niektórych zastosowaniach tego algorytmu sprawdzamy nie czterech, a ośmiu sąsiadów.

46.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Pizza_picnic_picle/solution.py.

```
def daSiePrzejechac(mapa, rozmiar):
    h = len(mapa)
    w = len(mapa[0])
    for y in range(h):
        ostatniaPrzeszkoda = -1
        for x in range(w):
            if mapa[y][x]:
                ostatniaPrzeszkoda = x
            elif x - ostatniaPrzeszkoda < rozmiar:
                mapa[y][x] = True
    for x in range(w):
        ostatniaPrzeszkoda = -1
        for y in range(h):
            if mapa[y][x]:
                ostatniaPrzeszkoda = y
            elif y - ostatniaPrzeszkoda < rozmiar:
                mapa[y][x] = True
    if mapa[h - 2][rozmiar] or mapa[rozmiar][w - 2]:
        return False
    stos = [(rozmiar, h - 2)]
    mapa[h - 2][rozmiar] = True
    while len(stos) > 0:
        pozycja = stos.pop()
        x, y = pozycja
        if not mapa[y][x + 1]:
            stos.append((x + 1, y))
            mapa[y][x + 1] = True
        if not mapa[y][x - 1]:
            stos.append((x - 1, y))
            mapa[y][x - 1] = True
        if not mapa[y + 1][x]:
            stos.append((x, y + 1))
            mapa[y + 1][x] = True
        if not mapa[y - 1][x]:
            stos.append((x, y - 1))
            mapa[y - 1][x] = True
    return mapa[rozmiar][w - 2]

w, h = [int(x) for x in input().split()]
mapa = [(w + 2) * [True]]
for _ in range(h):
    mapa.append([True] + [c == 'X' for c in input()] + [True])
mapa.append((w + 2) * [True])
a = 1
b = min(w, h)
while a < b:
    srednia = (a + b + 1) // 2
    if daSiePrzejechac([[wartosc for wartosc in linia] for linia in mapa], srednia):
```

```
        a = srednia
    else:
        b = srednia - 1
print(a)
```

46.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Pizza_picnic_picle/solution.cpp.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

struct punkt {
  int x = 0;
  int y = 0;
};

bool da_sie_przejechac(std::vector<std::string> mapa, int rozmiar) {
  int h = mapa.size();
  int w = mapa[0].size();
  for (int y = 0; y < h; y++) {
    int ostatnia_przeszkoda = -1;
    for (int x = 0; x < w; x++) {
      if (mapa[y][x] == 'X')
        ostatnia_przeszkoda = x;
      else if (x - ostatnia_przeszkoda < rozmiar)
        mapa[y][x] = 'X';
    }
  }
  for (int x = 0; x < w; x++) {
    int ostatnia_przeszkoda = -1;
    for (int y = 0; y < h; y++) {
      if (mapa[y][x] == 'X')
        ostatnia_przeszkoda = y;
      else if (y - ostatnia_przeszkoda < rozmiar)
        mapa[y][x] = 'X';
    }
  }
  if (mapa[h - 1][rozmiar - 1] == 'X' || mapa[rozmiar - 1][w - 1] == 'X')
    return false;
  std::vector<punkt> stos {{rozmiar - 1, h - 1}};
  mapa[h - 1][rozmiar - 1] = 'X';
  while (!stos.empty()) {
    punkt pozycja = stos.back();
    stos.pop_back();
    if (pozycja.x + 1 < w && mapa[pozycja.y][pozycja.x + 1] != 'X') {
      stos.push_back({pozycja.x + 1, pozycja.y});
      mapa[pozycja.y][pozycja.x + 1] = 'X';
```

```
    }
    if (pozycja.x - 1 >= 0 && mapa[pozycja.y][pozycja.x - 1] != 'X') {
      stos.push_back({pozycja.x - 1, pozycja.y});
      mapa[pozycja.y][pozycja.x - 1] = 'X';
    }
    if (pozycja.y + 1 < h && mapa[pozycja.y + 1][pozycja.x] != 'X') {
      stos.push_back({pozycja.x, pozycja.y + 1});
      mapa[pozycja.y + 1][pozycja.x] = 'X';
    }
    if (pozycja.y - 1 >= 0 && mapa[pozycja.y - 1][pozycja.x] != 'X') {
      stos.push_back({pozycja.x, pozycja.y - 1});
      mapa[pozycja.y - 1][pozycja.x] = 'X';
    }
  }
  return mapa[rozmiar - 1][w - 1] == 'X';
}

int main() {
  int w, h;
  std::cin >> w >> h;
  std::vector<std::string> mapa(h);
  for (int i = 0; i < h; i++) {
    std::cin >> mapa[i];
  }
  int a = 1;
  int b = std::min(w, h);
  while (a < b) {
    int srednia = (a + b + 1) / 2;
    if (da_sie_przejechac(mapa, srednia))
      a = srednia;
    else
      b = srednia - 1;
  }
  std::cout << a;
  return 0;
}
```

47 (V) – Ogrodzenie – Gremliny

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (IV edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

47.1 Treść zadania

Pewien pasterz, szeroko znany wśród miejscowych jako Dziadek Fortran, potrzebuje nowych ogrodzeń dla swego stale rosnącego stada owiec. Jeszcze nie tak dawno zbudowałby je sam, ale – jako że starość nie radość, a pracy jest dużo – tym razem zdecydował się skorzystać z usług ekipy budowlanej.

Gdy ekipa się zjawiła, jej przedstawiciel miał nie lada trudności z wyciągnięciem od pasterza, jak te ogrodzenia mają właściwie wyglądać – ten, przyzwyczajony do wykonywania pracy sam, ewidentnie miał bardzo jasny pomysł na ich kształt, ale trudno mu było go przekazać. Wreszcie udało się poczynić następujące ustalenia:

- Dziadek Fortran ma wiele prostokątnych działek i na każdej z nich chce, by powstało dokładnie jedno ogrodzenie.
- Ogrodzenie powinno mieć kształt wielokąta, którego każdy bok jest równoległy albo do osi północ-południe albo osi wschód-zachód. Boki wielokąta nie mogą się przecinać.
- Na każdej działce pasterz wbił w ziemię drewniane pale. Każdy taki pal ma się stać wierzchołkiem wielokąta (czyli końcem dokładnie dwóch prostopadłych do siebie boków). Wielokąt nie może mieć wierzchołka w żadnym punkcie, w którym nie znajduje się pal.

Pasterz twierdzi, że na każdej działce da się przy takich założeniach jednoznacznie wyznaczyć kształt ogrodzenia.

Zadanie

Szef ekipy załamał ręce, gdy przekazano mu powyższe ustalenia. Ale cóż... klient może i ekscentryczny, jednak dobrze płaci, więc trzeba się dostosować. Szef ma jednak wątpliwości wobec tego ostatniego wspomnianego stwierdzenia staruszka – podejrzewa, że na niektórych działkach da się wybrać ogrodzenie na wiele sposobów albo nie da się tego zrobić wcale. Twoim zadaniem jest dla każdej działki zliczyć, na ile sposobów da się poprawnie wybrać ogrodzenie.

Opis wejścia

Pierwsza linia standardowego wejścia zawiera jedną liczbę naturalną T $(1 \leq T \leq 100)$, po której następuje T opisów działek. Każdy opis działki rozpoczyna się linią zawierającą trzy liczby naturalne S, W, N (gdzie $S, W \geq 2$; $4 \leq N \leq 200\,000$), oznaczające odpowiednio szerokość (rozpiętość wschód-zachód) i wysokość (rozpiętość północ-południe) działki, oraz liczbę wbitych pali. Każda i-ta z kolejnych N linii opisu zawiera dwie liczby naturalne x_i, y_i $(1 \leq x_i \leq S;\ 1 \leq y_i \leq W)$ – współrzędne i-tego pala (oś OX jest skierowana na wschód, OY – na północ). Podane pale uporządkowane są rosnąco wierszami, a wewnątrz każdego wiersza –

kolumnami. W każdym wierszu od 1 do W i w każdej kolumnie od 1 do S znajduje się przynajmniej jeden pal. Suma wartości N ze wszystkich działek nie przekracza 1 000 000.

Opis wyjścia

Dla każdej działki, na wyjście należy wypisać linię zawierającą jedną liczbę całkowitą – liczbę sposobów, na które na danej działce da się wybrać ogrodzenie spełniające podane wymagania. Oczekiwana odpowiedź nie przekracza 10^{18} dla żadnej działki.

Przykład

Dla przykładowego wejścia:

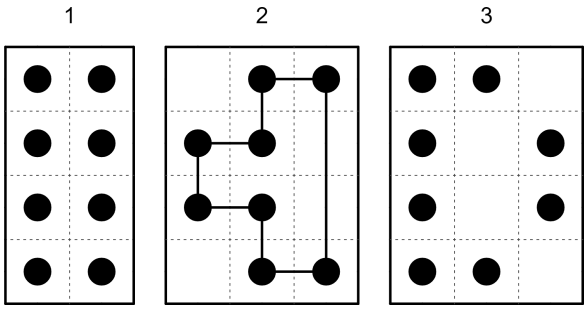
```
3
2 4 8
1 1
2 1
1 2
2 2
1 3
2 3
1 4
2 4
3 4 8
2 1
3 1
1 2
2 2
1 3
2 3
2 4
3 4
3 4 8
1 1
2 1
1 2
3 2
1 3
3 3
1 4
2 4
```

prawidłowym wyjściem jest:

```
0
1
0
```

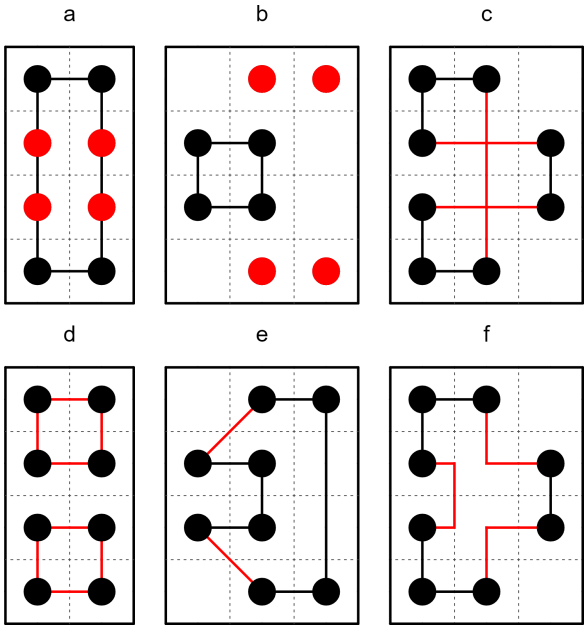
Wyjaśnienie

Tylko na drugiej działce da się skonstruować ogrodzenie i da się je wybrać tylko w jeden sposób. Poniżej zilustrowano wszystkie 3 działki wraz z jedynym możliwym ogrodzeniem.



Rysunek 47.1: Ilustracja przykładu

Dodatkowo, poniżej zamieszczono przykłady ogrodzeń, które z różnych przyczyn nie spełniają zadanych warunków i dlatego nie zostały zliczone, z błędami zaznaczonymi na czerwono.



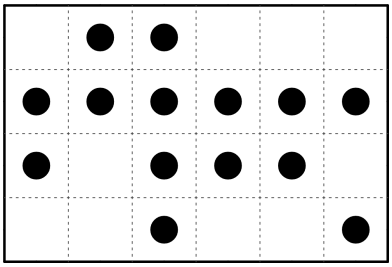
Rysunek 47.2: Ilustracje błędnych ogrodzeń

a. Niektóre pale nie znajdują się w wierzchołkach wielokąta, tylko na jego bokach.
b. Niektóre pale w ogóle nie należą do wielokąta.
c. Boki wielokąta przecinają się.
d. Jest więcej niż jedno ogrodzenie.
e. Niektóre boki nie są równoległe do osi północ-południe ani wschód-zachód.
f. Niektóre wierzchołki wielokąta nie zawierają pali.

47.2 Rozwiązanie

W zadaniu należało dla podanego zbioru punktów („pali”) zliczyć liczbę możliwych wielokątów („ogrodzeń”) takich, że każdy bok wielokąta jest równoległy do jednej z osi układu współrzędnych, zbiór wierzchołków tego wielokąta jest zadanym zbiorem punktów oraz jego boki nie przecinają się.

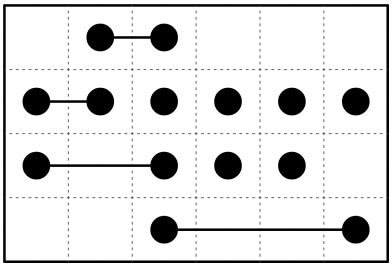
Załóżmy, że dla pewnego zbioru punktów taki wielokąt istnieje. Do demonstracji wykorzystamy poniższy przykładowy zbiór:



Rysunek 47.3: Przykładowy zbiór punktów

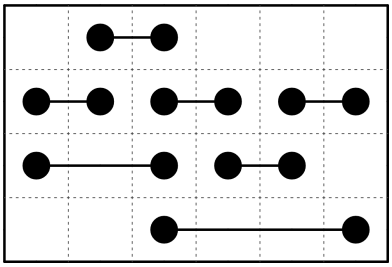
Ponieważ każdy zadany punkt musi się znajdować w wierzchołku, tj. punkcie wspólnym dwóch boków, a każdy bok jest równoległy do jednej z osi układu, to w każdej takiej parze boków jeden musi być poziomy a drugi – pionowy.

Zauważmy, że jeśli dany punkt w swoim wierszu jest pierwszy od lewej, to wychodzący z niego bok poziomy musi być skierowany w prawo i trafiać do punktu drugiego od lewej. Narysujmy te boki w naszym przykładzie.



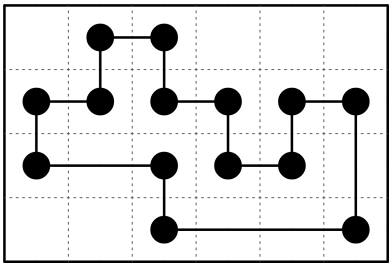
Rysunek 47.4: Przykładowy zbiór punktów z wybranymi bokami poziomymi

W takim razie, jeżeli punkt jest w swoim wierszu trzeci od lewej, to wychodzący z niego bok poziomy również musi być skierowany w prawo. Gdyby zamiast tego był skierowany w lewo, to do drugiego punktu w wierszu trafiałyby dwa boki poziome, co jest niemożliwe. W konsekwencji, bok poziomy wychodzący z punktu trzeciego musi trafiać do czwartego, a piąty musi znowu być skierowany w prawo i trafiać do szóstego, itd. W rezultacie dla każdego punktu potrafimy łatwo wskazać punkt, z którym jest on połączony bokiem poziomym.



Rysunek 47.5: Przykładowy zbiór punktów ze wszystkimi bokami poziomymi

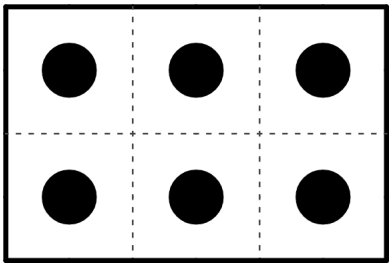
Analogiczną dedukcję możemy wykonać na kolumnach i bokach pionowych. Dla każdego punktu ze zbioru poznajemy wówczas zarówno bok poziomy i bok pionowy, które się w nim spotykają. Ponieważ każdy bok wielokąta zaczyna się i kończy w jednym z zadanych punktów, to taka procedura pozwoli nam ustalić wszystkie boki wielokąta.



Rysunek 47.6: Wielokąt otrzymany dla przykładowego zbioru

Wynika z tego jasno, że dla każdego zbioru punktów istnieje co najwyżej jeden wielokąt spełniający warunki zadania i w dodatku jest on prosty do wyznaczenia. Rozwiązanie zadania sprowadza się zatem do zweryfikowania, czy otrzymany zbiór boków faktycznie spełnia warunki zadania.

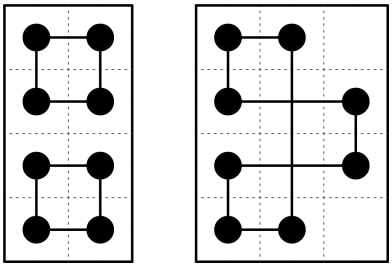
Zanim przejdziemy dalej, należy tu zwrócić uwagę na niejawne założenie w powyższej procedurze – zakłada ona, że zawsze uda nam się sparować punkty w każdym wierszu i kolumnie, co nie stanie się, jeżeli liczba punktów w danym wierszu lub kolumnie jest nieparzysta, jak na rysunku poniżej.



Rysunek 47.7: Przykład zbioru punktów o nieparzystych liczbach punktów w wierszach

Nietrudno się przekonać, że wówczas wielokąt nie może istnieć – któryś z punktów wierszu musiałby albo nie być połączony bokiem poziomym albo być połączony dwoma, a żadna z tych sytuacji nie jest dozwolona.

Aby zweryfikować poprawność otrzymanego zbioru boków, musimy sprawdzić dwa warunki: po pierwsze, upewnić się, że tworzą one jeden wielokąt – a nie wiele – a po drugie, że boki nie przecinają się. Nasza dotychczasowa procedura może skutkować wynikami nie spełniającymi jednego lub obu z tych warunków, np. w przykładach podanych w treści zadania.



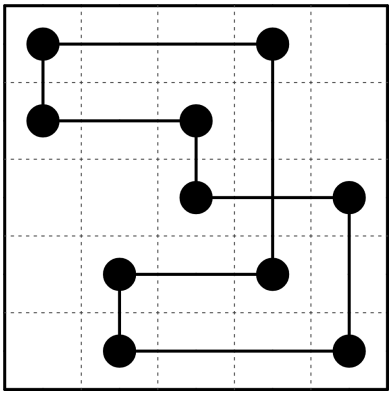
Rysunek 47.8: Przykładowe wyniki nie spełniające warunków zadania

By sprawdzić, czy nasz wynik składa się z jednego wielokąta, wystarczy wybrać dowolny wierzchołek i przejść na zmianę bokami pionowymi i poziomymi aż do trafienia z powrotem do tego wierzchołka. Jeżeli wykonamy

tyle kroków, ile jest wierzchołków, to wszystkie wierzchołki należą do jednego wielokąta, więc warunek jest spełniony. Można też zastosować dowolne podejście służące do sprawdzania spójności grafu (p. dział VIII).

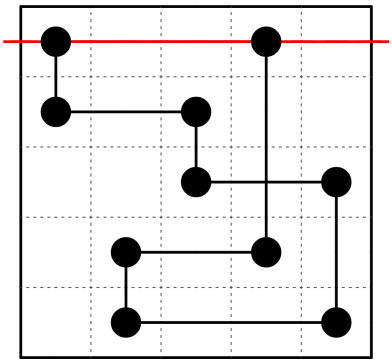
Aby sprawdzić, czy istnieje jakakolwiek para przecinających się boków, możemy przetestować każdą możliwą parę prostopadłych boków. Takie podejście posiada jednak złożoność obliczeniową $O(n^2)$. By zrobić to szybciej, możemy wykorzystać – znaną nam już z działu IV (p. zadania: „Omlety” i „Vururuk”) – technikę zamiatania płaszczyzny [4] (ang. *plane sweep algorithm*), która pozwoli nam osiągnąć złożoność $O(n \log n)$.

Sposób wykorzystania tej techniki omówimy na poniższym przykładzie:



Rysunek 47.9: Przykład do demonstracji techniki zamiatania płaszczyzny

Wyobraźmy sobie prostą poziomą („miotłę”) poruszającą się po płaszczyźnie z góry na dół (na poniższym rysunku zaznaczymy tę prostą kolorem czerwonym). Z punktu widzenia implementacji, jest to oczywiście nic innego jak pętla przechodząca po wszystkich punktach uporządkowanych według współrzędnej y.



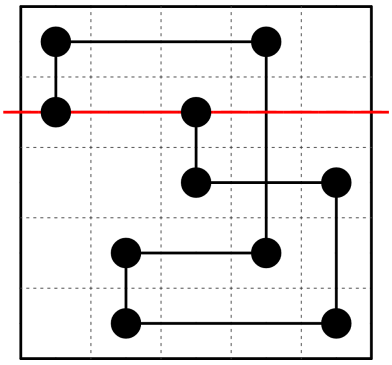
Rysunek 47.10: Przykład do demonstracji techniki zamiatania płaszczyzny z zaznaczoną prostą poziomą w pierwszym wierszu

Wyróżniamy trzy istotne rodzaje zdarzeń, jakie mogą mieć miejsce:

- prosta napotyka górny koniec odcinka pionowego – wówczas chcemy zapamiętać, że w danej kolumnie znajduje się obecnie odcinek;
- prosta napotyka dolny koniec odcinka pionowego – wówczas chcemy „zapomnieć” o owym odcinku;
- prosta napotyka odcinek poziomy – wówczas chcemy sprawdzić, czy którykolwiek ze spamiętanych odcinków pionowych się z nim przecina.

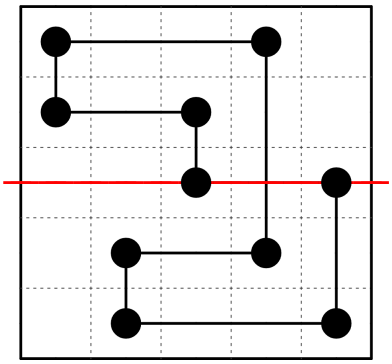
Kolejność przetwarzania zdarzeń wewnątrz jednego wiersza nie powinna mieć dla nas znaczenia. W pierwszym wierszu zapamiętujemy zatem, że w kolumnach 1 i 4 znajdują się odcinki pionowe, a także sprawdzamy dla odcinka poziomego, czy krzyżuje się z jakimkolwiek zapamiętanym odcinkiem poziomym – czyli czy

pamiętamy jakąkolwiek liczbę większą od 1 a mniejszą od 4. Nie pamiętamy, więc przechodzimy do kolejnego wiersza.



Rysunek 47.11: Przykład z prostą w drugim wierszu

W drugim wierszu usuwamy z pamięci odcinek w kolumnie 1, dodajemy odcinek w kolumnie 3, i sprawdzamy, czy pamiętamy liczbę większą od 1 a mniejszą od 3. Obecnie w pamięci przechowujemy liczby 3 i 4, więc odpowiedź znów brzmi „nie", czyli odcinki nie przecinają się.



Rysunek 47.12: Przykład z prostą w trzecim wierszu

W trzecim wierszu usuwamy z pamięci liczbę 3, zapamiętujemy liczbę 5, i sprawdzamy, czy pamiętamy liczbę większą od 3 a mniejszą od 5. Tym razem jest to prawda – pamiętamy bowiem o odcinku w kolumnie 4 – czyli wykryliśmy przecięcie dwóch odcinków i możemy wcześnie zakończyć działanie algorytmu.

Aby algorytm rzeczywiście posiadał zakładaną złożoność, do zapamiętywania liczb musimy wykorzystać strukturę, która pozwala nam na szybkie dodawanie elementów, ich usuwanie, oraz sprawdzanie, czy istnieją elementy zawarte w danym zakresie liczbowym. W C++ taką strukturą jest `set` z biblioteki standardowej, o ile znamy jego funkcje składowe `lower_bound` i `upper_bound`. W języku Python żadna taka struktura nie jest niestety dostępna z góry. Wydajne i szybkie w implementacji okazuje się wówczas np. drzewo przedziałowe (p. dział III, zadanie „Kto stoi przede mną").

47.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Ogrodzenie/solution.py.

```
from dataclasses import dataclass
from itertools import chain

@dataclass
class punkt:
    x: int = 0
    y: int = 0
    indeks_w_wierszu: int = 0
    indeks_w_kolumnie: int = 0

class drzewo_przedzialowe:
    def __init__(self, rozmiar: int):
        self._tablica = [0] * (2 << (rozmiar - 1).bit_length())

    def rozmiar(self):
        return len(self._tablica) // 2

    def ustaw(self, indeks: int, wartosc: int):
        i = indeks + self.rozmiar()
        roznica = wartosc - self._tablica[i]
        while i != 0:
            self._tablica[i] += roznica
            i >>= 1

    def wartosc(self, indeks: int):
        return self._tablica[indeks + self.rozmiar()]

    def suma(self, lewo: int, prawo: int):
        lewo += self.rozmiar()
        prawo += self.rozmiar()
        wynik = 0
        while lewo != prawo:
            if lewo & 1:
                wynik += self._tablica[lewo]
                lewo += 1
            if prawo & 1:
                wynik += self._tablica[prawo - 1]
            lewo >>= 1
            prawo >>= 1
        return wynik

def rozwiaz():
    w, h, n = [int(x) for x in input().split()]
    pale = []
    pale_wg_wiersza = [[] for _ in range(w)]
    pale_wg_kolumny = [[] for _ in range(h)]
    for i in range(n):
        p = punkt()
```

```
        p.x, p.y = [int(x) for x in input().split()]
        p.x -= 1
        p.y -= 1
        p.indeks_w_wierszu = len(pale_wg_wiersza[p.x])
        p.indeks_w_kolumnie = len(pale_wg_kolumny[p.y])
        pale.append(p)
        pale_wg_wiersza[p.x].append(i)
        pale_wg_kolumny[p.y].append(i)
    if any(len(a) % 2 != 0 for a in chain(pale_wg_wiersza, pale_wg_kolumny)):
        return 0
    indeks = 0
    odwiedzone = 0
    while True:
        indeks = pale_wg_wiersza[pale[indeks].x][pale[indeks].indeks_w_wierszu ^ 1]
        indeks = pale_wg_kolumny[pale[indeks].y][pale[indeks].indeks_w_kolumnie ^ 1]
        odwiedzone += 2
        if indeks == 0:
            break
    if odwiedzone != n:
        return 0
    otwarcie_linii_poziomej = -1
    otwarcia_linii_pionowych = drzewo_przedzialowe(w)
    for p in pale:
        if otwarcie_linii_poziomej < 0:
            otwarcie_linii_poziomej = p.x
        else:
            if otwarcia_linii_pionowych.suma(otwarcie_linii_poziomej + 1, p.x) != 0:
                return 0
            otwarcie_linii_poziomej = -1
        otwarcia_linii_pionowych.ustaw(p.x, otwarcia_linii_pionowych.wartosc(p.x) ^ 1)
    return 1

t = int(input())
for _ in range(t):
    print(rozwiaz())
```

47.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Ogrodzenie/solution.cpp.

```cpp
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>

struct punkt {
  int x = 0;
  int y = 0;
  int indeks_w_wierszu = 0;
  int indeks_w_kolumnie = 0;
};

int rozwiaz() {
  int w, h, n;
  std::cin >> w >> h >> n;
  std::vector<punkt> pale(n);
  std::vector<std::vector<int>> pale_wg_wiersza(w);
  std::vector<std::vector<int>> pale_wg_kolumny(h);
  for (int i = 0; i < n; i++) {
    punkt p;
    std::cin >> p.x >> p.y;
    p.x--;
    p.y--;
    p.indeks_w_wierszu = pale_wg_wiersza[p.x].size();
    p.indeks_w_kolumnie = pale_wg_kolumny[p.y].size();
    pale[i] = p;
    pale_wg_wiersza[p.x].push_back(i);
    pale_wg_kolumny[p.y].push_back(i);
  }
  auto is_odd_size = [](const auto& arg) {
    return std::size(arg) % 2 != 0;
  };
  if (std::any_of(pale_wg_wiersza.begin(), pale_wg_wiersza.end(), is_odd_size))
    return 0;
  if (std::any_of(pale_wg_kolumny.begin(), pale_wg_kolumny.end(), is_odd_size))
    return 0;
  int indeks = 0;
  int odwiedzone = 0;
  do {
    indeks = pale_wg_wiersza[pale[indeks].x][pale[indeks].indeks_w_wierszu ^ 1];
    indeks = pale_wg_kolumny[pale[indeks].y][pale[indeks].indeks_w_kolumnie ^ 1];
    odwiedzone += 2;
  } while (indeks != 0);
  if (odwiedzone != n)
    return 0;
  int otwarcie_linii_poziomej = -1;
  std::set<int> otwarcia_linii_pionowych;
  for (const auto& p : pale) {
```

```
    if (otwarcie_linii_poziomej < 0) {
      otwarcie_linii_poziomej = p.x;
    } else {
      auto it = otwarcia_linii_pionowych.upper_bound(otwarcie_linii_poziomej);
      if (it != otwarcia_linii_pionowych.end() && *it < p.x)
        return 0;
      otwarcie_linii_poziomej = -1;
    }
    auto it = otwarcia_linii_pionowych.find(p.x);
    if (it != otwarcia_linii_pionowych.end())
      otwarcia_linii_pionowych.erase(it);
    else
      otwarcia_linii_pionowych.insert(it, p.x);
  }
  return 1;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int t;
  std::cin >> t;
  for (int i = 0; i < t; i++) {
    std::cout << rozwiaz() << '\n';
  }
  return 0;
}
```

48 (V) – Smocze skarby – Gremliny

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (III edycja – liga algorytmiczna)
Język programowania: C++/Python

48.1 Treść zadania

Za siedmioma górami i siedmioma lasami leży bajkowe królestwo Kwantopia – albo raczej leżało, gdyż w dwudziestym pierwszym wieku Kwantopia jest już nie tyle królestwem, co rozwiniętym krajem demokratycznym, z siedmiu lasów wskutek niepohamowanej wycinki drzew pozostały tylko trzy, a niegdyś nękające krainę smoki wyginęły stulecia temu. Od czasu do czasu wciąż zdarza się jednak odkryć niespodziewane pozostałości z przeszłości, przypominające o tych wielkich gadach.

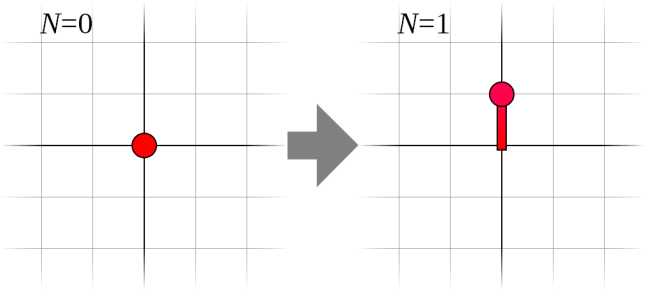
Niedawno archeologowie z Kwantopii odkryli prastary manuskrypt, traktujący o dawno zapomnianej smoczej jamie. Owa jama miałaby zawierać srebro, złoto i kamienie szlachetne zgromadzone za życia przez pewnego szczególnie chciwego smoka – stąd trafiła natychmiast na celownik poszukiwaczy skarbów. Manuskrypt opisuje nawet drogę do jamy. Jest tylko jeden szkopuł – droga ta jest długa i zawiła, przez co łatwo pomylić kierunki, i dlatego wciąż nikomu nie udało się znaleźć smoczego skarbca.

Zadanie

Stań się pierwszym odkrywcą jamy. W tym celu napisz program, który znajdzie jej współrzędne na mapie.

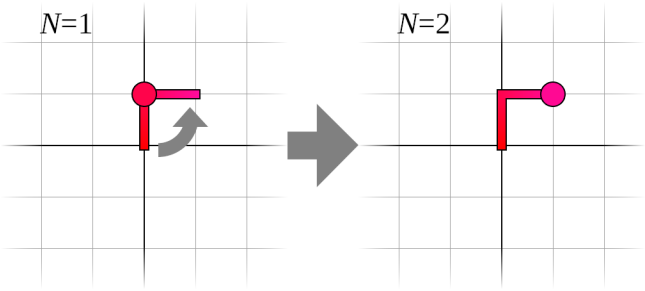
Według manuskryptu, drogę do jamy można wyznaczyć w następujący sposób:

Zacznij w punkcie $(0, 0)$ i wykonaj pierwszy krok na północ, do punktu $(0, 1)$.



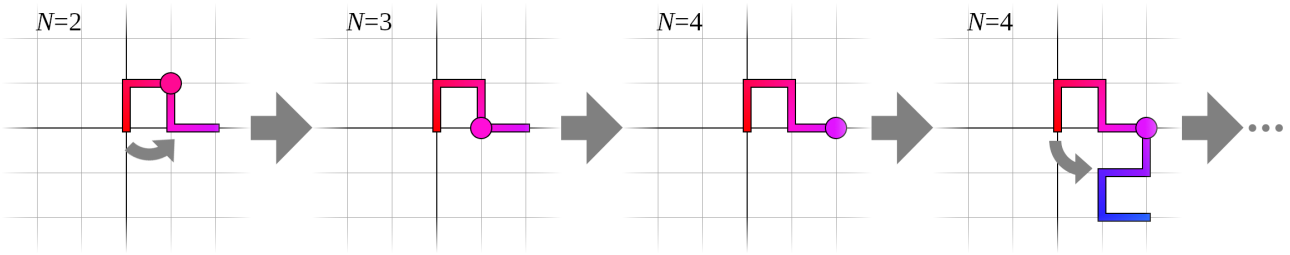
Rysunek 48.1: Krok 1

Następnie weź całą przebytą dotychczas drogę i obróć ją względem punktu, w którym się obecnie znajdujesz, o 90° przeciwnie do ruchu wskazówek zegara. Poruszaj się pojedynczymi krokami wzdłuż tej nowej drogi. Drugi krok prowadzi zatem do punktu $(1, 1)$.



Rysunek 48.2: Krok 2

Podobnie postąp, gdy dotrzesz do końca nowej drogi, a potem kolejnej, itd., aż wykonasz łącznie N pojedynczych kroków. Wówczas znajdziesz się u progu smoczej jamy.



Rysunek 48.3: Dalsze kroki

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba naturalna T ($1 \leq T \leq 100\,000$) oznaczająca liczbę niezależnych przypadków testowych do rozpatrzenia. Każdy i-ty z kolejnych T wierszy zawiera jedną liczbę całkowitą N_i ($0 \leq N_i \leq 10^{18}$) – liczbę kroków do wykonania w danym przypadku.

Opis wyjścia

Na standardowe wyjście należy wypisać T wierszy, każdy zawierający dwie liczby oddzielone spacją – w i-tym wierszu powinny znaleźć się współrzędne (x_i, y_i) smoczej jamy w i-tym przypadku testowym.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
4
3
0
30
8
```

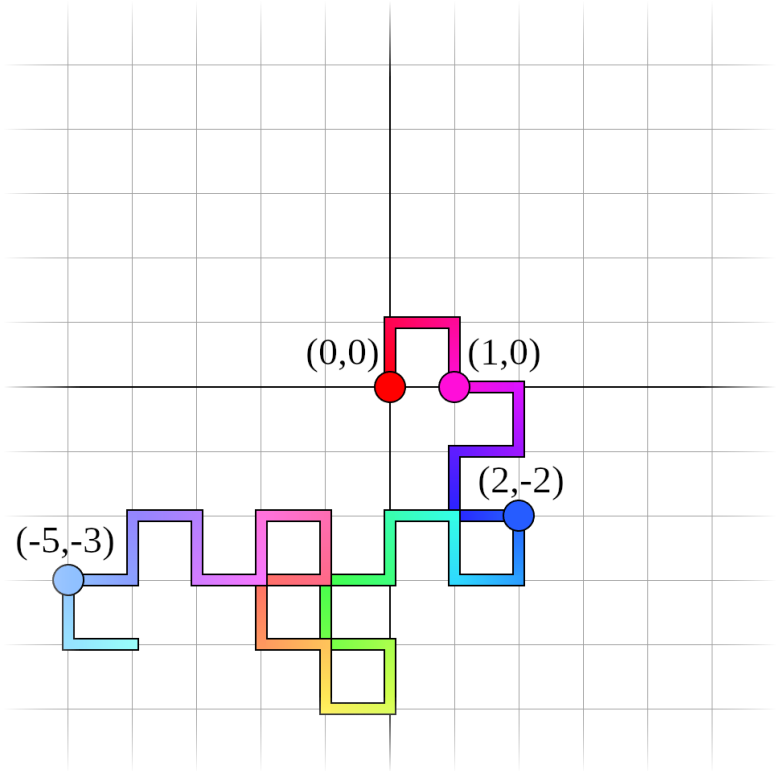
prawidłową odpowiedzią jest:

```
1 0
```

```
0 0
-5 -3
2 -2
```

Wyjaśnienie

Na poniższej ilustracji ukazana jest droga do smoczej jamy wraz z punktami, w których jama znajdowałaby się w każdym przypadku testowym.



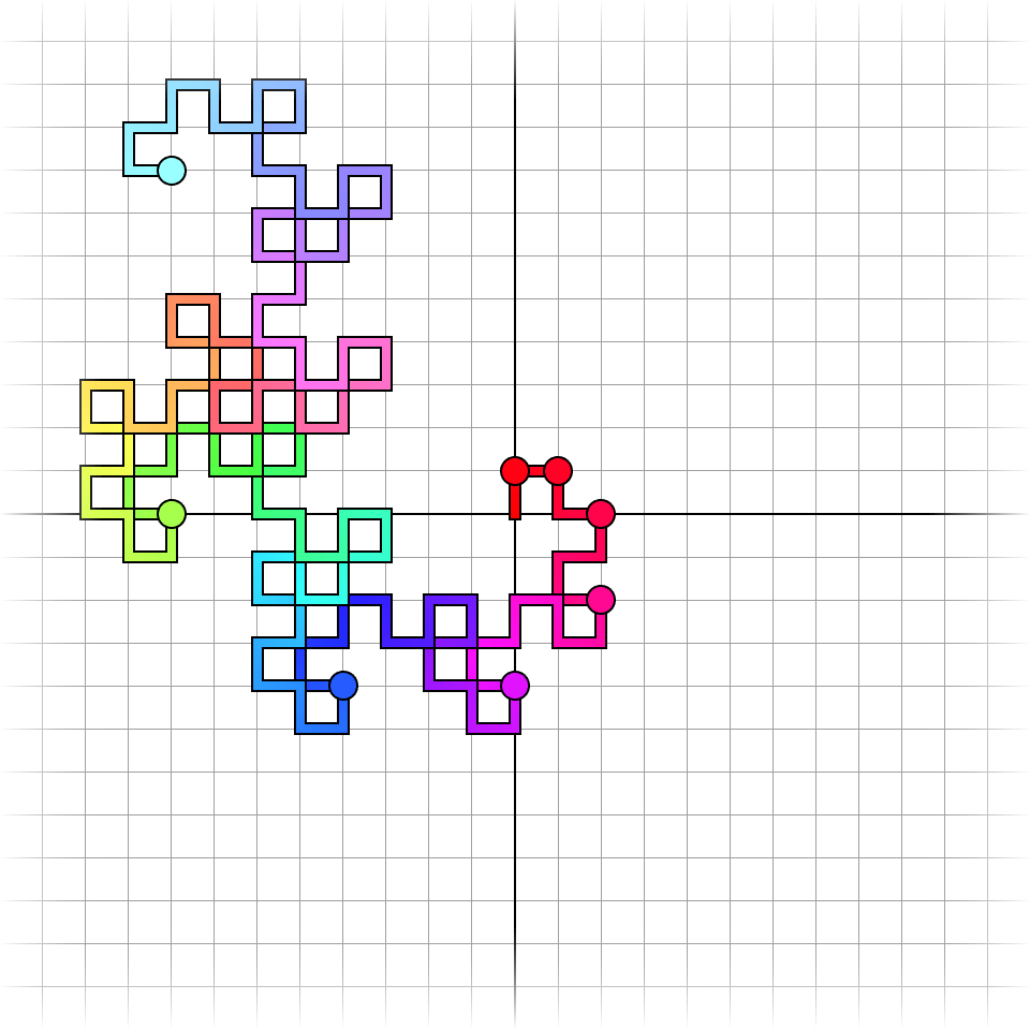
Rysunek 48.4: Ilustracja przykładu

48.2 Rozwiązanie

W zadaniu opisana jest pewna ścieżka – w rzeczywistości jest to fraktal znany m.in. pod nazwą *smok Heighwaya*. Celem zadania jest napisać program, który będzie w stanie szybko odpowiadać na zapytania, w jakim punkcie znajdziemy się po przejściu n kroków wzdłuż ścieżki zaczynając z punktu $(0, 0)$.

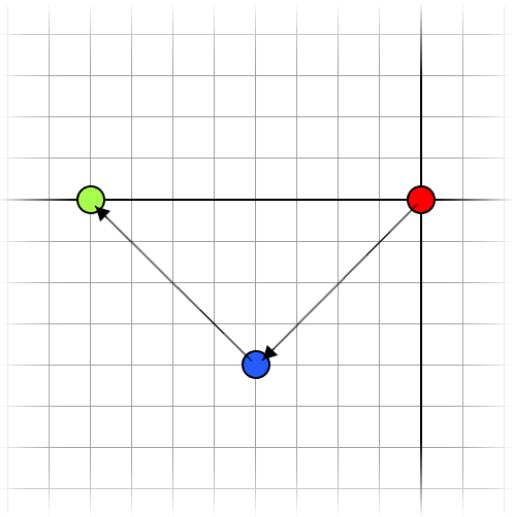
Na pierwszy rzut oka ścieżka może wydać się chaotyczna, ale ponieważ rozbudowujemy ją za każdym razem poprzez skopiowanie i obrót dotychczasowej ścieżki, to jej kształt zyskuje pewną powtarzalność, którą wykorzystamy do rozwiązania problemu.

Zacznijmy od spostrzeżenia, że przy każdej rozbudowie ścieżki wydłużamy ją dwukrotnie – zatem jej długości są kolejnymi potęgami liczby 2. Można się domyślać, że w takim razie dla n będących potęgami dwójki jest też najłatwiej znaleźć odpowiedź, i rzeczywiście tak jest.



Rysunek 48.5: Ścieżka o długości 128 z zaznaczonymi wszystkimi potęgami dwójki od 1 do 128. Zauważ regularność w ich rozmieszczeniu

Jeżeli znamy pozycję punktu odpowiadającego 2^k krokom dla pewnego k, to łatwo jest obliczyć pozycję punktu dla 2^{k+1} – wystarczy do wektora odpowiadającego tej pierwszej pozycji dodać ten sam wektor obrócony o $90°$ w prawo.

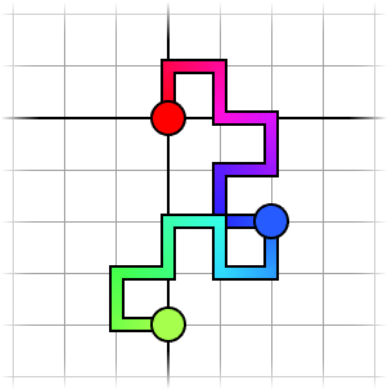


Rysunek 48.6: Opisana metoda dla punktów odpowiadających $n = 32$ i $n = 64$. Kąt między wektorami jest prosty, a ich suma daje pozycję punktu dla $n = 64$

Skoro tak, to możemy szybko znaleźć odpowiedź dla dowolnej potęgi liczby 2. Pozycje te przydadzą nam się w dalszej części programu, więc może być warto policzyć je tylko raz na początku programu i zapisać w tablicy, np. dla wszystkich potęg dwójki od 2^0 do 2^{60}.

Przejdźmy teraz do omówienia najistotniejszej części algorytmu. Powiedzmy, że otrzymamy zapytanie o pozycję punktu dla pewnego n. Jeżeli n jest potęgą dwójki, to wiemy już, jak odpowiedzieć na to zapytanie, więc załóżmy, że tak nie jest. Znajdźmy jak najmniejszą potęgę dwójki 2^p większą od n. Ponieważ jest ona większa, to szukany punkt leży gdzieś na ścieżce od 0 do 2^p. Co więcej, ponieważ p wybraliśmy tak, że $n > 2^{p-1}$, to znajduje się on bliżej 2^p niż 0, idąc wzdłuż ścieżki (2^{p-1} to inaczej $\frac{2^p}{2}$, czyli połowa drogi między tymi punktami – wiemy, że punkt jest dalej niż w połowie, więc musi być bliżej końca niż początku).

Ze względu na sposób, w jaki konstruujemy ścieżkę, droga „wstecz” od 2^p do 2^{p-1} jest obróconą kopią drogi od 0 do 2^{p-1}.



Rysunek 48.7: Ścieżka o długości 16 z zaznaczonymi punktami $0, 8, 16$. Zwróć uwagę, że droga od 0 do 8 wykonuje takie same zakręty, jak od 16 do 8 – różni się tylko początkowym kierunkiem podróży

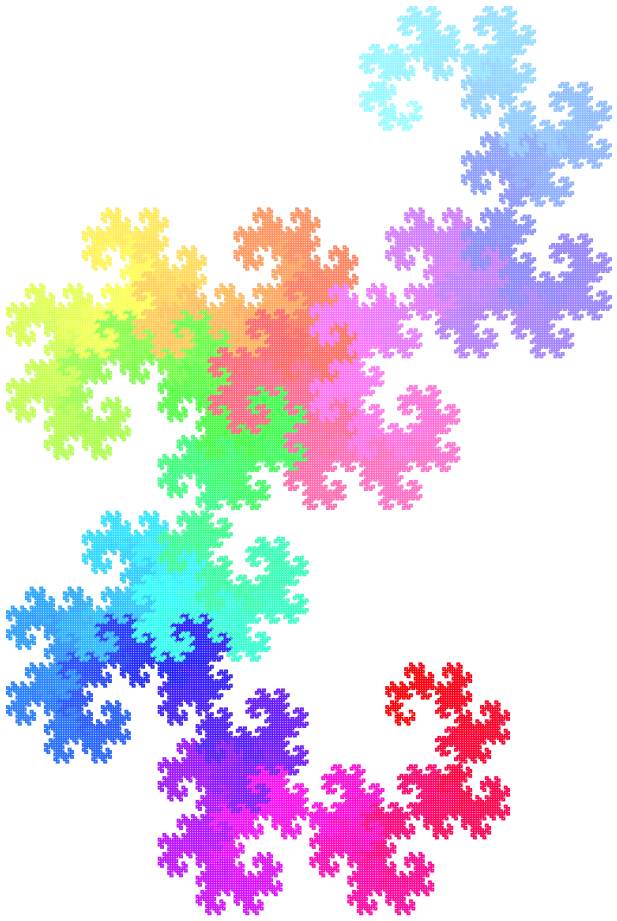
Skoro tak, to zamiast obliczać, na jakiej pozycji znajdziemy się wychodząc z $(0, 0)$ i wykonując n kroków, możemy obliczyć pozycję po wyjściu z pozycji odpowiadającej 2^p i wykonaniu $2^p - n$ kroków w obróconym kierunku – ale wzdłuż takiej samej drogi.

Daje nam to rekurencyjny sposób na rozwiązanie zadania. Co istotne, jest to sposób niezwykle szybki, bowiem

$2^p - n$ jest liczbą na pewno mniejszą od 2^{p-1} – czyli na następnym poziomie rekurencji wartość p na pewno będzie mniejsza niż na obecnym, a początkowo p nie przekracza 60. W istocie, taki algorytm, dobrze zaimplementowany, działa w czasie $O(\log n)$.

Bez wnikania w szczegóły implementacyjne, poniżej znajduje się możliwy pseudokod takiej funkcji rekurencyjnej. Zamieszczona w nim operacja dodawania jest dodawaniem wektorów (w znaczeniu matematycznym, czyli w tym wypadku po prostu par liczb (x, y)), zaś << jest operacją przesunięcia bitowego w lewo – podobnie jak inne operacje bitowe, często bardzo pomocną przy pracy z potęgami dwójki.

```
define f(n) {
  if (n == 0):
    return wektor(0, 0)
  p = sufit_z_log2(n)  // warto zaimplementować mądrzej niż int(ceil(log2(n)))
  dwa_do_potegi_p = 1 << p
  return wyniki_dla_poteg_2[p] + obroc_w_lewo(f(dwa_do_potegi_p - n))
}
```



Rysunek 48.8: Ścieżka dla $n = 2^{16} = 65536$

48.2.1 Python

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Smocze_skarby/solution.py.

```python
def dodaj(a, b):
    return (a[0] + b[0], a[1] + b[1])

def obroc_w_lewo(v):
    return (-v[1], v[0])

def rozwiaz(n):
    if n == 0:
        return (0, 0)
    wykladnik = (n - 1).bit_length()
    return dodaj(przesuniecia_dla_poteg_2[wykladnik], obroc_w_lewo(rozwiaz((1 << wykladnik) - n)))

przesuniecia_dla_poteg_2 = [0] * 64
def stablicuj_przesuniecia():
    przesuniecie_x, przesuniecie_y = 0, 1
    for i in range(64):
        przesuniecia_dla_poteg_2[i] = (przesuniecie_x, przesuniecie_y)
        przesuniecie_x, przesuniecie_y = przesuniecie_x + przesuniecie_y, \
            przesuniecie_y - przesuniecie_x

stablicuj_przesuniecia()
t = int(input())
for _ in range(t):
    print(*rozwiaz(int(input())))
```

48.2.2 C++

Źródło: ./zadania/05_Dwuwymiarowe_struktury/Smocze_skarby/solution.cpp.

```cpp
#include <iostream>
struct wektor {
  long long x = 0;
  long long y = 0;
};
wektor operator+(const wektor& lhs, const wektor& rhs) {
  return {lhs.x + rhs.x, lhs.y + rhs.y};
}
wektor& operator+=(wektor& lhs, const wektor& rhs) {
  return lhs = lhs + rhs;
}
wektor obroc_w_prawo(const wektor& v) {
  return {v.y, -v.x};
}
wektor obroc_w_lewo(const wektor& v) {
  return {-v.y, v.x};
}
wektor przesuniecia_dla_poteg_2[64];
void stablicuj_przesuniecia() {
  wektor przesuniecie {0, 1};
  for (int i = 0; i < 64; i++) {
    przesuniecia_dla_poteg_2[i] = przesuniecie;
    przesuniecie += obroc_w_prawo(przesuniecie);
  }
}
wektor rozwiaz(unsigned long long n, unsigned wykladnik = 63) {
  if (n == 0)
    return {};
  while (((n << 1) - 1) >> wykladnik == 0) {
    wykladnik--;
  }
  return przesuniecia_dla_poteg_2[wykladnik]
    + obroc_w_lewo(rozwiaz((1uLL << wykladnik) - n, wykladnik - 1));
}
int main() {
  stablicuj_przesuniecia();
  int t;
  std::cin >> t;
  for (int i = 0; i < t; i++) {
    unsigned long long n;
    std::cin >> n;
    wektor p = rozwiaz(n);
    std::cout << p.x << ' ' << p.y << '\n';
  }
  return 0;
}
```

Centrum Mistrzostwa Informatycznego

Dział VI

Zadania obliczeniowe, elementy teorii liczb, podzielność, algorytm Euklidesa

49 (VI) – Wprowadzenie

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie

Elementy teorii liczb całkowitych zajmują poczesne miejsce w problemach, jakie możemy napotkać podczas rozmaitych konkursów i zawodów algorytmicznych. Arytmetyka liczb całkowitych ma w sobie coś szczególnego: rządzi się ścisłymi prawami matematycznymi i logicznymi i nie jest „skażona" niedokładnościami obliczeń nieodłącznych w komputerowych działaniach na liczbach zmiennoprzecinkowych.

Do najczęściej spotykanych zagadnień należą: znajdowanie dzielników liczb naturalnych (zwłaszcza największego wspólnego dzielnika) oraz znajdowanie liczb pierwszych.

Algorytm Euklidesa

Już w szkole podstawowej uczniowie poznają najprostszą wersję algorytmu Euklidesa znajdowania największego wspólnego dzielnika (NWD) [4][17][5] dwóch liczb naturalnych (a, b) opierającą się na stwierdzeniu, że różnica między a i b posiada taki sam dzielnik jak rozważane liczby:

1. Jeśli liczby a oraz b są równe sobie, wtedy ich NWD jest równe którejkolwiek z nich;
2. Jeśli $a > b$, wtedy za a podstawiamy wartość $a - b$. W przeciwnym przypadku za b podstawiamy wartość $b - a$;
3. Wracamy do kroku 1.

Metoda jest prosta, ale mało efektywna, zwłaszcza w sytuacji, gdy liczby a, b są względnie pierwsze (tzn. gdy $\mathrm{NWD} = 1$) – wówczas posiada ona złożoność liniową. Można jednak zauważyć, że reszta z dzielenia a przez b (czyli $a \bmod b$) posiada również taki sam wspólny dzielnik, choć trzeba tu dodatkowo zabezpieczyć się przed sytuacją, gdy $b = 0$ (w takim przypadku za NWD przyjmujemy a):

1. Jeśli $b = 0$, wtedy NWD jest równe a;
2. Za a podstawiamy b, natomiast za b podstawiamy $a \bmod b$ (wykorzystując wartość a sprzed podstawienia);
3. Wracamy do kroku 1.

Tak sformułowany algorytm ma złożoność logarytmiczną, a więc jest znacznie szybszy.

Istnieje jeszcze efektywniejszy algorytm (zwany algorytmem Steina), który bazuje na obserwacji, że dla liczb parzystych a, b mamy:

$$\mathrm{NWD}(a, b) = 2 \cdot \mathrm{NWD}\left(\frac{a}{2}, \frac{b}{2}\right) .$$

Natomiast jeśli a jest parzyste, zaś b nieparzyste, wtedy mamy $\mathrm{NWD}(a, b) = \mathrm{NWD}(a/2, b)$ i tak dalej. Jeśli obydwie liczby są nieparzyste, wtedy większą z nich zastępujemy różnicą $|a - b|$, która jest już parzysta. Badanie parzystości i dzielenie przez 2 to elementarne operacje procesora, stąd duża efektywność tej metody.

Znając wartość największego wspólnego dzielnika liczb (a, b) możemy łatwo obliczyć najmniejszą wspólną wielokrotność (NWW) tych liczb, korzystając z tożsamości:

$$\mathrm{NWW}(a, b) = \frac{a \cdot b}{\mathrm{NWD}(a, b)} .$$

Faktoryzacja liczby naturalnej

Faktoryzacja to nic innego, jak rozkład liczby na czynniki – z reguły są to czynniki pierwsze [4]. Analiza takiego rozkładu pozwala zwykle wysnuć określone wnioski dotyczące własności rozkładanej liczby. Na przykład można stwierdzić, że liczba jest pełnym kwadratem innej liczby naturalnej lub jest liczbą pierwszą. Można także przy pomocy faktoryzacji znajdować największy wspólny dzielnik (lub najmniejszą wspólną wielokrotność) dwóch lub więcej liczb.

Rozkład na iloczyn czynników pierwszych można zrealizować bez sprawdzania pierwszości podzielników. Wystarczy zacząć od najmniejszych podzielników i – posuwając się w kierunku coraz większych liczb – eliminować z rozkładanej liczby znalezione podzielniki. W ten sposób wszelkie potencjalne podzielniki będą liczbami pierwszymi. Zaczynamy zatem od najmniejszego nietrywialnego podzielnika, czyli liczby 2, a następnie sprawdzamy kolejne możliwe podzielniki: 3, 4, 5, ... Liczba 4 nie jest pierwsza, ale jeśli rozkładana liczba dzieliła się przez 2 (lub jakąś wyższą potęgę dwójki), wtedy podzielnik został usunięty z rozkładanej liczby. Oto opisywany algorytm rozkładu liczby naturalnej n (w języku C++):

```
void factorize(int n)
{
  int p{ 2 };
  while(n > 1)
    if(n % p == 0)
    {
      std::cout << p << '\n';
      n /= p;
    }
    else p++;
}
```

I znów mamy algorytm prosty, ale niezbyt efektywny: przykładowo jeśli n jest dużą liczbą pierwszą, wtedy pętla `while` będzie działać bardzo długo – obróci się tyle razy, ile wynosi wartość n. Dlatego lepiej jest zoptymalizować ten algorytm, sprawdzając podzielniki tylko do pierwiastka kwadratowego z n (włącznie!), jednak wtedy musimy dodać sprawdzenie (na koniec), czy rzeczywiście zidentyfikowaliśmy wszystkie podzielniki:

```
void factorize_opt(int n)
{
  int p{ 2 };
  int sq = (int)(sqrt(n) + 0.5); // zaokrąglona wartość pierwiastka
  while(p <= sq)
    if(n % p == 0)
    {
      std::cout << p << '\n';
      n /= p;
    }
    else p++;
```

```
  if(n > 1)
    std::cout << n << '\n';
}
```

Sito Eratostenesa

Znajdowanie liczb pierwszych (zwłaszcza dużych) nie jest łatwym zadaniem, ale zawsze warto zaznajomić się z klasycznym algorytmem Eratostenesa [17] (zwanego *sitem E.*), który wyznacza wszystkie liczby pierwsze w wybranym przedziale, na przykład $\langle 2, n\rangle$ – bez potrzeby sprawdzania ich podzielności.

Na początku akceptujemy wszystkie liczby naturalne z tego zbioru, nadając wartość `true` elementom tablicy $P[2], P[3], P[4], \ldots, P[n]$, a następnie przechodzimy w pętli $i = 2, 3, \ldots, \sqrt{n}$ sprawdzając, czy liczba i jest uważana za pierwszą ($P[i] =$ `true`). Jeśli tak jest, wtedy wpisujemy wartość `false` na pozycjach odpowiadających wielokrotnościom liczby i – począwszy od i^2 (mniejsze wielokrotności były wykreślone wcześniej – podobny efekt mieliśmy w algorytmie faktoryzacji):

$$i^2, i^2 + i, i^2 + 2i, i^2 + 3i, \ldots, \text{ aż do końca zakresu.}$$

Jedyne pozycje, które pozostaną niewykreślone w tablicy, odpowiadają liczbom pierwszym. Oto przykładowa realizacja tego algorytmu w języku C++:

```
void erato(int n)
{
  vector<bool> P(n+1, true);
  P[0] = P[1] = false; // to nie są liczby pierwsze
  int sq = sqrt(n + 0.5);
  for(int i{ 2 }; i <= sq; i++)
    if(P[i])
      for(int j = i*i; j <= n; j += i)
        P[j] = false;
  for(int i{ 0 }; i <= n; i++)
    if(P[i])
      std::cout << i << '\n';
}
```

Może się zdarzyć, że dana wielokrotność będzie wykreślana więcej niż jeden raz, ale nie przekreśla to głównej zalety sita Eratostenesa: prostoty i przejrzystości.

Na następnej stronie przedstawiono przykład działania sita Eratostenesa dla $n = 32$. W pierwszej kolumnie mamy wszystkie liczby od 0 do n, w drugiej skreślono liczby 0 i 1, a w następnych kolumnach wykreślamy po kolei wielokrotności: dwójki, trójki, czwórki i piątki (jak łatwo zauważyć, skreślanie wielokrotności czwórki jest zbędne, z uwagi na wcześniejsze skreślenie wielokrotności dwójki).

		2	3	4	5
0	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~
1	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
2	2	2	2	2	2
3	3	3	3	3	3
4	4	~~4~~	~~4~~	~~4~~	~~4~~
5	5	5	5	5	5
6	6	~~6~~	~~6~~	~~6~~	~~6~~
7	7	7	7	7	7
8	8	~~8~~	~~8~~	~~8~~	~~8~~
9	9	9	~~9~~	~~9~~	~~9~~
10	10	~~10~~	~~10~~	~~10~~	~~10~~
11	11	11	11	11	11
12	12	~~12~~	~~12~~	~~12~~	~~12~~
13	13	13	13	13	13
14	14	~~14~~	~~14~~	~~14~~	~~14~~
15	15	15	~~15~~	~~15~~	~~15~~
16	16	~~16~~	~~16~~	~~16~~	~~16~~
17	17	17	17	17	17
18	18	~~18~~	~~18~~	~~18~~	~~18~~
19	19	19	19	19	19
20	20	~~20~~	~~20~~	~~20~~	~~20~~
21	21	21	~~21~~	~~21~~	~~21~~
22	22	~~22~~	~~22~~	~~22~~	~~22~~
23	23	23	23	23	23
24	24	~~24~~	~~24~~	~~24~~	~~24~~
25	25	25	25	25	~~25~~
26	26	~~26~~	~~26~~	~~26~~	~~26~~
27	27	27	~~27~~	~~27~~	~~27~~
28	28	~~28~~	~~28~~	~~28~~	~~28~~
29	29	29	29	29	29
30	30	~~30~~	~~30~~	~~30~~	~~30~~
31	31	31	31	31	31
32	32	~~32~~	~~32~~	~~32~~	~~32~~

Tabela 49.1: Przykład działania algorytmu sita Eratostenesa dla $n = 32$ (w kolejnych kolumnach – kolejne etapy skreślania wielokrotności liczb)

50 (VI) – Women's Day – Gnomy *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

50.1 Task description

It is Women's Day and William wants to buy flowers for the three most important women in his life – his mother and two grandmothers. Unfortunately, upon entering the flower shop, he discovers he can only afford to buy two bouquets of flowers. This is no problem for the resourceful William – he can just buy two bouquets now and later rearrange the flowers from them into three groups, one for each of the ladies. He decides that each group should be of equal size and no flowers should be wasted, so the total number of flowers he buys must be divisible by 3.

Task

The shop offers a wide choice of bouquets, small and large. Before making a choice, William wants to know how many options he has. Determine the number of ways to pick a pair of bouquets such that the total number of flowers in these bouquets is divisible by 3.

Input description

The first line of standard input contains one natural number N ($3 \leq N \leq 1\,000\,000$) – the number of different bouquets available at the flower shop. Each of the following N lines contains one natural number ranging from 2 to 100 – the size of the corresponding bouquet. Sizes may repeat but they still correspond to different bouquets and should be counted separately.

Output description

The standard output should contain one whole number – the number of ways William can select a pair of bouquets such that the sum of their sizes is divisible by 3.

Example

For sample input:

```
5
8
6
4
```

```
3
4
```

the correct output is:

```
3
```

Explanation

The first line of the input is the number of bouquets $N = 5$. The bouquet sizes are $(8, 6, 4, 3, 4)$. There are 3 possible ways for William to choose a valid pair of bouquets – he can choose those of size 6 and 3, because $6 + 3 = 9$, which is divisible by 3, or he can choose the bouquet of size 8 and either of the two bouquets of size 4, because $8 + 4 = 12$, which is also divisible by 3.

50.2 Rozwiązanie

Celem zadania było zliczenie wszystkich par bukietów kwiatów, suma rozmiarów których była liczbą podzielną przez 3.

Napisanie działającego rozwiązania nie jest trudne, o ile znany jest nam operator `%`, czyli modulo. Zarówno w C++, jak i w Pythonie, wyrażenie `a % b` oblicza resztę z dzielenia a przez b. W szczególności, `a % 3` to reszta z dzielenia a przez 3, a zatem liczba a jest podzielna przez 3 wtedy i tylko wtedy, gdy spełniony jest warunek `a % 3 == 0`.

Wiedząc to, możemy napisać zagnieżdżoną pętlę, która dla każdej możliwej pary bukietów oblicza sumę ich rozmiarów, a jeśli ta jest podzielna przez 3, powiększa o 1 zmienną służącą za licznik możliwych par. Po ukończeniu tego procesu, wartość tej zmiennej będzie poprawną odpowiedzią.

Takie podejście zdobędzie tylko częściowe punkty. Według opisu wejścia, wszystkich bukietów może być nawet milion, a z tego wynika, że liczba par bukietów może osiągnąć niemal pół biliona, zatem sprawdzanie każdej z nich osobno spowoduje przekroczenie limitu czasowego.

W napisaniu optymalnego rozwiązania pomaga pewna obserwacja o resztach z dzielenia – reszta z dzielenia sumy zależy tylko od reszt z dzielenia składników. Częstym przykładem tej własności są liczby parzyste i nieparzyste – suma dwóch liczb parzystych lub dwóch liczb nieparzystych jest zawsze parzysta, zaś suma liczby parzystej i nieparzystej jest zawsze nieparzysta. Parzystość to nic innego jak podzielność przez 2. W jaki sposób ta własność zachowuje się przy dzieleniu przez 3?

Okazuje się, że suma dwóch liczb jest podzielna przez 3 tylko w dwóch przypadkach:

- oba składniki są podzielne przez 3, albo
- jeden ze składników daje przy dzieleniu przez 3 resztę 1, a drugi 2.

To pozwala nam rozwiązać zadanie w następujący sposób: najpierw zliczamy, ile spośród wszystkich bukietów ma rozmiar o reszcie z dzielenia przez 3 równej 0, ile o reszcie 1, a ile o reszcie 2 – zapiszmy te wartości np. jako b_0, b_1, b_2, odpowiednio. Teraz musimy obliczyć, na ile sposobów można wybrać parę liczb spośród tych o reszcie 0, a na ile sposobów parę liczb takich, że jedna ma resztę 1, a druga 2.

To drugie pytanie jest prostsze – skoro jest b_1 sposobów na wybór pierwszej liczby i b_2 sposobów na wybór drugiej, to istnieje dokładnie $b_1 \cdot b_2$ takich par.

Pierwsze pytanie jest nieco trudniejsze, ale spotykamy się z nim dość często w praktyce – jest ono m.in. równoważne pytaniu: „jeżeli w pokoju znajduje się N osób i każda z każdą się przywita, to ile powitań nastąpi?"

Najłatwiej podejść do tego w następujący sposób: mamy b_0 sposobów na wybranie pierwszej liczby. Po tym wyborze pozostaje $(b_0 - 1)$ liczb, więc tyle sposobów mamy na wybranie drugiej liczby. Może się wydawać, że w takim razie jest $b_0 \cdot (b_0 - 1)$ sposobów na wybranie pary liczb, ale wówczas każdą parę liczylibyśmy podwójnie – jako (A, B) oraz jako (B, A). Poprawnie jest zatem podzielić otrzymany wynik przez 2.

Na podstawie powyższych rozważań, ostateczną odpowiedź możemy otrzymać ze wzoru:

$$\frac{b_0 \cdot (b_0 - 1)}{2} + b_1 \cdot b_2.$$

50.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/WomensDay_Gnomy/solution.py.

```python
n = int(input())
bukiety_o_reszcie = [0, 0, 0]
for _ in range(n):
    rozmiar = int(input())
    bukiety_o_reszcie[rozmiar % 3] += 1
print(bukiety_o_reszcie[0] * (bukiety_o_reszcie[0] - 1)
      // 2 + bukiety_o_reszcie[1] * bukiety_o_reszcie[2])
```

50.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/WomensDay_Gnomy/solution.cpp.

```cpp
#include <iostream>

int main() {
  int n;
  std::cin >> n;
  long long bukiety_o_reszcie[3] {};
  for (int i = 0; i < n; i++) {
    int rozmiar;
    std::cin >> rozmiar;
    bukiety_o_reszcie[rozmiar % 3]++;
  }
  std::cout << (bukiety_o_reszcie[0] * (bukiety_o_reszcie[0] - 1)
    / 2 + bukiety_o_reszcie[1] * bukiety_o_reszcie[2]) << '\n';
  return 0;
}
```

51 (VI) – Women's Day – Skrzaty *(zad. w jęz. angielskim)*

Autorzy: Jan Filipowicz, Politechnika Łódzka, Magdalena Wachulec, AGH w Krakowie
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

51.1 Task description

It is Women's Day and William wants to buy flowers for the three most important women in his life – his mother and two grandmothers. Unfortunately, upon entering the flower shop, he discovers he can only afford to buy two bouquets of flowers. This is no problem for the resourceful William – he can just buy two bouquets now and later rearrange the flowers from them into three groups, one for each of the ladies. He decides that each group should be of equal size and no flowers should be wasted, so the total number of flowers he buys must be divisible by 3.

Task

The shop offers a wide choice of bouquets, small and large. Before making a choice, William wants to know how many options he has. Determine the number of ways to pick a pair of bouquets such that the total number of flowers in these bouquets is divisible by 3.

Start from a new, empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `DANE` should be manually filled with example input data, while the list `WYNIKI` is a place where your program should output the obtained result.

Input description

The input list `DANE` contains N elements ($3 \leq N \leq 100\,000$) representing different bouquets available at the flower shop. Each element is a natural number ranging from 2 to 100 – the size of the corresponding bouquet. Sizes may repeat but they still correspond to different bouquets and should be counted separately.

Output description

After execution of your program, the list `WYNIKI` should contain one element – a whole number representing the number of ways William can select a pair of bouquets such that the sum of their sizes is divisible by 3.

Example

The following picture demonstrates a sample input and the correct output:

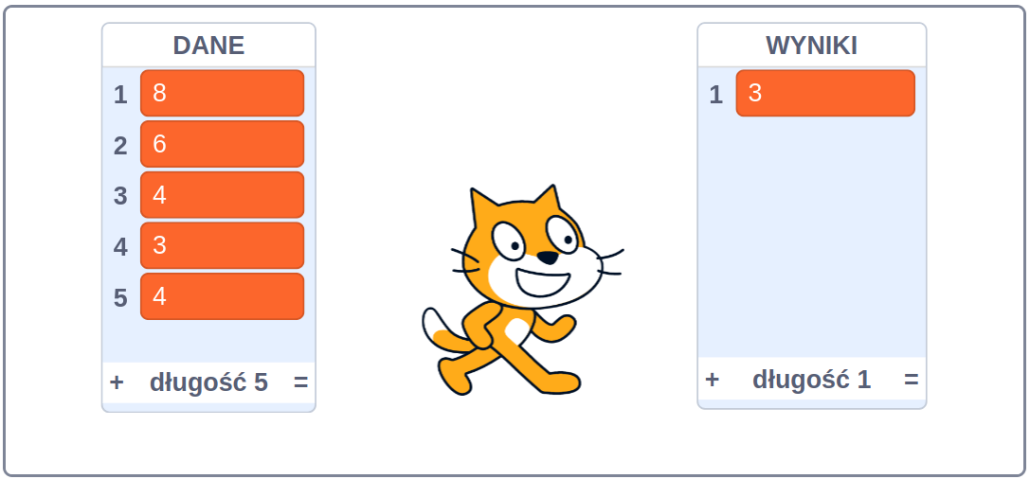


Figure 51.1: Sample case illustration

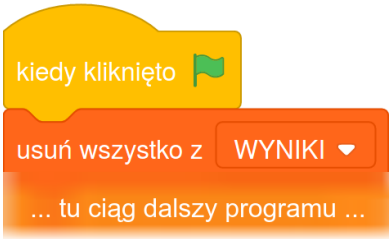
Explanation

The bouquet sizes are $(8, 6, 4, 3, 4)$. There are 3 possible ways for William to choose a valid pair of bouquets – he can choose those of size 6 and 3, because $6 + 3 = 9$, which is divisible by 3, or he can choose the bouquet of size 8 and either of the two bouquets of size 4, because $8 + 4 = 12$, which is also divisible by 3.

Remarks

Please remember, that the list `DANE` should be filled-in manually, by typing the input data with the keyboard. It is a good idea to test the program also with input values other than these in the example.

The content of the list `WYNIKI` should always be initially removed. Your program should therefore start as demonstrated below:



51.2 Rozwiązanie

Sposób rozwiązania jest taki sam, jak zostało to przedstawione w przypadku wersji dla Gnomów.

Przykładowe rozwiązanie znajdziesz w pliku:

./zadania/06_Zadania_obliczeniowe/WomensDay_Skrzaty/solution.sb3.

52 (VI) – Telefony alarmowe – Gnomy

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gnomy (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

52.1 Treść zadania

Firma telekomunikacyjna „Pioter i spółka” instaluje telefony alarmowe wzdłuż autostrady wiodącej z Bitawy do Bitowa na zlecenie bitolandzkiego Ministerstwa Drutów i Drucików.

Przy drodze zainstalowana jest już pewna liczba telefonów i trzeba jedynie uzupełnić brakujące urządzenia. Wymóg zleceniodawcy jest prosty: wszystkie sąsiadujące aparaty muszą znajdować się w jednakowych odległościach od siebie.

Pioter nie jest w ciemię bity i będzie starał się zainstalować jak najmniejszą liczbę telefonów, aby zminimalizować koszty własne. Czy pomożesz mu obliczyć, ile aparatów musi podłączyć?

Zadanie

Napisz program, który wczyta informacje o rozmieszczeniu dotychczasowych telefonów i obliczy minimalną liczbę aparatów, które trzeba zainstalować, tak aby odległość między dowolnymi sąsiednimi telefonami była taka sama.

Położenie telefonu określone jest jako liczba naturalna, oznaczająca jego odległość od Bitawy w bitometrach. Żadne dwa wcześniej zainstalowane telefony nie znajdują się w tym samym miejscu.

Uwaga: odległość pierwszego telefonu od Bitawy, ani ostatniego telefonu od Bitowa nie ma znaczenia. Istotna jest tylko odległość między sąsiednimi telefonami.

Opis wejścia

Dane wejściowe składają się z dwóch wierszy.

Pierwszy wiersz zawiera liczbę naturalną N, oznaczającą liczbę wcześniej zainstalowanych telefonów (przy czym $3 \leq N \leq 100\,000$).

W drugim wierszu znajduje się N liczb naturalnych nieprzekraczających 10^9 (z Bitawy do Bitowa jest szmat drogi!), oddzielonych pojedynczymi odstępami. Liczby te oznaczają położenia istniejących aparatów i podane są w losowej kolejności.

Opis wyjścia

Program powinien wypisać jedną liczbę naturalną oznaczającą liczbę aparatów, które należy zainstalować.

Przykłady

Dla danych wejściowych:

```
4
10 30 40 50
```

program powinien wypisać:

```
1
```

Wystarczy zainstalować jeden aparat w odległości 20 bitometrów od Bitawy.

Dla następujących danych wejściowych:

```
3
100 400 600
```

program powinien wypisać:

```
3
```

Należy zainstalować aparaty w odległościach 200, 300 oraz 500 bitometrów.

W przypadku danych wejściowych:

```
5
100 110 130 150 180
```

program powinien wypisać:

```
4
```

Należy zainstalować aparaty w odległościach 120, 140, 160 oraz 170 bitometrów.

Z kolei dla danych wejściowych:

```
4
100 300 200 400
```

program powinien wypisać:

```
0
```

Nie trzeba instalować żadnego aparatu, gdyż wszystkie znajdują się w tych samych odległościach od siebie.

52.2 Rozwiązanie

Punktem wyjścia do rozwiązania tego zadania jest rozważenie listy *odległości między kolejnymi telefonami*. Pierwszą operacją, którą należy wykonać jest *posortowanie* listy wejściowej – łatwo przeoczyć ten szczegół, bo wszystkie przykłady w treści zadania, oprócz ostatniego, podają odległości telefonów od Bitawy w kolejności rosnącej. Po posortowaniu, wystarczy odjąć każdą wartość od następnej (pierwszą od drugiej, drugą od trzeciej, itd.), aby otrzymać listę odległości między kolejnymi telefonami.

Jeśli wszystkie wartości na tej liście będą identyczne, to znaczy, że nie musimy instalować nowych telefonów i wypisujemy odpowiedź 0. W przeciwnym wypadku, trzeba będzie coś zainstalować – pytanie: gdzie?

Jeśli nie będziemy przez chwilę przejmować się ograniczeniami Piotera, możemy powiedzieć, że instalując telefon co bitometr (czyli w każdym możliwym miejscu), zapewnimy równe odległości między nimi. Liczba telefonów będzie jednak wtedy zapewne zbyt duża. Może więc moglibyśmy instalować telefony np. co dwa bitometry? Zauważmy jednak, że odległość każdego z istniejących telefonów od poprzednika musiałby w takim przypadku być liczbą parzystą (przykładowo, mając zainstalowane telefony odległe o 10 i 18 bitometrów od Bitawy, doinstalowalibyśmy telefony w punktach 12, 14, 16, ale dla telefonów w punktach 10 i 17 to się nie uda, bo 7 jest liczbą nieparzystą). Analogicznie, instalacja telefonów co trzy bitometry wymagałaby, aby każda z odległości między istniejącymi telefonami była podzielna przez trzy, itd.

Stąd prosty wniosek, że największą odległością między telefonami, jaką możemy uzyskać jest największy wspólny dzielnik (NWD) wszystkich odległości między kolejnymi telefonami podanymi w danych wejściowych.

Znając tę wartość, łatwo obliczyć ile ma być wszystkich telefonów (dzieląc przez nią całkowity dystans między ostatnim, a pierwszym telefonem i dodając jeden). Od tej liczby wystarczy odjąć liczbę istniejących telefonów, aby dowiedzieć się ile należy zainstalować.

52.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Telefony_alarmowe_Gnomy/solution.py.

```
import math

n = int(input())
a = sorted(map(int, input().split()))
q = 0
for i in range(1, n):
  q = math.gcd(q, a[i]-a[i-1])
print((a[-1]-a[0])//q - n + 1)
```

52.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Telefony_alarmowe_Gnomy/solution.cpp.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int n; cin >> n;
  int a[100001];
  for(int i=1; i<=n; i++)
    cin >> a[i];
  sort(a+1, a+n+1);
  int ans = a[2]-a[1];
  for(int i=2; i<n; i++)
    ans = __gcd(a[i+1]-a[i], ans);
  cout << (a[n]-a[1])/ans + 1 - n << '\n';
  return 0;
}
```

53 (VI) – Telefony alarmowe – Skrzaty

Autorzy: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (IV edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

53.1 Treść zadania

Firma telekomunikacyjna „Pioter i spółka" instaluje telefony alarmowe wzdłuż autostrady wiodącej z Bitawy do Bitowa na zlecenie bitolandzkiego Ministerstwa Drutów i Drucików.

Przy drodze zainstalowana jest już pewna liczba telefonów i trzeba jedynie uzupełnić brakujące urządzenia. Wymóg zleceniodawcy jest prosty: wszystkie sąsiadujące aparaty muszą znajdować się w jednakowych odległościach od siebie.

Pioter nie jest w ciemię bity i będzie starał się zainstalować jak najmniejszą liczbę telefonów, aby zminimalizować koszty własne. Czy pomożesz mu obliczyć, ile aparatów musi podłączyć?

Zadanie

W nowym, pustym projekcie Scratch, zadeklaruj dwie listy: `DANE` i `WYNIKI`. Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać efekt swojego działania. Program powinien wczytać z listy wejściowej `DANE` informacje o rozmieszczeniu dotychczasowych telefonów i obliczyć minimalną liczbę aparatów, które trzeba zainstalować, tak aby odległość między dowolnymi sąsiednimi telefonami była taka sama.

Położenie telefonu określone jest jako liczba naturalna, oznaczająca jego odległość od Bitawy w bitometrach. Żadne dwa wcześniej zainstalowane telefony nie znajdują się w tym samym miejscu.

Uwaga: odległość pierwszego telefonu od Bitawy, ani ostatniego telefonu od Bitowa nie ma znaczenia. Istotna jest tylko odległość między sąsiednimi telefonami.

Opis wejścia

Lista `DANE` zawiera N elementów ($3 \leq N \leq 1000$), będących liczbami naturalnymi nieprzekraczającymi 10^9 (z Bitawy do Bitowa jest szmat drogi!). Liczby te oznaczają położenia istniejących aparatów i podane są w losowej kolejności.

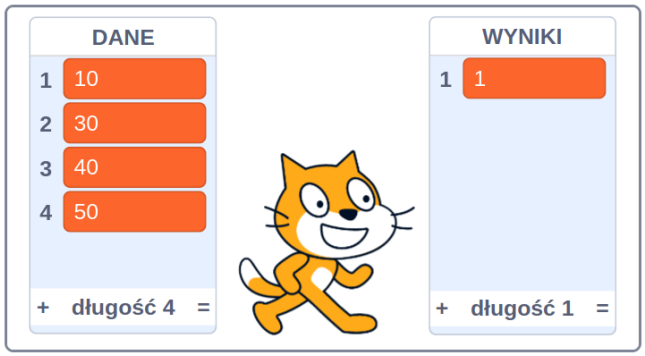
Opis wyjścia

Twój program, po uruchomieniu (za pomocą zielonej flagi), powinien umieścić na liście `WYNIKI` jeden element: liczbę naturalną oznaczającą liczbę aparatów, które należy zainstalować.

Przykłady

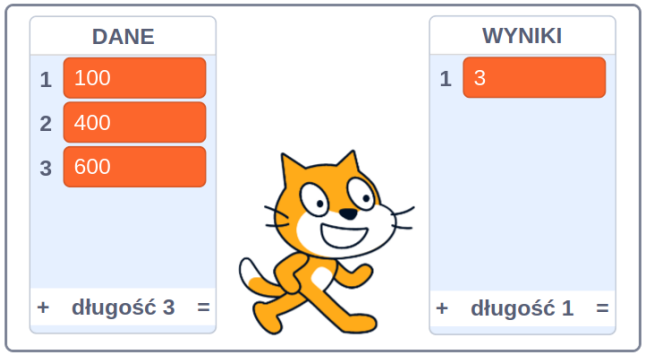
Na kolejnych rysunkach pokazano przykładowe wyniki działania programu.

Przykład 1



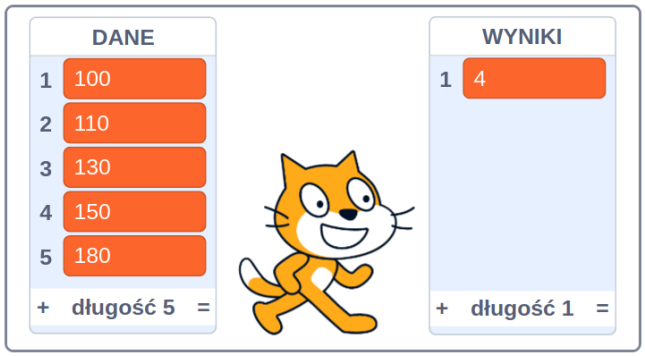
Wyjaśnienie przykładu: Przy autostradzie znajdują się już cztery telefony. Odległości między nimi są takie, że wystarczy zainstalować jeszcze tylko jeden aparat (w odległości 20 bitometrów od Bitawy).

Przykład 2



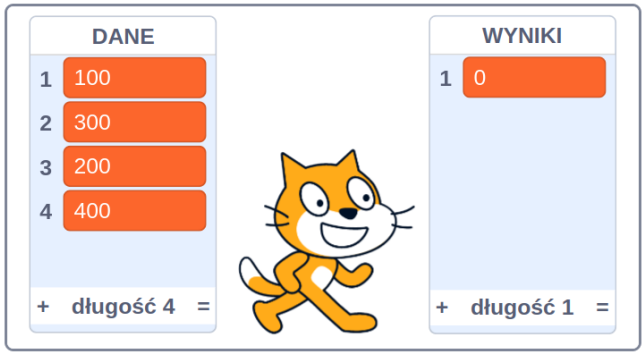
Wyjaśnienie przykładu: Należy zainstalować aparaty w odległościach 200, 300 oraz 500 bitometrów.

Przykład 3



Wyjaśnienie przykładu: Należy zainstalować aparaty w odległościach 120, 140, 160 oraz 170 bitometrów.

Przykład 4

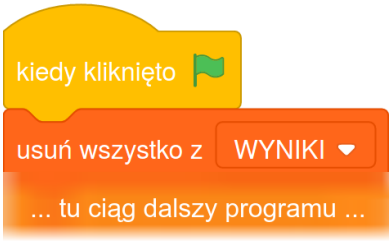


Wyjaśnienie przykładu: Nie trzeba instalować żadnego aparatu, gdyż wszystkie znajdują się w tych samych odległościach od siebie.

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano tutaj:



53.2 Rozwiązanie

Sposób rozwiązania został omówiony w wersji dla Gnomów. W przypadku Scratcha potrzebujemy oczywiście własnoręcznie zaimplementować sortowanie i obliczanie NWD.

Przykładowe rozwiązanie, wykorzystujące sortowanie bąbelkowe i iteracyjny algorytm obliczania NWD znajdziesz w pliku:

./zadania/06_Zadania_obliczeniowe/Telefony_alarmowe_Skrzaty/solution.sb3.

54 (VI) – Ada's Garden – Gnomy *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap regionalny/liga algorytmiczna)
Język programowania: C++/Python

54.1 Task description

Ada enjoys helping her grandpa in the garden. She dreams of having a garden of her own one day, where she could plant any flowers she wants. Grateful for her help, her grandpa decides to give her exactly that for her birthday – or perhaps *almost* exactly.

As her birthday gift, Ada can choose any part of her grandpa's garden – this part will belong to her, and she will be allowed to make decisions about it. Ada's grandpa only imposes the following limits on her choice:

- Ada's part must be entirely contained inside grandpa's garden, which is a rectangle with sides a, b, where a and b are whole numbers;
- Ada's part must itself be a rectangle with sides parallel to the sides of grandpa's garden, and its side lengths must be whole numbers;
- its area cannot exceed some value N chosen by grandpa.

Task

Of course, Ada wants her new garden to be as big as possible, meaning it should have the largest area she can reach within grandpa's limits. Help her find the dimensions of the rectangle she should choose. If there are several possible answers, choose the one that is closest in shape to a square – that is, the one with the smallest absolute difference between its side lengths.

Input description

The first and only line of the standard input contains three natural numbers separated by spaces. The first two are the side lengths a, b of grandpa's garden, where $a \leq b$. The third is N – the maximum area allowed for Ada's part (you may safely assume $N < a \times b$). All numbers range from 1 to $100\,000\,000\,000\,000$.

Output description

The standard output should contain one line with two natural numbers separated by a single space – the ideal side lengths for Ada's part of the garden. Just like in the input, the first number must be less than or equal to the second number (that is, if the side lengths differ, print the shorter one first).

Example

For sample input:

```
7 12 13
```

the correct output is:

```
3 4
```

whereas for sample input:

```
7 15 63
```

the correct output is:

```
7 9
```

Explanation

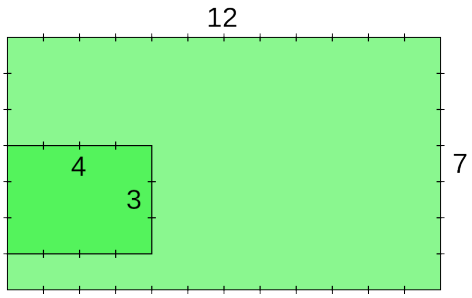


Figure 54.1: Sample case illustration

In the first test case, there is no way to fit a rectangle with area 13 inside the given garden – but there are several ways to choose a rectangle with area 12: 1×12, 2×6, or 3×4. The correct choice is 3×4 because its side lengths differ by just 1.

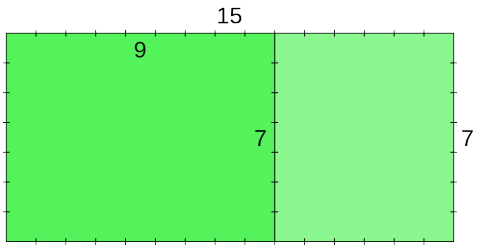


Figure 54.2: Sample case illustration

In the second test case, it is possible to choose a rectangle with the maximum allowed area 63, and 7×9 is the only way to do so.

54.2 Rozwiązanie

W zadaniu należało znaleźć wymiary prostokąta o całkowitych długościach boków i jak największym polu mniejszym od zadanego N, który mieściłby się w zadanym prostokącie o bokach a, b. Boki prostokątów miały być odpowiednio wzajemnie równoległe, a remisy należało rozstrzygać poprzez wybór prostokąta o jak najmniejszej różnicy długości boków.

Zacznijmy od najprostszego pomysłu na rozwiązanie: byłoby nim wypróbowanie każdego możliwego prostokąta o całkowitych długościach boków c, d takich, że $1 \leq c \leq a$ oraz $1 \leq d \leq b$. Dla każdego takiego prostokąta sprawdzalibyśmy, czy spełnia on warunki zadania, a jeśli tak, to czy jest lepszy od najlepszego dotychczas znalezionego kandydata. Takie rozwiązanie można zaimplementować łatwo przy pomocy dwóch zagnieżdżonych pętli, ale jest ono mało wydajne, gdyż wszystkich możliwych prostokątów jest $a \cdot b$.

Znaczną poprawę uzyskalibyśmy, gdybyśmy znaleźli sposób na wydajne wywnioskowanie najlepszej wartości drugiego boku d dla danego pierwszego boku c. Da się to zrobić prostą operacją dzielenia liczb całkowitych: `d = N / c` (w języku Python `d = N // c`). Otrzymujemy wtedy największą wartość d taką, że $c \cdot d \leq N$. Może się przy tym okazać, że otrzymana wartość d przekroczy b – wówczas należy ją zmniejszyć do b.

Ta obserwacja pozwala nam pozbyć się jednej z pętli. Teraz wystarczy dla wszystkich możliwych wartości c obliczyć odpowiadającą im wartość d i sprawdzić, czy prostokąt o bokach c, d jest lepszy od dotychczasowego najlepszego wyniku. Liczba wykonań pętli zmniejszyła się zatem z $a \cdot b$ do a, lecz to wciąż jest niewystarczające, by przejść największe testy.

W tym celu konieczna jest kolejna obserwacja: możemy ograniczyć rozważania do przypadków, w których $c \leq d$. Można to wyjaśnić w następujący sposób: gdyby najlepszy prostokąt nie spełniał tego warunku, czyli miał $c > d$, np. $c = 9, d = 7$, to z całą pewnością znaleźlibyśmy go już wcześniej rozpatrując odwrotne przypisanie wartości: $c = 7, d = 9$.

Zauważmy też, że podczas gdy wartość c wraz z postępem pętli rośnie, wartość d będzie tylko maleć. W takim razie, w momencie gdy wartość c przekroczy d, możemy natychmiast przerwać pętlę i wypisać znalezioną odpowiedź, bo dalsze wykonania pętli nie przyniosłyby żadnego przydatnego skutku.

Taka optymalizacja znacząco przyspiesza program, bowiem punkt, w którym c przekracza d, następuje stosunkowo szybko. Jako że d to $\frac{N}{c}$ (z dokładnością do zaokrąglenia), to owo przekroczenie ma miejsce wtedy, gdy osiągniemy $c > \frac{N}{c}$, czyli $c \cdot c > N$. Graniczna wartość c, tzn. taka, że $c \cdot c = N$, to oczywiście $\sqrt{N}$, czyli *pierwiastek kwadratowy* z N. Dla maksymalnego $N = 100\,000\,000\,000\,000$, wartość $\sqrt{N}$ wynosi zaledwie 10 milionów, a tyle powtórzeń pętli jest już wykonalne w rozsądnym czasie.

54.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Adas_garden_Gnomy/solution.py.

```python
a, b, n = [int(m) for m in input().split()]
pole_wyniku = 0
wynik_a = 0
wynik_b = 0
for krotszy_bok in range(1, a + 1):
    dluzszy_bok = n // krotszy_bok
    if dluzszy_bok < krotszy_bok:
        break
    if dluzszy_bok > b:
        dluzszy_bok = b
    pole = krotszy_bok * dluzszy_bok
    if pole >= pole_wyniku:
        pole_wyniku = pole
        wynik_a = krotszy_bok
        wynik_b = dluzszy_bok
print(wynik_a, wynik_b)
```

54.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Adas_garden_Gnomy/solution.cpp.

```cpp
#include <iostream>

int main() {
  long long a, b, n;
  std::cin >> a >> b >> n;
  long long pole_wyniku = 0;
  long long wynik_a = 0;
  long long wynik_b = 0;
  for (long long krotszy_bok = 1; krotszy_bok <= a; krotszy_bok++) {
    long long dluzszy_bok = n / krotszy_bok;
    if (dluzszy_bok < krotszy_bok)
      break;
    if (dluzszy_bok > b)
      dluzszy_bok = b;
    long long pole = krotszy_bok * dluzszy_bok;
    if (pole >= pole_wyniku) {
      pole_wyniku = pole;
      wynik_a = krotszy_bok;
      wynik_b = dluzszy_bok;
    }
  }
  std::cout << wynik_a << ' ' << wynik_b << '\n';
  return 0;
}
```

55 (VI) – Ada's Garden – Skrzaty *(zad. w jęz. angielskim)*

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (III edycja – zawody algorytmiczne – etap regionalny/liga algorytmiczna)
Język programowania: Scratch

55.1 Task description

Ada enjoys helping her grandpa in the garden. She dreams of having a garden of her own one day, where she could plant any flowers she wants. Grateful for her help, her grandpa decides to give her exactly that for her birthday – or perhaps *almost* exactly.

As her birthday gift, Ada can choose any part of her grandpa's garden – this part will belong to her, and she will be allowed to make decisions about it. Ada's grandpa only imposes the following limits on her choice:

- Ada's part must be entirely contained inside grandpa's garden, which is a rectangle with sides a, b, where a and b are whole numbers;
- Ada's part must itself be a rectangle with sides parallel to the sides of grandpa's garden, and its side lengths must be whole numbers;
- its area cannot exceed some value N chosen by grandpa.

Task

Of course, Ada wants her new garden to be as big as possible, meaning it should have the largest area she can reach within grandpa's limits. Help her find the dimensions of the rectangle she should choose. If there are several possible answers, choose the one that is closest in shape to a square – that is, the one with the smallest absolute difference between its side lengths.

Start from a new, empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `DANE` should be manually filled with example input data, while the list `WYNIKI` is a place where your program should output the obtained results.

Input description

The input list `DANE` contains three natural numbers. The first two are the side lengths a, b of grandpa's garden, where $a \leq b$. The third is N – the maximum area allowed for Ada's part (you may safely assume $N < a \times b$). All numbers range from 1 to 1 000 000.

Output description

After execution of your program, the list `WYNIKI` should contain two natural numbers – the ideal side lengths for Ada's part of the garden. Just like in the input, the first number must be less than or equal to the second number (that is, if the side lengths differ, print the shorter one first).

Example

The following figure demonstrates sample input and correct output:

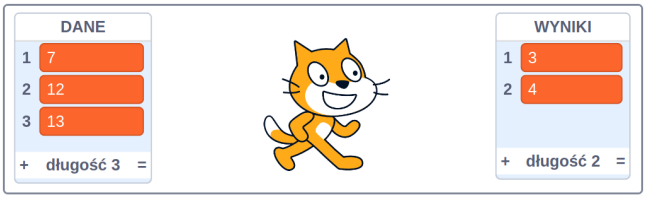


Figure 55.1: Example 1

Another example is presented below:

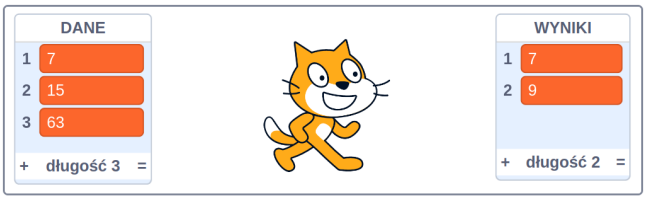


Figure 55.2: Example 2

Explanation

In the first test case, there is no way to fit a rectangle with area 13 inside the given garden – but there are several ways to choose a rectangle with area 12: 1×12, 2×6, or 3×4. The correct choice is 3×4 because its side lengths differ by just 1:

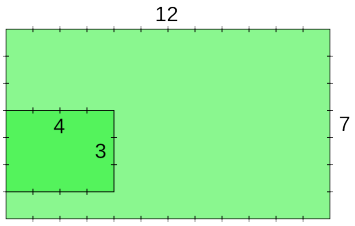


Figure 55.3: Example 1 – explanation

In the second test case, it is possible to choose a rectangle with the maximum allowed area 63, and 7×9 is the only way to do so:

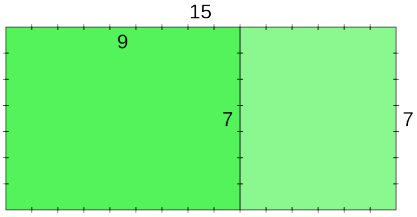


Figure 55.4: Example 2 – explanation

55.2 Rozwiązanie

W wersji dla Scratcha wynik obliczany jest w taki sam sposób, jak zostało to omówione w wersji dla Gnomów. Przykładową implementację znajdziesz w pliku:

./zadania/06_Zadania_obliczeniowe/Adas_garden_Skrzaty/solution.sb3.

56 (VI) – Hipoteza Bajtacha – Gnomy

*Autor: **Filip Turoboś**, **Politechnika Łódzka***

*Kategoria: **Gnomy (II edycja – zawody algorytmiczne – etap lokalny)***

*Język programowania: **C++/Python***

56.1 Treść zadania

Prof. Bajtach jest wybitnym uczonym pracującym w Bajtockiej Akademii Nauk Wszelakich. Jego specjalizacją jest matematyka, a w szczególności – liczby pierwsze. Już jako student miał pewne przypuszczenie, że każdą parzystą liczbę całkowitą począwszy od 4 da się zapisać jako sumę dwóch liczb pierwszych[1].

Dla małych liczb jest to łatwo zweryfikować: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 5 + 3$, ale dla większych... jest to problem nie lada!

Niestety, przez długi czas jako zwykły pracownik Bajtockiej Akademii nie mógł poświęcić się weryfikacji prawdziwości swoich przypuszczeń. Ale teraz, gdy dostał upragniony grant badawczy od Króla Bajtozaura Piętnastego, może wreszcie pozwolić sobie na zastosowanie w tym celu słynnej metody Bajtockiej Indukcji.

Ta rewolucyjna metoda dowodzenia twierdzeń wymaga bardzo dużej (ściśle rzecz biorąc: nieskończonej) liczby pracowników. Bez wchodzenia w detale, polega ona na tym, że każdy z pracujących badaczy weryfikuje prawdziwość dla kilku przydzielonych mu liczb.

Jednym z badaczy zatrudnionych przez profesora Bajtacha jesteś Ty. Wykonaj zlecone zadanie i rozłóż podane liczby parzyste na sumę dwóch liczb pierwszych.

Zadanie

Prof. Bajtach chce możliwie ujednolicić uzyskiwane przez swoich pracowników wyniki. W tym celu przekazał im bardzo szczegółowe wytyczne, opisujące co należy zrobić z otrzymanym zestawem danych.

> *Najdrożsi!*
> W załączniku do poniższego listu znajdziecie T liczb parzystych, nie mniejszych niż cztery. Jako, że nie chcę Was przeciążać, obiecuję, że T nigdy nie przekroczy $50\,000$. Dla każdej podanej liczby parzystej n, znajdźcie proszę takowe liczby pierwsze p i q, które by wymogi poniższe spełniały:
> - $p \leqslant q$;
> - $p + q = n$;
> - $q - p$ jest możliwie największe.
>
> Proszę, zabierzcie się za zadanie jak najszybciej! Nauka nie może czekać!
> Z serdecznymi pozdrowieniami,
> *S. Bajtach*

[1] Jest to prawdziwy, nierozwiązany dotychczas w pełnej ogólności problem znany pod nazwą *hipotezy Goldbacha*.

Po otwarciu przypisanego Ci zestawu liczb zorientowałeś się, że nie są one duże. Żadna z nich nie przekracza bowiem 10^7. Oddychając z ulgą, zabierasz się za zlecone Ci zadanie...

Opis wejścia

Pierwszy wiersz standardowego wejścia zawiera pojedynczą liczbę T ($1 \leqslant T \leqslant 50\,000$), określającą liczbę przypadków testowych, które masz przebadać. W każdej z kolejnych T linii wejścia znajdziesz pojedynczą liczbę parzystą n, spełniającą ograniczenia $4 \leqslant n \leqslant 10\,000\,000$.

Opis wyjścia

Na wyjściu standardowym dla każdej badanej przez Ciebie liczby n wypisz liczby pierwsze $p \leqslant q$ spełniające warunki zadania. Pamiętaj, że 1 i 0 nie są liczbami pierwszymi.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
3
12
14
18
```

prawidłową odpowiedzią jest:

```
5 7
3 11
5 13
```

Wyjaśnienie

W powyższym przykładzie 12 jest sumą 5 i 7. Jest to jedyny możliwy taki rozkład, więc nie musimy sprawdzać czy maksymalizuje on różnicę między tymi liczbami pierwszymi.

W drugim przypadku możemy zapisać 14 jako $7 + 7$ oraz $3 + 11$. Zauważmy, że różnica $7 - 7$ wynosi zero, podczas gdy $11 - 3 = 8$. Jako że $8 > 0$, odpowiedzią jest właśnie para 3 i 11.

W ostatnim przypadku możemy przedstawić 18 jako $7 + 11$ lub $5 + 13$. Ponownie, w drugim z tych przypadków różnica między składnikami tej sumy jest większa.

56.2 Rozwiązanie

Zadanie polegało na rozłożeniu zadanej liczby parzystej n, nie mniejszej niż 4, na sumę liczb pierwszych tak, by różnica między tymi liczbami była możliwie największa.

Zadanie inspirowane jest problemem znanym jako *Hipoteza Goldbacha*, który można sformułować następująco:

„*Każda liczba naturalna parzysta większa od 2 jest sumą dwóch liczb pierwszych.*”

Wiadomo, że taki rozkład jest możliwy dla wszystkich $n \leqslant 4 \cdot 10^{17}$.

Optymalne rozwiązanie problemu (którego implementacje przedstawiono na kolejnych stronach) składa się z następujących kroków:

- Wczytujemy wszystkie występujące w zadaniu liczby n_i, wybieramy największą z nich i konstruujemy dla niej sito Eratostenesa;
- Uzyskane liczby pierwsze „sklejamy” w jeden kontener (listę, wektor), dzięki któremu przeszukiwania prowadzić będziemy *tylko* wśród liczb spełniających warunki zadania;
- Dla każdej liczby n_i wskazujemy jej rozkład na składniki poprzez iterowanie po naszej liście/wektorze liczb pierwszych, sprawdzając, czy spełniony jest warunek sumowania się do n_i.

Rozwiązania nieoptymalne (zwracające poprawne wyniki, ale zbyt wolne) mogą obejmować m.in.:

- Wielokrotne wyznaczanie sita;
- Iterowanie po wszystkich liczbach z zakresu (zamiast ograniczyć się do liczb pierwszych);
- Naiwne weryfikowanie pierwszości składników „liczba po liczbie”.

56.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Hipoteza_Bajtacha/solution.py.

```
T = int(input())
X = []
maximum = 0
for i in range(T):
    tau = int(input())
    X.append(tau)
    if tau>maximum:
        maximum = tau

#Sieve of Eratostenes
prime = [True for i in range(maximum)]
primes = []
p = 2
while (p * p <= maximum):
    if (prime[p]):
        primes.append(p)
        for i in range(p * 2, maximum, p):
            prime[i] = False
    p += 1
while (p<=maximum/2):
    if (prime[p]):
        primes.append(p)
    p += 1

prime[0] = False
prime[1] = False

for j in range(T):
    i=0
    q = X[j]-primes[i]
    while not (prime[q]):
        i+=1
        q = X[j] - primes[i]
    print(str(primes[i])+" "+str(q))
```

56.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Hipoteza_Bajtacha/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(){
  //Precalculating Sieve of Eratostenes
  int T;
  cin>>T;
  vector<int> X = vector<int>(T,0);
  int maximum=0;
  for(int i = 0; i < T; i++){
    cin>>X[i];
    if(X[i]>maximum){ maximum = X[i];}
  }
  vector<bool> prime = vector<bool>(maximum,true);
  vector<int> primes = vector<int>();
  int p = 2;
  for(;p * p <= maximum; p++){
    if(prime[p]){
      primes.push_back(p);
      for(int i = p * 2; i<=maximum; i+=p){
        prime[i] =false;
      }
    }
  }
  prime[0] = false;
  prime[1] = false;
  for(p; p<=maximum/2; p++){
    if (prime[p]){
      primes.push_back(p);
    }
  }
  for(int j = 0; j < T; j++){
    int i=0;
    int q = X[j]-primes[i];
    while(not prime[q]){
      i++;
      q = X[j] - primes[i];
    }
    cout<<primes[i]<<" "<<q<<'\n';
  }
  return 0;
}
```

57 (VI) – Przeprawa przez cieśninę – Gremliny

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (III edycja – liga algorytmiczna)
Język programowania: C++/Python

57.1 Treść zadania

Pewien pasterz, powszechnie znany wśród miejscowych jako Dziadek Fortran, posiada stale rosnące stado owiec. W istocie, owych owiec może być obecnie nawet nieco zbyt wiele – na łące należącej do Dziadka Fortrana powoli zaczyna brakować miejsca, stąd roztropny pasterz zakupił nową, większą działkę gruntu i rozpoczął plany transportu trzody.

Jest tylko jeden problem – nowa łąka znajduje się po przeciwnej stronie Cieśniny Gigabajtarskiej. Dlatego też Dziadek Fortran, nie chcąc płacić olbrzymiej kwoty za wynajem statku transportowego, wziął się za budowę solidnej tratwy. Pomóż mu ustalić, w jaki sposób może efektywnie wykorzystać czas, aby przeprowadzka trwała jak najkrócej.

Zadanie

Początkowo tratwa pasterza posiada dość miejsca tylko dla niego samego. Każdą godzinę pracy może on dowolnie wykorzystać na jeden z dwóch sposobów:

- może rozbudować tratwę tak, by była w stanie utrzymać jedną dodatkową owcę,

lub:

- może wypełnić obecnie posiadaną tratwę owcami i przetransportować je na drugą stronę cieśniny, a następnie wrócić tratwą samemu.

Znając liczbę owiec ustal, ilu godzin pracy minimalnie potrzebuje pasterz, aby szczęśliwie zakończyć transport stada.

Opis wejścia

W pierwszym i jedynym wierszu standardowego wejścia znajduje się jedna liczba naturalna N (spełniająca warunek: $2 \leq N \leq 2 \cdot 10^{18}$), oznaczająca liczbę owiec w trzodzie Dziadka Fortrana.

Opis wyjścia

Na standardowe wyjście należy wypisać jedną liczbę naturalną – minimalną liczbę godzin pracy niezbędną do przetransportowania całej trzody na drugą stronę cieśniny.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
7
```

prawidłową odpowiedzią jest:

```
6
```

Wyjaśnienie

Jeśli Dziadek Fortran ma 7 owiec, może na przykład tak rozłożyć pracę:

1. rozbudowa tratwy do rozmiaru 1;
2. rozbudowa tratwy do rozmiaru 2;
3. transport 2 owiec na drugą stronę cieśniny;
4. rozbudowa tratwy do rozmiaru 3;
5. transport 3 owiec na drugą stronę cieśniny;
6. transport ostatnich 2 owiec na drugą stronę cieśniny.

Naturalnie istnieje też kilka innych sposobów, na które mógłby wykonać zadanie w przeciągu 6 godzin – liczy się tylko fakt, że nie ma możliwości zrobić tego szybciej, a zatem poprawną odpowiedzią jest 6.

57.2 Rozwiązanie

W zadaniu chcemy ustalić minimalną liczbę akcji, które pozwolą nam przetransportować N owiec na drugą stronę cieśniny, jednak aby najlepiej zrozumieć rozwiązanie, warto rozważyć prostszą wersję problemu: powiedzmy, że owiec jest nieograniczona ilość, natomiast mamy do wykorzystania tylko A akcji – ile maksymalnie owiec możemy przetransportować?

Po pierwsze zauważmy, że najbardziej optymalnie jest najpierw powiększyć tratwę do pewnego rozmiaru, a potem wykonywać jedynie akcje transportu. Możemy tego łatwo dowieść: jeżeli w pewnym rozwiązaniu jakaś akcja transportu następuje zaraz przed rozbudową tratwy, to zamiana kolejności tych dwóch akcji pozwoli nam bez żadnej straty przetransportować jedną dodatkową owcę.

Ile owiec możemy w takim razie przemieścić? Jeżeli poświęcimy R akcji na rozbudowę i T akcji na transport, gdzie $R+T = A$, to zdołamy przenieść $R \cdot T$ owiec (T przewozów po R owiec każdy). Innymi słowy staramy się zmaksymalizować iloczyn dwóch liczb o zadanej sumie. Intuicja powinna nam podpowiadać, że największy iloczyn osiągniemy, jeśli $R = T = \frac{A}{2}$.

Żeby tego dowieść, oznaczmy $a = \frac{A}{2}$. Jeśli $R = T = a$, to iloczyn $R \cdot T$ wynosi po prostu a^2. Jeśli $R \neq T$, to aby zachowały tę samą sumę, jedną z liczb musimy powiększyć o pewną wartość, a drugą odpowiednio pomniejszyć. Nazwijmy tę wartość b, wówczas $R = a + b$ i $T = a - b$, a iloczyn $R \cdot T$ wynosi $(a + b)(a - b) = a^2 - b^2$. Zauważmy, że b^2 występuje w tym ostatnim wyrażeniu ze znakiem minus – zatem im bardziej wartości R i T różnią się, tym mniejszy ich iloczyn.

Z zastosowaniem powyższego rozumowania w praktyce jest oczywiście drobny problem – jeżeli liczba A jest nieparzysta, to nie możemy wykonać po równo każdej z akcji. Wówczas jednej z nich musimy po prostu wykonać o jedną więcej.

To rozwiązuje prostszą wersję problemu. Wracając do wersji oryginalnej zauważmy, że nie ma powodu do zmiany strategii – znaleźliśmy wszak właśnie sposób na optymalne wykorzystanie czasu, by przetransportować jak najwięcej owiec.

Pomijając na razie przypadek nieparzystej wartości A: ustaliliśmy, że liczba akcji każdego rodzaju, które powinniśmy wykonać, wynosi $a = \frac{A}{2}$, i jesteśmy wówczas w stanie przetransportować a^2 owiec. Jako że naszym celem jest przetransportować N owiec, to potrzebna jest nam taka wartość a, że $a^2 \geq N$, czyli $a \geq \sqrt{N}$. Wiemy, że a musi być jak najmniejszą liczbą całkowitą spełniającą ten warunek, co możemy zapisać jako $a = \lceil \sqrt{N} \rceil$, gdzie $\lceil x \rceil$ jest symbolem funkcji „sufit", czyli po prostu zaokrąglenia w górę. Ponieważ $A = 2a$, to naszym wynikiem jest $2\lceil \sqrt{N} \rceil$.

Pozostaje jeszcze kwestia nieparzystej wartości A. W takim przypadku powyższe rozwiązanie przeszacuje optymalną wartość A o dokładnie 1. Najłatwiej zatem po prostu skorygować wynik po fakcie, tj. obliczyć $a = \lceil \sqrt{N} \rceil$, a następnie sprawdzić, czy $a(a - 1) \geq N$ – jeśli tak, to możemy zaoszczędzić jedną akcję i poprawnym wynikiem jest $2a - 1$.

Pod koniec warto wspomnieć o kwestii obliczania wartości $\lceil \sqrt{N} \rceil$ w praktyce. W językach C++ i Python istnieją odpowiednie poświęcone temu funkcje `sqrt(x)` oraz `ceil(x)`, ale mają one w tym przypadku istotną wadę – operują na liczbach zmiennoprzecinkowych, a te mają ograniczoną precyzję i w razie potrzeby ulegają zaokrągleniom. Dane w zadaniu potrafią być na tyle duże, że bezpośrednie zastosowanie tych funkcji może prowadzić niekiedy do błędnych wyników.

Aby temu zapobiec, można wykorzystać dowolny dostatecznie szybki algorytm obliczający część całkowitą pierwiastka – w języku Python taki algorytm jest dostępny jako funkcja `isqrt`, zaś w C++ wymagałby własnej implementacji. Alternatywnie, można oszacować wartość odpowiedzi funkcją `sqrt`, a następnie sprawdzić i w razie potrzeby odpowiednio skorygować wynik.

57.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Przeprawa_przez_ciesnine/solution.py.

```python
from math import isqrt

liczba_owiec = int(input())
czas_powiekszania_tratwy = isqrt(liczba_owiec - 1)
czas_transportu = czas_powiekszania_tratwy + 1
if czas_powiekszania_tratwy * czas_transportu < liczba_owiec:
    czas_powiekszania_tratwy += 1
print(czas_powiekszania_tratwy + czas_transportu)
```

57.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Przeprawa_przez_ciesnine/solution.cpp.

```cpp
#include <cmath>
#include <iostream>

int main() {
  long long liczba_owiec;
  std::cin >> liczba_owiec;
  long long czas_powiekszania_tratwy = (long long)std::sqrt(liczba_owiec);
  long long czas_transportu = czas_powiekszania_tratwy;
  if (czas_powiekszania_tratwy * czas_transportu < liczba_owiec)
    czas_powiekszania_tratwy++;
  if (czas_powiekszania_tratwy * czas_transportu < liczba_owiec)
    czas_transportu++;
  std::cout << (czas_powiekszania_tratwy + czas_transportu) << '\n';
  return 0;
}
```

58 (VI) – Zaginione liczby pierwsze – Gremliny

Autor: Paweł Tarasiuk, Politechnika Łódzka

Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap finałowy)

Język programowania: C++/Python

58.1 Treść zadania

Bajtek miał wielki talent do znajdywania rzeczy tam, gdzie nikt inny by się ich nie spodziewał. Znajdywanie igieł w stogu siana, dziur w całym, wzorców w tekście lub kluczy w lodówce to dla niego codzienność. Wydarzeniem wyjątkowym, nawet jak na Bajtka, było jednak odnalezienie podczas leśnego spaceru pary nieparzystych liczb pierwszych. Bajtek – wiedząc, że na pewno da się je w ten sposób jednoznacznie odtworzyć – zapisał sobie na kartce ich iloczyn, czyli liczbę a. Dopóki Bajtek pamiętał obie liczby, mógł wykonywać z ich użyciem przeróżne obliczenia – i tak odkrył kolejną szczególną liczbę, tym razem b. Bajtek badał po kolei dodatnie liczby naturalne, ale żadna mniejsza liczba nie była tak wyjątkowa, jak b. Otóż dla dowolnej liczby naturalnej x z przedziału od 1 do $a-1$, która była względnie pierwsza z a, liczba $x^b - 1$ była wielokrotnością liczby a.

Na drugi dzień Bajtek kompletnie zapomniał, ile wynosiły jego dwie liczby pierwsze. Spora strata – pomyślał, szykując się na mozolne próby rozbicia liczby a. Niestety, ani w pamięci, ani w notatkach nie znalazł swoich liczb pierwszych. Jedyne, co zostało z jego odkryć, to liczby a i b. Bohater, jak to zwykle, zwrócił się o pomoc właśnie do Ciebie.

Zadanie

Pomóż Bajtkowi odzyskać zaginione liczby pierwsze. Na podstawie liczb a oraz b trzeba ustalić, jakie dwie liczby pierwsze na początku historii odnalazł w lesie Bajtek.

Opis wejścia

Na początku danych wejściowych znajduje się informacja o tym, ile razy trzeba zademonstrować zdolność do rozwiązywania Bajtkowych problemów, czyli:

t – liczba przypadków testowych, $1 \leq t \leq 1\,000\,000$.

Każda z kolejnych t linii zawiera parę liczb zanotowanych przez Bajtka, w kolejności zgodnej z ich odkrywaniem, czyli:

a – iloczyn zaginionych liczb pierwszych, $1 \leq a \leq 1\,000\,000\,000$,

b – wyjątkowa liczba, $1 \leq b \leq 1\,000\,000\,000$. Wiemy, że jest to najmniejsza taka liczba naturalna, że $\forall_{x \in \{1,2,\ldots,a-1\},\ \mathrm{NWD}(x,a)=1} \quad x^b - 1 \equiv 0 \pmod{a}$.

Oczywiście Bajtek nie kłamie. Dla każdego testu możesz założyć, że a faktycznie jest iloczynem dwóch nieparzystych liczb pierwszych.

Opis wyjścia

Dla każdej kolejnej pary liczb a, b należy wypisać w kolejności niemalejącej dwie liczby pierwsze p, q. Oznacza to, że kompletna odpowiedź powinna się składać z t linii.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
1
15 4
```

prawidłową odpowiedzią jest:

```
3 5
```

Z kolei dla przykładowego wejścia:

```
2
247 36
713 330
```

prawidłową odpowiedzią jest:

```
13 19
23 31
```

Wyjaśnienie przykładów

W pierwszym przypadku liczba a to 15. Dla tak małej liczby odpowiedź postaci $a = 3 \cdot 5$ nasuwa się sama i jest możliwa do obliczenia w pamięci. Standardowo, mamy jednak także informację o liczbie $b = 4$. Zgodnie z treścią zadania – liczby:

- $1^4 - 1 = 0$;
- $2^4 - 1 = 15$;
- $4^4 - 1 = 255 = 15 \cdot 17$;
- $7^4 - 1 = 2\,400 = 15 \cdot 160$;
- $8^4 - 1 = 4\,095 = 15 \cdot 273$;
- $11^4 - 1 = 1\,4640 = 15 \cdot 976$;
- $13^4 - 1 = 28\,560 = 15 \cdot 1\,904$;
- $14^4 - 1 = 38\,415 = 15 \cdot 2\,561$;

są całkowitymi wielokrotnościami 15. Podstawy 3, 5, 6, 9, 10 i 12 nie mają takiej własności, ale ich warunek z zadania nie dotyczy, gdyż nie są względnie pierwsze z 15. Warto też zauważyć, że liczby mniejsze od 4 nie nadają się na b – choćby dlatego, że $2^1 - 1 = 1$, $2^2 - 1 = 3$ ani $2^3 - 1 = 7$ nie dzielą się przez 15.

Drugi przykład składa się z dwóch testów. Zgodnie z oczekiwanymi wyjściami, $247 = 13 \cdot 19$ oraz $713 = 23 \cdot 31$. Liczba b w obu tych testach spełnia warunek z zadania – Bajtek wracał z lasu na tyle wolno, że sprawdził to w pamięci. Wspominał, że jak ktoś mu nie wierzy, to może to sprawdzić samemu – nie stanowi to jednak wprost części niniejszego zadania.

58.2 Rozwiązanie

Zadanie dotyczy teorii liczb [4][17]. Pozornie dotyczy ono faktoryzacji liczby a, przy której liczba b nie byłaby potrzebna. Testów w zestawie jest jednak bardzo dużo (do 1 000 000), natomiast liczba a jest raczej-duża (do 1 000 000 000, czyli do standardowej „dużej liczby" mieszczącej się w 32 bitach ze znakiem). Przeprowadzenie odpowiednio szybkiej faktoryzacji przy takich warunkach może stanowić problem i zazwyczaj będzie oznaczało przekroczenie limitu czasu.

Nie jest to po prostu zadanie o faktoryzacji, gdyż dysponujemy liczbą b. Jest to najmniejsza naturalna liczba dodatnia, spełniająca warunek:

$$\forall_{x\in\{1,2,\ldots,a-1\},\ \mathrm{NWD}(x,a)=1} \quad x^b \equiv 1 \pmod{a}.$$

Warunek ten trudno w ogóle rozpatrywać nie znając liczb p i q, dla których $a = p \cdot q$. Zresztą odkrycie takich x, dla których $\mathrm{NWD}(x,\ a) \neq 1$, od razu prowadziłoby do rozwiązania. Opisujących to notatek Bajtek jednak nie zachował.

Musimy zatem zauważyć, czym jest taka liczba b. Otóż nasz warunek przedstawiony wyżej oznacza, że b musi być podzielne przez $(p-1)$ oraz przez $(q-1)$[1]. Skoro b jest najmniejszą taką liczbą, to:

$$b = \mathrm{NWW}(p-1,\ q-1).$$

Bez straty dla ogólności uporządkujmy, że $p \leq q$. Przydatne jest wyróżnienie dwóch przypadków:

- Jeśli $b = q-1$, to dalsze rozważania nie są nawet potrzebne. Aby wyeliminować ten przypadek, wystarczy przed dalszymi obliczeniami sprawdzić, czy $b+1$ nie jest nietrywialnym dzielnikiem liczby a – jeśli tak, to $b+1$ oraz $\frac{a}{b+1}$ to szukane liczby pierwsze.
- Jeśli $b > q-1$, to mamy okazję do ciekawszych rozważań. Liczba b i tak musi być wielokrotnością $q-1$, więc otrzymujemy $b \geq 2(q-1)$. Ponadto, skoro $b \neq q-1$, to $p \neq q$. Liczby p i q są nieparzyste, więc $p \leq q-2$.
  Największa wspólna wielokrotność pary liczb jest zawsze dzielnikiem ich iloczynu, więc dla pewnej liczby naturalnej k możemy zapisać, że:
  $$a > (p-1)(q-1) = k \cdot b.$$
  Jednocześnie:
  $$(k+1)\cdot b = (p-1)(q-1) + b \geq (p-1)(q-1) + 2(q-1) = pq + q - p - 1 \geq pq + 1 = a + 1.$$
  Zatem $k \cdot b$ jest największą wielokrotnością liczby b, która jest mniejsza lub równa a. Zatem $k = \lfloor \frac{a}{b} \rfloor$ oraz:
  $$(p-1)(q-1) = \left\lfloor \frac{a}{b} \right\rfloor \cdot b.$$
  Alternatywnie, możemy to samo spostrzeżenie przedstawić[2] jako:
  $$a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b = p + q - 1,$$
  $$p + q = a \,\%\, b + 1.$$

[1] W ogólności powiedzielibyśmy co najwyżej, że b to tocjent Carmichaela z liczby a. W tym zadaniu jednak wiemy, że a jest iloczynem dwóch nieparzystych liczb pierwszych.

[2] W przedstawionych tu wzorach zastosowano operator $\%$ tak, jak w językach C++ i Python.

Odkrycie liczb p, q pozostaje już drobiazgiem, skoro znamy ich iloczyn (jest to liczba a) oraz sumę. Rozwiązując równanie kwadratowe lub stosując znany wynik, otrzymujemy:

$$p = \frac{1}{2}\left(a \,\%\, b + 1 - \sqrt{(a \,\%\, b + 1)^2 - 4a}\right),$$
$$q = \frac{1}{2}\left(a \,\%\, b + 1 + \sqrt{(a \,\%\, b + 1)^2 - 4a}\right).$$

Przy programowaniu, warto uważać na typy danych, gdy oblicza się występujące we wzorze kwadraty (wymaga to zastosowania liczb 64-bitowych) oraz składniki $4a$ (liczby 64-bitowe lub bez znaku). Obliczanie pierwiastka kwadratowego jest wystarczająco dokładne, jeśli odniesiemy się do wersji z biblioteki matematycznej, która działa na 52 bitach mantysy (`double` w C++, `float` w Pythonie – funkcje `sqrt`/`math.sqrt` z tych języków domyślnie działają na właściwych typach danych).

Rozwiązanie wykorzystujące wyprowadzony wzór ma złożoność $O(1)$ dla każdego testu. Podobną złożoność ma samo wczytywanie danych / wypisywanie wyników, więc jest to algorytm optymalny.

Lektura tego wyprowadzenia może stwarzać pozory, że zadanie jest bardzo skomplikowane. Jednak po rozpoznaniu, że b jest podzielne przez $p - 1$ oraz $q - 1$, odtworzenie przedstawionej serii oszacowań ma szansę być całkiem przystępne intuicyjnie. Przedstawione zależności dużo łatwiej jest zauważyć, niż uzasadnić.

58.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Zaginione_liczby_pierwsze/solution.py.

```python
# -*- coding: utf-8 -*-

import math
import sys


def solve_test() -> None:
    """Rozwiąż pojedynczy przypadek testowy."""
    word: str
    task_a: int
    task_b: int
    # Uwaga: sys.stdin.readline() nie przejmuje się kodowaniem znaków,
    # przez co może być nawet kilka razy szybsze, niż input().
    task_a, task_b = (int(word) for word in sys.stdin.readline().split())
    ans_p: int
    ans_q: int
    if task_a % (task_b + 1) == 0:
        # Przypadek I: (p - 1) jest dzielnikiem (q - 1)
        ans_p = task_b + 1
        ans_q = task_a // ans_p
        if ans_p > ans_q:
            ans_p, ans_q = ans_q, ans_p
    else:
        # Przypadek II: znamy iloczyn i sumę zaginionych liczb
        pq_sum: int = task_a % task_b + 1
        pq_diff: int = int(math.sqrt(pq_sum * pq_sum - 4 * task_a) + 0.5)
        ans_p = (pq_sum - pq_diff) // 2
        ans_q = ans_p + pq_diff
    sys.stdout.write(f'{ans_p} {ans_q}\n')


def main() -> None:
    """Rozwiąż wszystkie testy."""
    tests: int = int(sys.stdin.readline())  # Liczba testów
    _: int
    for _ in range(tests):
        solve_test()

if __name__ == '__main__':
    main()
```

58.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Zaginione_liczby_pierwsze/solution.cpp.

```cpp
// C++11 program
// -*- coding: utf-8 -*-
#include <cmath>
#include <iostream>
#include <algorithm>
using namespace std;

void solve_test() {
    // Rozwiąż pojedynczy przypadek testowy.
    int task_a, task_b;
    int ans_p, ans_q;
    cin >> task_a >> task_b;
    if(task_a % (task_b + 1) == 0) {
        // Przypadek I: (p - 1) jest dzielnikiem (q - 1)
        ans_p = task_b + 1;
        ans_q = task_a / ans_p;
        if(ans_p > ans_q) {
            swap(ans_p, ans_q);
        }
    } else {
        // Przypadek II: znamy iloczyn i sumę zaginionych liczb
        int pq_sum = task_a % task_b + 1;
        int pq_diff = sqrt(
            pq_sum * int64_t(pq_sum) - 4 * int64_t(task_a)) + 0.5;
        ans_p = (pq_sum - pq_diff) / 2;
        ans_q = ans_p + pq_diff;
    }
    // Uwaga o wydajności: warto stosować '\n', a nie endl, aby nie czyścić bufora.
    cout << ans_p << ' ' << ans_q << '\n';
}

int main() {
    // Rozwiąż wszystkie testy.
    // Ustawienia wydajności strumieni
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int tests;
    cin >> tests;
    for(int i = 0; i < tests; i++) {
        solve_test();
    }
    return 0;
}
```

59 (VI) – Almost square – Gremliny *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

59.1 Task description

Your friend attends computer science classes in another school. She told you about a recent extra credit homework assignment she received there: her teacher secretly chose some natural number k and used a random number generator to pick a positive integer n that has exactly k digits. Then, on the blackboard, he wrote down the value of n^2 – i.e. the square of n. In order to solve the task, the students were to find the value of n.

"Easy enough," you thought. However, tragedy has struck – when your friend returned home, she discovered that the value she had noted down was not a square number at all! She said that one of the digits on the blackboard had been hard to read, and that she must have noted it down wrong. Without the correct number, she cannot finish the assignment!

Task

Perhaps you can help your friend. Of course, you do not want to give her the final answer n – that is her homework, not yours – but you could find and correct the mistake she made. Given the value your friend noted down, find the digit that has to be modified to turn it into a square number.

Input description

The first and only line of standard input contains one integer m ($100 < m < 10^{50\,000}$) – the number your friend noted down, which is not a square and does not start with a zero.

You may assume that the input has been generated exactly as described above – where k was selected by a human, but n was chosen randomly (with a uniform distribution, i.e. all k-digit numbers were equally likely to be chosen), and m was obtained by changing a randomly selected digit of n^2 into a random different digit.

Additionally, you may safely assume that only one correct solution exists.

Output description

The standard output should contain two whole numbers separated by a space – the first indicating at which position the mistake occurred, and the second equal to the digit which should be at that position instead.

Example

For sample input:

```
445
```

the correct output is:

```
3 1
```

whereas for input:

```
159
```

the correct output is:

```
2 6
```

Explanation

If we replace the 3rd digit of 445 with a 1, we obtain a square: $441 = 21^2$.

In the second test, if we replace the 2nd digit of 159 with a 6, we obtain $169 = 13^2$.

59.2 Rozwiązanie

Zadanie polegało na odnalezieniu liczby będącej kwadratem, różniącej się od zadanej liczby jedną cyfrą.

Aby rozwiązać ten problem, rozważmy możliwe sposoby na sprawdzenie, czy pewna zadana liczba a jest kwadratem. Najbardziej oczywistą metodą byłoby obliczenie pierwiastka kwadratowego tej liczby – jeśli ten jest całkowity, to liczba jest kwadratem. Liczby w zadaniu są jednak na tyle duże, że takie podejście jest wykluczone.

Niektórym może być znany fakt, że każdy kwadrat kończy się cyfrą 0, 1, 4, 5, 6, lub 9 – czyli jeżeli a kończy się np. cyfrą 2, to możemy oznajmić, że nie jest kwadratem. Samo w sobie nie pomoże nam to rozwiązać zadania, ale zbadajmy dokładniej, dlaczego tak się dzieje.

Warto zwrócić uwagę na fakt, że to, co nazywamy „ostatnią cyfrą", to jedynie konsekwencja dziesiątkowego systemu zapisu liczb. Bardziej matematycznie możemy powiedzieć, że dla każdej liczby naturalnej n, wyrażenie $(n^2 \bmod 10)$, czyli reszta z dzielenia n^2 przez 10, przyjmuje tylko wartości ze zbioru $\{0, 1, 4, 5, 6, 9\}$. Wówczas nasuwa się pytanie: czy jest to jakaś szczególna cecha liczby 10? Oczywiście, nie. Na przykład, dla liczb naturalnych n, $(n^2 \bmod 7)$ przyjmuje jedynie wartości ze zbioru $\{0, 1, 2, 4\}$.

Wartości, jakie może przyjąć reszta z dzielenia n^2 przez m, noszą miano *reszt kwadratowych modulo* m (ang. *quadratic residue*). Jeśli m jest nieparzystą liczbą pierwszą, to reszt kwadratowych modulo m jest dokładnie $\frac{m+1}{2}$, czyli jest nimi około połowa wszystkich możliwych reszt (dla odpowiednio dużych m).

Wynika z tego, że jeśli wylosujemy z pewnych przedziałów liczbę naturalną a i nieparzystą liczbę pierwszą p, to jest około 50% szans na to, że $(a \bmod p)$ będzie resztą kwadratową modulo p.

Daje nam to zarys probabilistycznego sposobu na sprawdzenie, czy liczba jest kwadratem. Jeżeli a jest kwadratem, to $(a \bmod p)$ będzie resztą kwadratową modulo p niezależnie od wyboru p. Jeżeli nim nie jest, to wielokrotnie wybierając losowo p, prędzej czy później powinniśmy znaleźć wartość taką, że $(a \bmod p)$ nie jest resztą kwadratową modulo p, przy czym prawdopodobieństwo jest po naszej stronie: jeżeli np. wybierzemy 50 różnych p, to szanse na to, że błędnie uznamy a za kwadrat, wyniosą w przybliżeniu $(\frac{1}{2})^{50}$. Dla porównania, jest to mniej, niż wynosi prawdopodobieństwo bezbłędnego odgadnięcia na loterii 6 liczb z 49... dwa razy pod rząd.

Można tu zapytać: czy jest możliwy celowy dobór liczby a takiej, że będzie się opierać naszym testom? Otóż tak: wyobraźmy sobie np. liczbę $A = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot ... \cdot P$, tj. iloczyn wszystkich liczb pierwszych aż do pewnej dużej wartości P. Jeżeli będziemy wybierać tylko p mniejsze lub równe P, to $(A \bmod p)$ zawsze będzie wynosić 0, a 0 jest resztą kwadratową dla każdego p. Szczęśliwie, treść zadania zapewnia nas, że wszystkie pliki wejściowe zostały wygenerowane losowo, a dla liczb wybranych losowo nasz test działa doskonale.

Powróćmy wreszcie do oryginalnego problemu. Rozwiązanie możemy napisać odpowiednio zoptymalizowaną metodą siłową. W każdym kroku wybieramy nową losową liczbę pierwszą p i zapamiętujemy, jakie liczby z zakresu od 0 do $p-1$ są resztami kwadratowymi modulo p. Następnie dla każdej możliwej liczby otrzymanej poprzez zmianę cyfry w zadanej na wejściu liczbie m, obliczamy jej resztę z dzielenia przez p. Jeżeli wartość ta nie jest resztą kwadratową, to dane potencjalne rozwiązanie odrzucamy. Kontynuujemy w ten sposób, aż pozostanie nam tylko jedno potencjalne rozwiązanie – musi ono być prawidłowe.

W jaki sposób w rozsądnym czasie policzyć reszty z dzielenia wielu tak dużych liczb? W rzeczywistości wystarczy policzyć raz wartość $(m \bmod p)$, a następnie korzystać z niej, by w czasie stałym policzyć wartość

reszty po zmianie danej cyfry – zmiana cyfry to wszak nic innego, jak dodanie lub odjęcie odpowiedniej wartości pomnożonej przez poprawnie dobraną potęgę liczby 10.

Otrzymany w ten sposób algorytm *randomizowany* [6] jest tzw. algorytmem *Las Vegas*, co oznacza, że pomimo wykorzystania losowości, jego wynik nigdy nie będzie błędny (ponieważ kontynuuje on pracę, aż pozostanie tylko jedno możliwe rozwiązanie), ale czas jego działania może się różnić w zależności od „szczęścia" – teoretycznie, algorytm taki mógłby działać bardzo długo, ale w praktyce oczekiwany czas uzyskania odpowiedzi jest skończony i stosunkowo krótki. Oczekujemy bowiem, że w każdym kroku odrzucimy około połowę możliwych rozwiązań, a więc liczbą kroków powinno przeważnie być $O(\log n)$, gdzie n jest długością liczby. Każdy krok jest liniowy w n, więc złożonością średnią całego rozwiązania jest $O(n \log n)$.

Oczywiście poszczególne elementy powyższego opisu algorytmu można zaimplementować w różny – mniej lub bardziej wydajny sposób. W zaprezentowanych poniżej przykładowych implementacjach do wygenerowania listy liczb pierwszych zastosowano sito Eratostenesa, a weryfikacja, czy dana liczba jest resztą kwadratową modulo oparta jest zasadniczo na prostym sprawdzeniu wszystkich możliwości i stablicowaniu odpowiedzi. Alternatywną metodą weryfikacji mogłoby być wykorzystanie tzw. *kryterium Eulera*, co pozwoliłoby na zastosowanie większych liczb pierwszych (ale jednocześnie znacząco skomplikowałoby implementację).

59.2.1 Python

Źródło: ./zadania/06_Zadania_obliczeniowe/Almost_square/solution.py.

```
from random import shuffle

def liczby_pierwsze_ponizej(n):
    sito = [True] * n
    wynik = []
    for i in range(2, n):
        if sito[i]:
            wynik.append(i)
            for j in range(i * i, n, i):
                sito[j] = False
    return wynik

def indeks_listy_zawierajacej_true(lista):
    for i, wartosc in enumerate(lista):
        if True in wartosc:
            return i

limit_liczb = 100000
liczby_pierwsze = liczby_pierwsze_ponizej(limit_liczb)
shuffle(liczby_pierwsze)
duza_liczba = list(input())
n = len(duza_liczba)
moze_byc_rozwiazaniem = [[True] * 10 for _ in range(n)]
moze_byc_rozwiazaniem[0][0] = False
for i in range(n):
    duza_liczba[i] = int(duza_liczba[i])
    moze_byc_rozwiazaniem[i][duza_liczba[i]] = False
potencjalne_rozwiazania = n * 9 - 1
```

```
for p in liczby_pierwsze:
    m = p
    while m * p < limit_liczb:
        m *= p
    reszta_kwadratowa_modulo_m = [False] * m
    for i in range(m):
        reszta_kwadratowa_modulo_m[i * i % m] = True
    reszta = 0
    for c in duza_liczba:
        reszta = (reszta * 10 + c) % m
    potega_10 = 1
    for i in reversed(range(n)):
        for j in range(10):
            if moze_byc_rozwiazaniem[i][j]:
                reszta_po_zmianie = (reszta + (j - duza_liczba[i]) * potega_10) % m
                if not reszta_kwadratowa_modulo_m[reszta_po_zmianie]:
                    moze_byc_rozwiazaniem[i][j] = False
                    potencjalne_rozwiazania -= 1
        potega_10 = potega_10 * 10 % m
    if potencjalne_rozwiazania == 1:
        pozycja = indeks_listy_zawierajacej_true(moze_byc_rozwiazaniem)
        print(pozycja + 1, moze_byc_rozwiazaniem[pozycja].index(True))
        break
```

59.2.2 C++

Źródło: ./zadania/06_Zadania_obliczeniowe/Almost_square/solution.cpp.

```cpp
#include <algorithm>
#include <bitset>
#include <iostream>
#include <random>
#include <string>
#include <vector>

std::vector<int> liczby_pierwsze_ponizej(int n) {
  std::vector<bool> sito(n, true);
  std::vector<int> wynik;
  for (long long i = 2; i < n; i++) {
    if (sito[i]) {
      wynik.push_back((int)i);
      for (long long j = i * i; j < n; j += i) {
        sito[j] = false;
      }
    }
  }
  return wynik;
}

int main() {
  const int limit_liczb = 100'000;
  auto liczby_pierwsze = liczby_pierwsze_ponizej(limit_liczb);
  std::random_device dev;
  std::mt19937_64 random(dev());
  std::shuffle(liczby_pierwsze.begin(), liczby_pierwsze.end(), random);
  std::string duza_liczba;
  std::cin >> duza_liczba;
  int n = (int)duza_liczba.size();
  std::vector<std::bitset<10>> moze_byc_rozwiazaniem(n, std::bitset<10>(~0u));
  moze_byc_rozwiazaniem[0][0] = false;
  for (int i = 0; i < n; i++) {
    auto& c = duza_liczba[i];
    c -= '0';
    moze_byc_rozwiazaniem[i][c] = false;
  }
  int potencjalne_rozwiazania = n * 9 - 1;
  for (const auto& p : liczby_pierwsze) {
    long long m = p;
    while (m * p < limit_liczb)
      m *= p;
    std::vector<bool> reszta_kwadratowa_modulo_m(m);
    for (long long i = 0; i < m; i++) {
      reszta_kwadratowa_modulo_m[i * i % m] = true;
    }
    long long reszta = 0;
    for (const auto& c : duza_liczba) {
```

```
      reszta = (reszta * 10 + c) % m;
    }
    long long potega_10 = 1;
    for (int i = n; i--;) {
      for (int j = 0; j < 10; j++) {
        if (moze_byc_rozwiazaniem[i][j]) {
          long long reszta_po_zmianie = ((reszta + (j - duza_liczba[i]) * potega_10) % m + m) % m;
          if (!reszta_kwadratowa_modulo_m[reszta_po_zmianie]) {
            moze_byc_rozwiazaniem[i][j] = false;
            potencjalne_rozwiazania--;
          }
        }
      }
      potega_10 = potega_10 * 10 % m;
    }
    if (potencjalne_rozwiazania == 1) {
      auto it = std::find_if(moze_byc_rozwiazaniem.begin(),
        moze_byc_rozwiazaniem.end(), [](const std::bitset<10>& b) {
        return b.any();
      });
      int pozycja = (int)(it - moze_byc_rozwiazaniem.begin()) + 1;
      int nowa_wartosc = 0;
      while (!(*it)[nowa_wartosc]) {
        nowa_wartosc++;
      }
      std::cout << pozycja << ' ' << nowa_wartosc << '\n';
      return 0;
    }
  }
  return 1;
}
```

Centrum Mistrzostwa Informatycznego

Dział VII

Algorytmy tekstowe

60 (VII) – Wprowadzenie

*Autor: **Filip Turoboś**, **Politechnika Łódzka***

Jednym z najprostszych i najstarszych typów danych są oczywiście dane tekstowe. W formie, w jakiej się z nimi spotykamy najczęściej, mają postać nieuporządkowanego tekstu (bez sformalizowanej struktury), którego przetwarzanie jest nieodłącznym elementem naszej codzienności – również tej programistycznej. Modyfikacja tekstów, przetwarzanie i wyszukiwanie informacji w nich zawartych stanowią istotny element wielu aplikacji używanych na co dzień.

Najbardziej podstawowe operacje, które wykonujemy na tekstach, to ich tworzenie, łączenie, rozdzielanie i wyszukiwanie szczególnych ich fragmentów. Zadania zawarte w tym dziale w dużej części będą polegały właśnie na efektywnym przetwarzaniu danych tekstowych. Operacje na tekstach są kluczowe zwłaszcza w analizie danych, na ogół w kontekście przeszukiwania i filtracji dużych zbiorów informacji przechowywanych w formie tekstowej. Nie będzie więc dużej przesady w stwierdzeniu, że algorytmy tekstowe są jednym z tych obszarów, w których praktyka ściśle łączy się z teorią.

Najbardziej charakterystyczny dla przetwarzania tekstów problem, jakim jest znajdowanie jednego lub wszystkich wystąpień pewnego ciągu znaków w tekście, nazywany jest zwykle problemem wyszukiwania wzorca (ang. *pattern matching*). Przykładowo, załóżmy, że czytamy książkę dotyczącą algorytmiki (jak choćby tę) i zainteresowani jesteśmy znalezieniem wszystkich miejsc, w których poruszane jest zagadnienie sortowania. O ile dysponujemy wersją elektroniczną dokumentu, zazwyczaj w tym celu wystarczy nam wciśnięcie kombinacji klawiszy `Control` i `F`, a następnie wpisanie szukanej frazy, np. *„Sortowanie”*. Wówczas w dokumencie zostają nam wskazane wszystkie miejsca, gdzie podane hasło zostało odnalezione. Często z podobnym problemem spotykamy się w zadaniach algorytmicznych – z tym, że to co dzieje się po wciśnięciu wspomnianej kombinacji klawiszy musimy zaimplementować własnoręcznie, a sam proces wyszukiwania wzorca stanowi serce tworzonego przez nas algorytmu.

Spróbujmy teraz nieco sformalizować sobie nasz problem, a następnie omówić możliwe strategie jego rozwiązania. Zwykle punktem wyjścia jest dla nas tekst o długości n, w którym interesuje nas znalezienie wszystkich wystąpień wzorca (o długości m). Naiwne rozwiązanie polegałoby np. na stworzeniu „okienka” na m znaków, którym trzeba by „skanować” tekst (przesuwając je w każdym kroku o jeden znak) w nadziei, że fragment tekstu zawarty w okienku będzie odpowiadał szukanemu wzorcowi. To rozwiązanie niekoniecznie jest dobrym podejściem – porównanie zawartości okienka z wzorcem ma złożoność $O(m)$, co oznacza, że czas potrzebny na analizę całego tekstu będzie wynosić $O(mn)$.

Początkujący czytelnik może być zaskoczony tym faktem, ale to, że złożoność tego procesu nie wynosi po prostu $O(n)$ wynika z tego, że wzorce mogą się na siebie nakładać. W realistycznym tekście, takim jak ten, jest to dość mało prawdopodobne. Co jednak, gdy autor zadania ułoży złośliwy test, polegający na znalezieniu liczby wystąpień ciągu `aaa...aab` w tekście składającym się z samych liter `a`? Albo gdy wzorców o długości m, które mamy znaleźć, będzie więcej niż jeden? Na takie okazje przydaje się znajomość algorytmów bazujących na tzw. „hashowaniu”, takich jak algorytm *Rabina-Karpa* [2][4][5].

Hash, czyli *funkcja skrótu*, to nic innego jak skrócona informacja o zawartości zmiennej – w naszym wypadku tekstowej – reprezentowana zwykle w postaci pojedynczej liczby. Ideą stojącą za stosowaniem takich funkcji

w kontekście algorytmów tekstowych jest możliwość wstępnego porównania ze sobą dwóch napisów poprzez zestawienie wartości ich hashów ze sobą. Ponieważ hash jest tylko skrótem informacji oryginalnej, więc może się zdarzyć, że dwa zupełnie różne ciągi znakowe będą mieć identyczny hash – podobnie jak mogą istnieć dwie lub więcej osób o takim samym imieniu i nazwisku. Dlatego jeśli po porównaniu hashów okaże się, że są zgodne, trzeba wykonać drugi krok i porównać reprezentowane przez nie ciągi znakowe bezpośrednio (znak po znaku). Z tego względu najważniejszymi cechami dobrej funkcji skrótu są:

1. szybkość obliczania (w tym możliwość wykorzystania poprzednio obliczonych wartości funkcji skrótu);
2. niskie prawdopodobieństwo kolizji (szanse na to, że dwa różne ciągi znaków wygenerują tę samą wartość funkcji skrótu powinny być możliwie jak najmniejsze).

Funkcje skrótu można wykorzystywać nie tylko w celu znajdowania pojedynczego wzorca w tekście lub poszukiwania wielu haseł jednocześnie. Ciekawe zastosowanie tego podejścia możemy znaleźć w zadaniach *By było bezpiecznie* i *By było bezpieczniej* (omówionych dalej w tym dziale), gdzie wykorzystujemy hashe do znajdowania *dwusłów* – struktur nieco przypominających palindromy, w których pierwsza i druga połowa słowa są takie same.

W gruncie rzeczy, algorytmy tekstowe często sprowadzają się do napisania w umiejętny sposób funkcji hashującej. W większości zadań algorytmicznych polega to na napisaniu funkcji hashującej w taki sposób, żeby obliczona wartość skrótu ze znaków $a_1, \ldots, a_m$ była możliwa do wykorzystania w obliczaniu skrótu z następnego „okienka", tj. znaków $a_2, \ldots, a_m, a_{m+1}$. Przykładem takiej funkcji hashującej (wykorzystanej w rozwiązaniu wzorcowym zadania *By było bezpieczniej*) może być funkcja „rolującego" hashowania wielomianowego modulo P (ang. *rolling hash*), gdzie P jest pewną dużą liczbą pierwszą. Wówczas skrót z m-elementowego ciągu znaków (traktowanych jako liczby całkowite $a_1, a_2, \ldots, a_m$) stanowić może wartość poniższego wyrażenia:

$$\left(a_1 \cdot L^{m-1} + a_2 \cdot L^{m-2} + \cdots + a_m\right) \mod P,$$

gdzie L jest liczbą liter w rozpatrywanym alfabecie. Operacja modulo P, czyli obliczanie reszty z dzielenia przez P pozwala nam zapobiegać przekroczeniu określonego zakresu uzyskiwanych wartości (dzięki tej operacji wartość wynikowa na pewno będzie mniejsza niż P). Z drugiej strony, może to prowadzić do kolizji hashy (bo wiele różnych liczb może dawać tę samą resztę z dzielenia przez P).

Rozważmy prosty przykład, w którym alfabet ograniczymy do dziesięciu pierwszych liter:

Litera:	A	B	C	D	E	F	G	H	I	J
Numer:	0	1	2	3	4	5	6	7	8	9

zaś tekst, dla którego będziemy liczyć wartości funkcji skrótu ma postać:

B I E G A J

Załóżmy, że interesuje nas znalezienie w tym tekście ciągu GAJ. Zaczynamy więc od policzenia funkcji skrótu dla tego ciągu, zgodnie z podanym wyżej wzorem. Jego długość to 3, zatem przyjmujemy $m = 3$. Litery G, A, J są reprezentowane jako liczby 6, 0, 9, odpowiednio, a zatem poszukiwany skrót, to:

$$6 \cdot 10^2 + 0 \cdot 10^1 + 9 = 609\ .$$

Pomijamy tu operację modulo (zauważmy, że dla $L = 10$ i $m = 3$ przyjęcie jakiejkolwiek wartości P większej niż 1000 oznacza *de facto* brak konieczności liczenia modulo P). Możemy też zauważyć, że obliczenie wartości

naszej funkcji hashującej sprowadza się w podanym przykładzie do prostego zestawienia numerów kolejnych liter w liczbę trzycyfrową. To dlatego, że rozważamy tu przypadek szczególny, w którym liczba liter alfabetu wynosi dokładnie 10 (to celowy wybór – dla łatwiejszej ilustracji zagadnienia). W przypadku innej długości alfabetu takiej prostej zależności już nie będzie.

Mając obliczony hash szukanego wzorca powinniśmy teraz analogicznie policzyć hash dla każdego możliwego spójnego podciągu o długości 3 w naszym tekście (czyli dla każdych trzech kolejnych liter):

1. B(1), I(8), E(4):
   
   $1 \cdot 10^2 + 8 \cdot 10^1 + 4 = 184$;
2. I(8), E(4), G(6):
   
   $8 \cdot 10^2 + 4 \cdot 10^1 + 6 = 846$;
3. E(4), G(6), A(0):
   
   $4 \cdot 10^2 + 6 \cdot 10^1 + 0 = 460$;
4. G(6), A(0), J(9):
   
   $6 \cdot 10^2 + 0 \cdot 10^1 + 9 = 609$.

Najistotniejszą obserwacją jest tu zauważenie, że nie ma potrzeby liczyć wszystkich powyższych funkcji skrótu z definicji. Chcąc policzyć hash danego trzyelementowego ciągu, wystarczy wziąć hash poprzedniego, odjąć od niego wartość pierwszej litery pomnożoną przez L^{m-1}, co w naszym uproszczonym przypadku odpowiada usunięciu pierwszej (najstarszej) cyfry, a następnie pomnożyć wynik przez L, co odpowiada u nas przesunięciu o jedną cyfrę w lewo. Na koniec dodajemy liczbę reprezentującą nową literę, co w naszym przypadku oznacza, że odpowiadająca jej cyfra pojawi się na ostatniej (najmłodszej) pozycji.

Procedura ta wymaga wykonania dwóch mnożeń, dodawania i odejmowania (modulo P), więc na pierwszy rzut oka nic dzięki niej nie zyskujemy. Zauważmy jednak, że tyle samo operacji arytmetycznych będziemy musieli wykonać podczas wyszukiwania ciągów o dowolnej długości (złożoność stała ze względu na m), podczas gdy liczenie funkcji skrótu bezpośrednio z definicji dla dłuższych ciągów wymagałoby proporcjonalnie dłuższych obliczeń (złożoność liniowa ze względu na m). To już jest istotny zysk, decydujący o praktycznej przydatności algorytmu Rabina-Karpa, zwłaszcza w przypadku wyszukiwania dłuższych ciągów w tekście.

Na koniec, po obliczeniu w przedstawiony sposób wartości funkcji skrótu dla wszystkich kolejnych spójnych podciągów m-elementowych (tu: o długości 3), wystarczy tylko sprawdzić, czy któryś z nich nie jest równy skrótowi ciągu, którego szukamy (tu: 609). Sprawdzenie to wykonamy oczywiście w czasie liniowym ze względu na n. Zgodnie z tym co zauważyliśmy wcześniej, jeśli nasza funkcja hasująca dopuszcza możliwość występowania kolizji, wówczas dla każdego pasującego skrótu powinniśmy jeszcze dodatkowo dokonać porównania bezpośredniego (znak po znaku).

Przykładowa implementacja algorytmu obliczania funkcji skrótu w języku Python – przy założeniu, że zmienna `s` zawiera hashowany tekst, zaś `m` oznacza długość wzorca (nie dłuższego niż cały tekst!) – może wyglądać następująco:

```
def hash_function(s,m, P =  2015177, alphabet_size = 256):
    hashtable = []
    pol = pow(alphabet_size,m-1,P)
```

```
    myHash = 0
    #Obliczenie pierwszego skrótu
    for k in range(m):
        myHash= (myHash*alphabet_size+ ord(s[k])) % P
        hashtable.append(myHash)

    #Obliczanie kolejnych wartości funkcji hashującej dla przesuniętego "okna"
    #hashującego, z wykorzystaniem dotychczas obliczonych wartości
    for k in range(len(s)-m):
        myHash = (alphabet_size*(myHash - ord(s[k])*pol) + ord(s[k+m]))%P
        if(myHash<0):
            myHash+=P
        hashtable.append(myHash);
    return hashtable
```

W powyższej implementacji możemy zauważyć dwa argumenty o wartościach domyślnych. Pierwszym z nich jest `P`, liczba pierwsza względem której będziemy przeprowadzać operacje modulo dla naszej funkcji skrótu tak, by jej wartości nie wkroczyły w zakres dużych liczb całkowitych. Unikamy w ten sposób spowolnienia operacji potrzebnych do wyliczenia wartości funkcji skrótu. Z kolei `alphabet_size` jest zmienną opisującą rozmiar „alfabetu". W przypadku operowania tylko na literach alfabetu łacińskiego możemy w ogóle pominąć kwestię tworzenia własnej konwersji liter na liczby i skorzystać z funkcji `ord` lub rzutowania zmiennych typu `char` na liczby całkowite, zaoszczędzając w ten sposób trochę czasu na pisanie kodu. Wówczas rozmiar alfabetu będzie wynosić 256, co zostało wprowadzone jako domyślna wartość tego argumentu.

Algorytm Rabina-Karpa, wykorzystujący omówiony wyżej mechanizm hashowania może stanowić pewnego rodzaju punkt odniesienia w wielu zadaniach (nie tylko omawianych w ramach tej książki). Spróbujmy zatem zaprezentować go tutaj w postaci bardziej formalnej zakładając, że dysponujemy już funkcją skrótu – przykładowo tą, którą omawialiśmy powyżej. Wówczas algorytm Rabina-Karpa sprowadza się do wykonania następujących kroków:

1. Tworzymy pustą tablicę $Found$, której elementami będą pozycje znalezionych wystąpień wzorca;
2. Obliczamy wartość skrótu dla poszukiwanego wzorca W (o długości m), oznaczając ją przez H_{pat};
3. Hashujemy całość tekstu T (o długości n) wykorzystując okno rozmiaru m, otrzymując $(n - m + 1)$-elementową tablicę $HTable$;
4. Dla każdego elementu tablicy $HTable$ o indeksie i, dla którego $HTable[i]$ jest równe H_{pat}:
   - Dokonujemy porównania element po elemencie wzorca W z fragmentem tekstu T o długości m, zaczynającego się od pozycji i.
   - Jeżeli udało się znaleźć wzorzec, zapamiętujemy wartość i w tablicy $Found$;
5. Zwracamy tablicę $Found$.

Szybkość algorytmu Rabina-Karpa wynika z małego prawdopodobieństwa zajścia równości, o której mowa w czwartym punkcie powyższego algorytmu. Warto jednak zauważyć, że jeżeli spodziewamy się bardzo dużej ilości nachodzących na siebie wzorców w tekście (przykładowo, wyszukiwanie ciągu liter '`aaa...a`' w tekście o równie ambitnej, choć dłuższej treści '`aaa...a`'), to algorytm Rabina-Karpa okaże się wolniejszy niż zwykłe przeszukiwanie metodą siłową (*brute-force*)! Szczęśliwie, takie sytuacje nie występują zbyt często „w naturze",

a również i w zadaniach algorytmicznych zdarza się to raczej sporadycznie.

W tym dziale przygotowaliśmy kilka zadań, w których należało się wykazać przede wszystkim pomysłowością w modyfikacji znanych algorytmów tekstowych, a często również umiejętnym zastosowaniem kombinatoryki. Przykładami takich zadań mogą być *Reverse Code* i *Scrabbits*. Podobnie, nieco praktyki kombinatorycznej wymagają także zadania *Crossout Cipher* i *Wisielczy Humor*.

Na koniec należałoby podzielić się jeszcze pewnymi refleksjami o naturze dość ogólnej. O ile język C++ dysponuje oddzielnymi typami dla pojedynczego znaku i dla ciągu znakowego, o tyle w Pythonie takiego rozróżnienia nie ma – znak traktowany jest jako jednoelementowy napis (typ: `string`). W C++ zmienne typu `string` są modyfikowalne. Aby zmienić pojedynczy znak w zmiennej tekstowej, możemy odwołać się do niej poprzez zwyczajną indeksację. Python natomiast nie umożliwia modyfikacji `string`ów – chcąc uzyskać nowy tekst na podstawie starego, konieczne jest utworzenie nowego obiektu typu `string`. W niektórych sytuacjach może być więc konieczne przechowywanie tekstu jako tablicy pojedynczych znaków, co jest na szczęście łatwe do uzyskania przy pomocy metody `split`.

Znakomita większość języków programowania, w tym C++ i Python, domyślnie rozróżnia wielkie i małe litery[1] – należy mieć ten fakt na uwadze, wyszukując wzorzec w sformatowanym tekście, zawierającym zaczynające się z wielkiej litery zdania, imiona czy choćby interpunkcję (ważne przy wyszukiwaniu kilkuwyrazowych wzorców). Pracę z takim tekstem warto więc zacząć od usunięcia wielkich liter (np. w Pythonie istnieje dedykowana do tego celu funkcja `lower`, która konwertuje wielkie litery na małe), a często również – od pozbycia się ewentualnych znaków diakrytycznych, takich jak polskie litery „ą”, „ę”, itd. Jeżeli mamy taką możliwość, to wygodne jest w tym celu wykorzystywanie tzw. *wyrażeń regularnych* (ang. *regular expressions*). Pozwalają one na szybkie wychwytywanie w tekście całych rodzin wzorców. Pomimo, że wyuczenie się korzystania z nich wymaga zainwestowania w to nieco czasu, inwestycja ta na pewno szybko się zwróci, z uwagi na wszechobecność zadań związanych z przetwarzaniem tekstu w codziennej pracy zarówno programisty, jak i „zwykłego” użytkownika komputera.

W zadaniach związanych z operowaniem na danych tekstowych, bardzo pomocna bywa też znajomość pewnych struktur grafowych (p. następny dział), w szczególności drzew. Przykładowo, struktura nazywana *drzewem trie* [2][7] pozwala na szybkie zidentyfikowanie, czy w przeanalizowanym tekście choć raz wystąpiło poszukiwane słowo lub na wypisanie wszystkich słów o podanym prefiksie. Oczywiście stosowanie takich struktur ma sens praktycznie tylko wtedy, gdy będziemy musieli zmierzyć się z dużą liczbą zapytań. W przeciwnym razie analiza tekstu *ad hoc* może okazać się zdecydowanie szybsza.

Temat algorytmów tekstowych jest zagadnieniem niezwykle rozległym, którego nie będziemy tu omawiać szczegółowo. Warto jednak wiedzieć, że oprócz algorytmu Rabina-Karpa istnieje wiele innych skutecznych algorytmów pozwalających na szybkie wyszukiwanie wzorców w tekście (np. algorytm KMP: *Knutha-Morrisa-Pratta* [4][2][17]), czy na realizację bardziej szczegółowych zadań, takich jak wyszukiwanie palindromów (np. algorytm *Manachera* [14]) czy określanie podobieństwa tekstów (*odległość edycyjna*). Zainteresowanego czytelnika zachęcamy do sięgnięcia do literatury [4][2] i... oczywiście do zapoznania się z naszymi zadaniami oraz opisami ich rozwiązań przedstawionymi w dalszej części niniejszego działu.

[1] Scratch jest tu niestety wyjątkiem, jak zauważyliśmy we wprowadzeniu do działu II.

61 (VII) – Crossout Cipher – Skrzaty *(zad. w jęz. angielskim)*

Autorzy: Jan Filipowicz, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: Scratch

61.1 Task description

You and your friend invented a special code to pass secret messages to each other during classes. You chose to call it Crossout Cipher because of the way that you can decode a secret message. Say, for example, that the coded message you receive is "`HWelloldor`". In the first step, you will read every second letter of the code (starting from the first): "`H e l l o`" and cross these letters out, leaving only "`Wlodr`". Then you repeat the process with the remaining part – reading "`W o r`". and leaving only "`ld`". The process continues until no letters are left, in this case revealing the message to be "`HelloWorld`".

Task

Recently your friend decided to challenge you by sending you a very long secret message. Decoding it all on paper would be a waste of time, but you want to prove that you are up to the challenge, so you have to write a program that will do it for you.

Start from a new, empty Scratch project and create two lists with names:

- `DANE`
- `WYNIKI`

The list `DANE` should be manually filled with example input data, while the list `WYNIKI` is a place where your program should output the obtained results.

Input description

The input list `DANE` represents the secret code you received and it contains n elements (where $1 \leq n \leq 200\,000$). Each element is just one letter of the English alphabet (no spaces allowed).

Output description

After execution of your program, the list `WYNIKI` should contain the decoded message. It should be represented in the same format as in the case of the input list `DANE` (n elements, each containing a single letter of the English alphabet).

Example

In the example, the input list `DANE` contains a secret message. After decoding, the program writes the result to the output list `WYNIKI`.

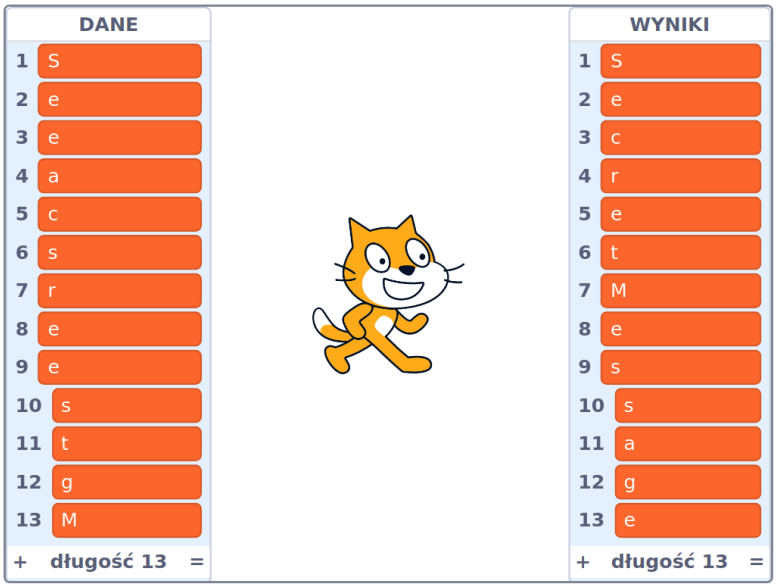
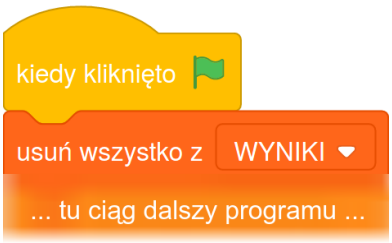


Figure 61.1: An example of the input and the correct output data

Remarks

Please remember, that the list `DANE` should be filled-in manually, by typing the input data with the keyboard. It is a good idea to test the program also with input values other than these in the examples.

The content of the list `WYNIKI` should always be initially removed. Your program should therefore start as demonstrated below:



61.2 Rozwiązanie

Zadanie nie zawiera większych problemów algorytmicznych i można je rozwiązać bezpośrednio, po prostu „skreślając” co drugi znak (tzn. usuwając go z listy `DANE` i dodając do listy `WYNIKI`):

Rysunek 61.2: Crossout Cipher – rozwiązanie naiwne

Zauważmy, że pomimo zmiany wartości `i` tylko o jeden w każdym kroku powyższego skryptu, w istocie „przeskakujemy” co dwa elementy, co wynika z obecności operacji `usuń`.

To rozwiązanie znajdziesz w pliku:

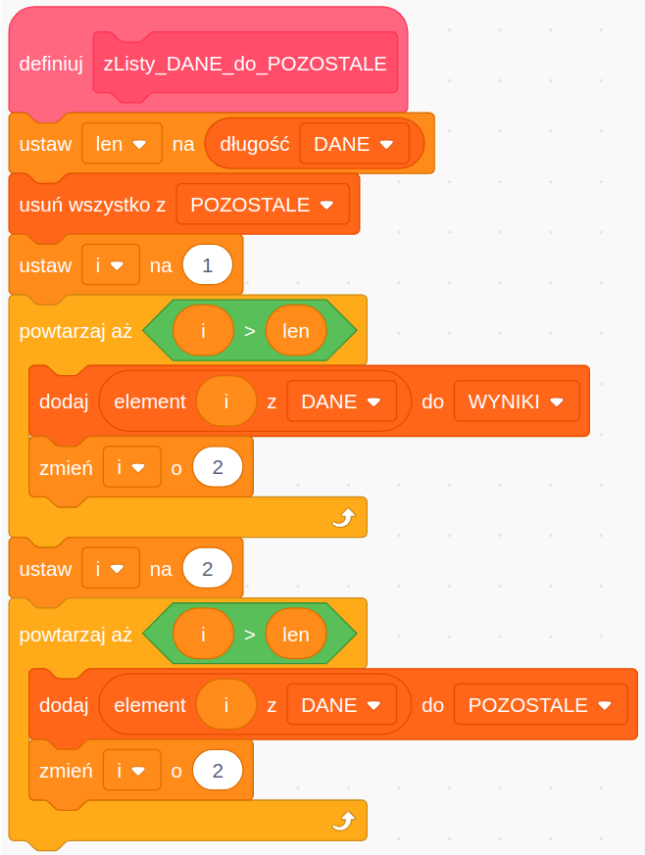
./zadania/07_Algorytmy_tekstowe/Crossout_Cipher_Skrzaty/solution_naive.sb3.

Okazuje się jednak, że – pomimo swojej prostoty – zadanie to wymaga przemyślenia pewnych szczegółów implementacyjnych, mogących mieć znaczący wpływ na czas wykonania skryptu. W szczególności, usuwanie lub wstawianie elementów na listę w Scratchu jest realizowane najszybciej, jeśli robimy to na końcu listy. W przedstawionym rozwiązaniu wąskim gardłem jest usuwanie elementów z różnych pozycji listy `DANE`, w tym z samego jej początku (co powoduje konieczność „przesuwania w górę” wszystkich elementów następnych).

Jak można by tego uniknąć? Zamiast fizycznie usuwać elementy z listy `DANE`, moglibyśmy przepisać elementy nieskreślone do osobnej listy (nazwijmy ją: `POZOSTALE`). Listę tę w kolejnym powtórzeniu algorytmu możemy przetwarzać tak samo jak listę `DANE`, przepisując nieskreślone elementy do jeszcze jednej listy (np. `POZOSTALE_2`) i tak dalej.

Przykład bloku realizującego tę funkcjonalność (`zListy_DANE_do_POZOSTALE`) przedstawiono na następnej stronie. Jak widać, elementy nr 1, 3, 5,... przepisujemy tu do listy `WYNIKI` (nie usuwając ich fizycznie z listy `DANE`), a elementy nr 2, 4, 6,... do listy `POZOSTALE`. W obu przypadkach elementy dodawane są na końcu (instrukcją `dodaj`), co jest realizowane bez zbędnych narzutów czasowych, a zawartość listy `DANE` nie jest modyfikowana w żaden sposób.

Jedynym problemem w tym podejściu jest konieczność tworzenia nowej listy przy każdym powtórzeniu algo-



Rysunek 61.3: Crossout Cipher – rozwiązanie szybkie, bez usuwania elementów

rytmu. Możemy ten problem jednak dość łatwo obejść, wykorzystując – zamiast kolejnej listy `POZOSTALE_2` – z powrotem listę `DANE`. Nasz algorytm może w kolejnych krokach po prostu „przerzucać" nieskreślone dane na zmianę między listą `DANE`, a `POZOSTALE`:

Rysunek 61.4: Crossout Cipher – rozwiązanie szybkie (pętla główna)

Zauważmy tylko, że z uwagi na ograniczenia Scratcha (brak możliwości tworzenia wskaźników lub referencji do listy) musimy zaimplementować drugi, bliźniaczy blok `zListy_POZOSTALE_do_DANE`, przepisujący dane w odwrotnym kierunku.

Kompletne rozwiązanie oparte na tym podejściu znajdziesz w pliku:

./zadania/07_Algorytmy_tekstowe/Crossout_Cipher_Skrzaty/solution.sb3.

62 (VII) – Crossout Cipher – Gnomy *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

62.1 Task description

You and your friend invented a special code to pass secret messages to each other during classes. You chose to call it Crossout Cipher because of the way that you can decode a secret message. Say, for example, that the coded message you receive is "`HWelloldor`". In the first step, you will read every second letter of the code (starting from the first): "`H e l l o`" and cross these letters out, leaving only "`Wlodr`". Then you repeat the process with the remaining part – reading "`W o r`". and leaving only "`ld`". The process continues until no letters are left, in this case revealing the message to be "`HelloWorld`".

Task

Recently your friend decided to challenge you by sending you a very long secret message. Decoding it all on paper would be a waste of time, but you want to prove that you are up to the challenge, so you have to write a program that will do it for you.

Input description

The first and only line of standard input consists of a non-empty string of up to 2 million letters of the English alphabet (no spaces allowed) – the secret code you received.

Output description

The standard output should contain one line – the decoded message.

Example

For sample input:

```
TohdiussLHoenogeEwxMaTmapoleeDSsheoswcsgYo
```

the correct output is:

```
ThisLongExampleShowsYouHowToDecodeMessages
```

62.2 Rozwiązanie

Sposób rozwiązania problemu jest analogiczny, jak w przypadku wersji dla Skrzatów. W implementacji wykorzystujemy konstrukcje specyficzne dla Pythona i C++, co pozwala nam – szczególnie w przypadku Pythona – na bardzo zwięzły zapis algorytmu.

62.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/Crossout_Cipher_Gnomy/solution.py.

```
szyfr = input()
wynik = ""
while szyfr:
    wynik += szyfr[0::2]
    szyfr = szyfr[1::2]
print(wynik)
```

62.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/Crossout_Cipher_Gnomy/solution.cpp.

```
#include <iostream>
#include <string>

int main() {
  std::string szyfr;
  std::cin >> szyfr;
  while (!szyfr.empty()) {
    int n = szyfr.size();
    std::string pozostale_litery;
    for (int i = 0; i < n; i += 2) {
      std::cout << szyfr[i];
    }
    for (int i = 1; i < n; i += 2) {
      pozostale_litery += szyfr[i];
    }
    szyfr = pozostale_litery;
  }
  std::cout << '\n';
  return 0;
}
```

63 (VII) – Scrabbits – Gnomy

*Autor: **Filip Turoboś**, **Politechnika Łódzka***

*Kategoria: **Gnomy (II edycja – liga algorytmiczna)***

*Język programowania: **C++/Python***

63.1 Treść zadania

Bajten wraz z kolegami i koleżankami spędzają długie godziny po zajęciach na grze w Scrabbits^{TM}. Gra ta do złudzenia przypomina wiele innych gier planszowych, polegających na układaniu wyrazów z oznaczonych literami kwadratów, przypominających połówki kostek domina.

Scrabbits^{TM} okazało się być wciągające do tego stopnia, że Bajten myśli o dotychczasowych rozgrywkach jeszcze długo po powrocie do domu. Rozważa, czy w danej sytuacji był w stanie zagrać lepiej, a przede wszystkim – jakie słowa mógł ułożyć z kombinacji liter, którą miał do dyspozycji.

Twoim zadaniem jest pomóc Bajtenowi w jego rozważaniach i określić, które wyrazy mógł ułożyć, dysponując danym zestawem kostek Scrabbits^{TM}.

Zadanie

Bajten, leżąc na łóżku i próbując zasnąć, rozważa T sytuacji z gry w Scrabbits^{TM}, z którymi zetknął się danego dnia. W każdej z tych sytuacji dysponował pewną liczbą n płytek z literami.

Dla danego zestawu płytek Bajten zapisywał na kartce listę k słów, które przychodziły mu do głowy. Następnie chciał zweryfikować, czy możliwe było utworzenie każdego wypisanego słowa przy danym zestawie posiadanych liter. Ponieważ Bajten nie był w stanie zapamiętać układu liter na planszy, w swoich rozważaniach pomijał problem możliwości ułożenia danego słowa na planszy.

Pomóż Bajtenowi – dla zadanej listy k różnych słów zweryfikuj, które z nich były możliwe do ułożenia przy posiadanym przez Bajtena zestawie płytek Scrabbits^{TM}.

Opis wejścia

Pierwszy wiersz standardowego wejścia składa się z pojedynczej liczby $1 \leq T \leq 50$, opisującej liczbę przypadków testowych.

Dalej następuje T opisów testów. Pierwsza linijka opisu testu zawiera dwie liczby naturalne: n oraz k (gdzie $4 \leq n \leq 10\,000$; $k \leq 10\,000$), opisujące odpowiednio liczbę płytek Scrabbits^{TM}, znajdujących się w posiadaniu Bajtena oraz liczbę rozpatrywanych przez niego słów do ułożenia. Kolejna linia testu składa się z ciągu n liter oddzielonych spacjami, stanowiących opis poszczególnych kostek. Każda płytka jest opisana pojedynczą, wielką literą alfabetu łacińskiego.

W kolejnych k wierszach znajdziesz po jednym słowie, stanowiącym ciąg małych liter z zakresu od a do z. Możesz założyć, że długość pojedynczego słowa nie przekracza $10\,000$ znaków.

Opis wyjścia

Na wyjściu standardowym, dla każdego przypadku testowego powinno zostać wypisane k linii tekstu. Każda z nich powinna zawierać pojedyncze „TAK” w sytuacji, gdy dane słowo może zostać ułożone z podanego zestawu kostek domina, lub „NIE” w przeciwnym wypadku.

Zestawy odpowiedzi dotyczące poszczególnych przypadków testowych należy oddzielać pustą linią (tak jak zaprezentowane to zostało w poniższym przykładzie).

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
2
4 3
A A L N
ala
ma
lala
5 5
A K N O O
```

```
okon
okno
kanal
ala
oko
```

prawidłową odpowiedzią jest:

```
TAK
NIE
NIE

TAK
TAK
NIE
NIE
TAK
```

Wyjaśnienie przykładu

W pierwszym przypadku testowym łatwo zauważyć, że wyraz `ALA` jest możliwy do ułożenia z wykorzystaniem pierwszych trzech płytek Scrabbits^{TM}. Nie jest możliwe ułożenie wyrazu `MA` – wynika to z braku płytki z literą `M`. Wyraz `LALA` też nie jest możliwy do ułożenia – dysponujemy bowiem tylko jedną literą `L`.

W drugim przypadku testowym `OKON`, `OKNO` i `OKO` są możliwe do ułożenia. W obydwu pozostałych słowach, tj. `KANAL` i `ALA` brakuje nam dwóch kostek: jednej z literą `L` i jednej z literą `A`.

63.2 Rozwiązanie

Jest to jedno z bardziej „rozgrzewkowych” zadań – polega zasadniczo na prostej weryfikacji, czy dla podanego zestawu liter jest możliwe ułożenie słów z zadanej listy. Jednak bezpośrednie rozwiązania symulacyjne, próby rozróżniania wielkości liter i stosowanie nieoptymalnych struktur danych mogą tu prowadzić do implementacji o zbyt dużej złożoności obliczeniowej. Zauważmy, że jeśli przeglądając kolejne litery danego słowa, będziemy dla *każdej* z nich przeglądać *wszystkie* litery, które mamy do dyspozycji, to w najgorszym przypadku będzie trzeba wykonać aż $nk = 10\,000 \cdot 10\,000 = 10^8$ operacji.

W poniższym rozwiązaniu najpierw zliczamy ile mamy poszczególnych liter w zestawie oraz w każdym ze słów występujących w zadaniu, czyli technicznie rzecz biorąc – budujemy odpowiednie histogramy (p. dział III), które następnie wystarczy porównać ze sobą. Jeśli liczba wystąpień którejkolwiek litery w dostępnym zestawie jest mniejsza niż liczba wystąpień tej litery w danym słowie, to słowa tego na pewno nie da się ułożyć.

Na koniec, zauważmy, że podejście to jest efektywne z uwagi na oczywisty fakt, że długość zestawu liter (lub długość słowa) może być nieporównywalnie większa niż sama liczba liter (26).

63.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/Scrabbits/solution.py.

```
def compareWordWithAvailableArray(histogram_letters):
    s = input()
    histogram_word = [0 for col in range(26)]
    for letter in s:
        histogram_word[ord(letter) - ord('a')] += 1
    for i in range(26):
        if histogram_word[i] > histogram_letters[i]:
            print("NIE")
            return
    print("TAK")

def test():
    Z = input().strip().split(' ')
    n = int(Z[0])
    k = int(Z[1])
    Z = input().strip().split(' ')
    histogram_letters = [0 for col in range(26)]
    for letter in Z:
        histogram_letters[ord(letter) - ord('A')] += 1
    for i in range(0,k):
        compareWordWithAvailableArray(histogram_letters)

T = int(input())
while T > 0:
    T -= 1
    test()
    print('')
```

63.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/Scrabbits/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

void test(){
  int n,k;
  cin>>n>>k;
  char a;

  //An array to keep the available letters count.
  int letters [26];
  for(int i = 0; i < 26; i++){
    letters[i]=0;
  }

  //Import letters to the array
  for(int i = 0; i < n; i++){
    cin>>a;
    letters[a-'A']++;
  }

  //Letters array for a single word;
  int wordletters[26];
  for(int i =0; i<k;i++){

    for(int j = 0; j < 26; j++){
      wordletters[j]=0;
    }

    string word;
    cin>>word;
    for(int j = 0; j < word.length(); j++){
      wordletters[word[j]-'a']++;
    }

    bool ispossible = true;

    for(int j = 0; j < 26; j++){
      if(wordletters[j]>letters[j]){
        ispossible=false;
        break;
      }
    }
    if(ispossible){
      cout<<"TAK\n";
    }
```

```
    else{
      cout<<"NIE\n";
    }
  }
}


int main(){
//  ios_base::sync_with_stdio(false);
//  cin.tie(NULL);
  int T;
  cin>>T;
  while(T-->0){
    test();
    cout<<'\n';
  }

  return 0;
}
```

64 (VII) – Pierwsza krzyżówka – Gnomy

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

64.1 Treść zadania

Bajtek zainteresował się tworzeniem krzyżówek. Nie do końca rozumie, jak się do tego zabrać, ale ma duże ambicje. Jego plan jest prosty: wybrać dwa słowa, skrzyżować je, a reszta powinna już pójść z górki.

Ten pierwszy krok okazał się prosty: Bajtek wybrał już parę słów – jedno przeznaczone na hasło poziome i jedno pionowe. Niekoniecznie występują one w słowniku i mogą być dłuższe niż jakiekolwiek słowo w języku polskim, ale według Bajtka stanowią doskonały wybór do krzyżówki. Natomiast drugi krok jest dla Bajtka nieprzeniknioną zagadką. Skrzyżować słowa – brzmi łatwo, ale w jaki sposób?

Zadanie

Aby pomóc Bajtkowi, napisz program, który obliczy, na ile sposobów da się skrzyżować wybrane przez niego słowa. Skrzyżowanie słów oznacza tu zapis pierwszego z nich poziomo a drugiego pionowo tak, by współdzieliły jedną literę.

Opis wejścia

Na standardowe wejście podane są dwa wiersze – każdy z nich zawiera jeden ciąg wielkich liter alfabetu łacińskiego (od A do Z, bez polskich znaków). Pierwszy ciąg oznacza słowo wybrane przez Bajtka jako hasło poziome, drugi – pionowe. Każdy z ciągów zawiera przynajmniej dwie litery, ale nie więcej niż 2 000 000.

Opis wyjścia

Na standardowe wyjście należy wypisać jedną liczbę całkowitą – liczbę sposobów, na które można skrzyżować zadane słowa.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
SZKOLNE
MISTRZOSTWA
```

prawidłową odpowiedzią jest:

```
4
```

natomiast dla wejścia:

```
ABCDEF
XYZ
```

prawidłową odpowiedzią jest:

```
0
```

Wyjaśnienie

W pierwszym przykładzie, następujące sposoby skrzyżowania słów są możliwe:

```
M          M          M          M
I          I          I          I
SZKOLNE    S          S          S
T          T          T          T
R          R          R          R
Z          Z        SZKOLNE      Z
O        SZKOLNE      O          O
S          S          S        SZKOLNE
T          T          T          T
W          W          W          W
A          A          A          A
```

W drugim przykładzie, żadna litera nie występuje w obu słowach, a więc nie jest możliwe ich skrzyżowanie.

64.2 Rozwiązanie

Problem ten jest dość zbliżony do poprzedniego („Scrabbits”) i rozwiążemy go w podobny sposób. Naszym zadaniem jest zliczyć możliwe sposoby na skrzyżowanie dwóch zadanych słów tak, jak dzieje się to w krzyżówkach.

Skrzyżowaniem może stać się dowolna para pozycji, na których w obu słowach stoi taka sama litera. Najprostszym rozwiązaniem byłoby zatem zagnieżdżoną pętlą zliczyć wszystkie takie pary. Takie rozwiązanie będzie jednak działać powolnie, bo w czasie kwadratowym.

Aby osiągnąć dużo krótszy czas wykonania, możemy najpierw zliczyć wystąpienia każdej możliwej litery w pierwszym słowie (czyli *de facto* zbudować histogram – p. dział III), a następnie przejść jednokrotnie pętlą po drugim słowie, dla każdej litery dodając stosowną wartość do wyniku. W ten sposób unikamy wielokrotnego przechodzenia po tym samym słowie i otrzymujemy rozwiązanie liniowe.

W C++ do przechowywania liczby wystąpień danych liter najlepiej nadaje się prosta tablica, zaś w języku Python lepiej sprawdza się typ słownikowy (ang. *dictionary*).

W przypadku korzystania z języka C++ należało również pamiętać o zastosowaniu dostatecznie dużego typu danych dla wyniku, np. `long long`, gdyż w największych testach wyniki mogły przekraczać zakres liczbowy typu `int`.

64.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/Pierwsza_krzyzowka/solution.py.

```
a = input()
b = input()
litery_a = {}
for litera in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
    litery_a[litera] = 0
for litera in a:
    litery_a[litera] += 1
wynik = 0
for litera in b:
    wynik += litery_a[litera]
print(wynik)
```

64.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/Pierwsza_krzyzowka/solution.cpp.

```
#include <iostream>
#include <string>

int main() {
  const int liczba_liter = 'Z' - 'A' + 1;
  std::string a, b;
  std::cin >> a >> b;
  int litery_a[liczba_liter] {};
  for (char litera : a) {
    litery_a[litera - 'A']++;
  }
  long long wynik = 0;
  for (char litera : b) {
    wynik += litery_a[litera - 'A'];
  }
  std::cout << wynik << '\n';
  return 0;
}
```

65 (VII) – By było bezpiecznie – Gnomy

Autor: Filip Turoboś, Politechnika Łódzka
Kategoria: Gnomy (II edycja – liga algorytmiczna)
Język programowania: C++/Python

65.1 Treść zadania

W silnie zinformatyzowanym królestwie Bajtocji dużą wagę przywiązuje się do bezpieczeństwa – zwłaszcza w cyberprzestrzeni. Od kilkudziesięciu lat bowiem członkowie stowarzyszenia ZŁO (Złośliwi Łamacze Oprogramowania) bezustannie próbują uzyskać dostęp do kont mailowych zwykłych obywateli Bajtocji. Na szczęście Bajtocjanie nie są pozostawieni sami sobie. Nad ich bezpieczeństwem w Bitonecie czuwa gildia ŻAK (Żartobliwi Amatorzy Kryptografii).

ŻAKowie nie są profesjonalistami, ale dość dobrze wykonują powierzoną im misję. Niedawno jednak zostali zaskoczeni przez nowatorski atak stowarzyszenia ZŁO. Atak ten wykorzystywał pewne specjalne kombinacje znaków w hasłach zwykłych Bajtocjan, które czyniły je podatnymi na złamanie. Kombinacje te nazwano *dwusłowami*. Dwusłowo o długości k to nic innego jak ciąg znaków o długości $2k$, którego pierwsza połowa jest identyczna jak druga. Przykładowo:

- wyraz `dumdum` jest dwusłowem o długości 3, ponieważ jego pierwsza połowa (czyli `dum`) jest identyczna jak druga;
- wyraz `owwo` nie jest dwusłowem, ponieważ jego pierwsza połowa `ow` jest różna od drugiej `wo`;
- wyraz `aa` jest dwusłowem o długości 1.

Trwa wyścig z czasem – ŻAKowie muszą jak najszybciej przebadać wszystkie stosowane przez Bajtocjan hasła by zidentyfikować występujące w nich dwusłowa i w porę ostrzec właścicieli zagrożonych kont mailowych. Pomóż im, opracowując odpowiedni program.

Zadanie

Mimo tego, że członkowie stowarzyszenia ŻAK nie są profesjonalistami, mają bardzo wysokie standardy. Jako osoba z zewnątrz dostaniesz dostęp jedynie do niewielkiej liczby $T \leqslant 10$ haseł Bajtocjan. Każde z nich masz przebadać pod kątem występowania dwusłów o długości nie większej niż pewna, zadana dla danego hasła, maksymalna długość j. Masz gwarancję, że j nie przekroczy $\frac{1}{3}$ długości całego hasła.

Z uwagi na wysokie standardy narzucone przez ŻAKów, Twoim zadaniem jest przygotowanie stosownego raportu dla każdego z badanych przez Ciebie haseł.

Raport dotyczący i-tego hasła ma ustaloną strukturę i składa się z j podraportów zgodnie z formatem przedstawionym na następnej stronie.

```
Raport Generalny Numer i:
Podraport Numer 1:
...
Podraport Numer 2:
...
Podraport Numer 3:
⋮
Podraport Numer j:
...
```

W miejsce wielokropków oczywiście należy podać najwcześniej pojawiające się w badanym haśle dwusłowo o ustalonej długości, równej numerowi podraportu (lub wypisać tajny komunikat „`Konto jest bezpieczne`” w sytuacji, gdy takowe nie istnieje). Innymi słowy – podraport 1 opisuje wynik poszukiwań dwusłów o długości 1, podraport 2 opisuje analogiczny wynik dla dwusłów długości 2 i tak dalej.

Opis wejścia

Pierwszy wiersz standardowego wejścia składa się z pojedynczej liczby T ($1 \leqslant T \leqslant 10$) i opisuje liczbę przypadków testowych, które masz przebadać.

Każda z kolejnych T par linii wejścia opisuje pojedynczy przypadek testowy i zawiera, kolejno:

- pierwsza linia: pojedynczy ciąg wyrazowy składający się z s małych liter od „a” do „z”, bez polskich znaków (trzeba przyznać, że Bajtocjanie muszą wprowadzić solidniejsze reguły zarządzania hasłami). Wiadomo, że $4 \leqslant s \leqslant 9\,000$;
- druga linia: liczba j opisująca zakres długości poszukiwanych w haśle dwusłów, przy czym $j \leqslant \frac{s}{3}$.

Opis wyjścia

Na wyjściu standardowym, dla każdego przypadku testowego powinno się znaleźć $2j + 1$ linii. Pierwszą linią jest wyrażenie „`Raport Generalny Numer i:`”, gdzie i jest numerem przypadku testowego (uwaga, ten napis musi mieć dokładnie taką treść i formatowanie – w szczególności nie używaj żadnych dodatkowych spacji).

W kolejnych liniach powinno się znaleźć j podraportów, w których pojawi się tajny komunikat (gdy dwusłowo danej długości nie zostało znalezione) lub najwcześniej występujące w tekście dwusłowo rozpatrywanej długości.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
1
rabarbarbarar
3
```

prawidłową odpowiedzią jest:

```
Raport Generalny Numer 1:
Podraport Numer 1:
Konto jest bezpieczne
Podraport Numer 2:
arar
Podraport Numer 3:
barbar
```

Wyjaśnienie przykładu

W powyższym przykładzie słowo `rabarbarbarar` nie zawiera dwusłów o długości 1 – żadna litera bowiem nie występuje w tym słowie dwa razy pod rząd.

Rozpatrywane hasło zawiera jednak dokładnie jedno dwusłowo długości 2 – jest nim `arar`.

Hasło to posiada też kilka dwusłów długości 3. Zauważmy bowiem, że zarówno `barbar` jak i `arbarb` czy też `rbarba` są dwusłowami, które pojawiają się w badanym haśle. Z wymienionych tu przypadków jednak to `barbar` pojawia się najwcześniej, dlatego jest uwzględniony w raporcie.

65.2 Rozwiązanie

Aby przedstawić sposób optymalnego rozwiązania tego zadania, przypomnijmy pojęcia występujące w treści zadania. *Dwusłowem* nazywać będziemy ciąg znaków parzystej długości, którego pierwsza połowa jest identyczna jak druga (np. `mama`, `dumdum`, `aa`).

W zadanych hasłach o długości nieprzekraczającej $9\,000$ znaków należy znaleźć *dwusłowa* o długościach nie większych niż pewna wartość j. Wartość ta, zgodnie z treścią zadania, nigdy nie przekroczy $\frac{1}{3}$ długości całego hasła.

Pierwszym, dość naiwnym podejściem do rozwiązania tego problemu mogłoby być wykorzystanie *„ruchomego okienka”*. Polegałoby ono na przyglądaniu się podciągowi hasła o długości $2k$ i porównaniu, znak po znaku, pierwszej i drugiej połowy wycinka. Zastanówmy się jednak, jaką złożonością obliczeniową charakteryzuje się owo rozwiązanie? Dla ustalonej długości dwusłowa wynoszącej $2k$, zweryfikowanie, czy wyselekcjonowany kawałek tekstu jest dwusłowem zajmuje $\mathcal{O}(k)$ czasu. Kandydatów na bycie dwusłowem mamy $n - 2k + 1$ – tyle, ile spójnych podciągów naszego n-literowego hasła o długości $2k$. Zatem uzyskana złożoność wynosi $\mathcal{O}(nk)$. Zastanówmy się, czy nie jesteśmy w stanie tego procesu przeprowadzić w bardziej efektywny sposób. W tym celu wykorzystamy ideę rolującego hashowania, występującą też w klasycznej implementacji algorytmu Rabina-Karpa (p. wprowadzenie do tego działu).

W klasycznym podejściu porównywaliśmy skrót fragmentu tekstu z obliczoną wartością hashu poszukiwanego wzorca. Tym razem jednak nie mamy żadnego konkretnego wzorca – staramy się porównać ze sobą dwa sąsiadujące podciągi. Zatem zamiast dopasowywać hash do hashu wzorca, próbujemy dopasować hash podciągu `s[i..m+i-1]` do hashu podciągu `s[i+m..i+2m-1]`. Warto zauważyć, że testy w tym zadaniu były skonstruowane tak, że rozwiązanie naiwne (przesuwające się okienko) pozwalało uzyskać sensowny czas wykonania w prostych przypadkach, ale było eliminowane na przypadkach specjalnych (np. hasła używające tylko dwóch znaków).

Przypomnijmy, że wyniki wyszukiwań należało podać w specyficznym formacie, zgodnym z opisem zadania:

```
Raport Generalny Numer i:

Podraport Numer 1:

...

Podraport Numer 2:

⋮

Podraport Numer j:

...
```

Teraz już, na szczęście, uzupełnienie podraportów będzie stosunkowo proste, jak zobaczymy w poniższych rozwiązaniach przykładowych.

65.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/By_bylo_bezpiecznie_Gnomy/solution.py.

```
prime =  2015177
alphabet_size = 256

def exactSearch(s, pos, len):
  for j in range(pos,pos+len):
    if(s[j]!=s[j+len]):
      return False
  return True;

def lookForSafewords(s, j):
    hashtable = []
    pol = pow(alphabet_size,j-1,prime)
    myHash = 0
    for k in range(j):
        myHash= (myHash*alphabet_size+ ord(s[k])) % prime
    hashtable.append(myHash)
    for k in range(len(s)-j):
        myHash = (alphabet_size*(myHash - ord(s[k])*pol) + ord(s[k+j]))%prime
        if(myHash<0):
            myHash+=prime
        hashtable.append(myHash);
    ret = "Podraport Numer "+str(j)+":\n"
    for k in range(len(hashtable)-j):
        if(hashtable[k]==hashtable[k+j]):
            if(exactSearch(s,k,j)):
                end = k+2*j
                return ret+s[k:end]
    return ret+"Konto jest bezpieczne"

def test(i):
    S=input()
    K=int(input())
    print("Raport Generalny Numer "+str(i)+":")
    for j in range(1,K+1):
        print(lookForSafewords(S,j))

T=int(input())
for i in range(1,T+1):
    test(i)
```

65.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/By_bylo_bezpiecznie_Gnomy/solution.cpp.

```
#include<iostream>
#include<vector>
#include<string>
#include<stdint.h>

#define prime 20151777
#define alphabet_size 256


using namespace std;

int64_t myPow(int64_t x, int64_t p)
{
  if (p == 0) return 1;
  if (p == 1) return x %prime;

  int64_t tmp = myPow(x, p/2) % prime;
  if (p%2 == 0) return (tmp * tmp) % prime;
  else return (x * ((tmp * tmp)%prime)) % prime;
}

bool exactSearch(string &s, int pos, int len){
  for(int j = pos; j<pos+len; j++){
    if(s[j]!=s[j+len]){
      return false;
    }
  }
  return true;
}

string lookForSafewords(string &s, int j){
  vector<int64_t> hashtable = vector<int64_t>();
  int64_t pol = myPow(alphabet_size,j-1);

  int64_t myHash = 0;
  for(int k = 0; k<j; k++){
    myHash= (myHash*alphabet_size+ s[k]) % prime;
  }
  hashtable.push_back(myHash);
  for(int k=0; k<s.size()-j; k++){
    myHash = (alphabet_size*(myHash - s[k]*pol) + s[k+j])%prime;
    if(myHash<0){
      myHash+=prime;
    }
    hashtable.push_back(myHash);
  }
/*
//Debug purpose only
```

```
  for(int zz = 0; zz <hashtable.size(); zz++){
    cout<<" "<<hashtable[zz];
  }
*/
  string ret = "Podraport Numer "+to_string(j)+":";
  for(int k = 0; k<hashtable.size()-j; k++){
    if(hashtable[k]==hashtable[k+j]){
      if(exactSearch(s,k,j)){
        return ret+"\n"+s.substr(k,2*j)+"\n";
      }
    }
  }
  return ret +"\n"+"Konto jest bezpieczne"+"\n";
}

void test(int i){
  string S;
  cin>>S;
  int K;
  cin>>K;
  cout<<"Raport Generalny Numer "+to_string(i)+":\n";
  for(int j = 1; j<=K; j++){
    cout<<lookForSafewords(S,j);
  }
}

int main(){
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  //Number of cases
  int T;
  cin>>T;
  for(int i = 1; i<=T; i++){
    test(i);
  }
  return 0;
}
```

66 (VII) – By było bezpiecznie – Skrzaty

Autorzy: Filip Turoboś, Politechnika Łódzka, Bartłomiej Stasiak, Politechnika Łódzka
Kategoria: Skrzaty (II edycja – liga algorytmiczna)
Język programowania: Scratch

66.1 Treść zadania

W silnie zinformatyzowanym królestwie Bajtocji dużą wagę przywiązuje się do bezpieczeństwa – zwłaszcza w cyberprzestrzeni. Od kilkudziesięciu lat bowiem członkowie stowarzyszenia ZŁO (Złośliwi Łamacze Oprogramowania) bezustannie próbują uzyskać dostęp do kont mailowych zwykłych obywateli Bajtocji. Na szczęście Bajtocjanie nie są pozostawieni sami sobie. Nad ich bezpieczeństwem w Bitonecie czuwa gildia ŻAK (Żartobliwi Amatorzy Kryptografii).

ŻAKowie nie są profesjonalistami, ale dość dobrze wykonują powierzoną im misję. Niedawno jednak zostali zaskoczeni przez nowatorski atak stowarzyszenia ZŁO. Atak ten wykorzystywał pewne specjalne kombinacje znaków w hasłach zwykłych Bajtocjan, które czyniły je podatnymi na złamanie. Kombinacje te nazwano *dwusłowami*. Dwusłowo o długości k to nic innego jak ciąg znaków o długości $2k$, którego pierwsza połowa jest identyczna jak druga. Przykładowo:

- wyraz `dumdum` jest dwusłowem o długości 3, ponieważ jego pierwsza połowa (czyli `dum`) jest identyczna jak druga;
- wyraz `owwo` nie jest dwusłowem, ponieważ jego pierwsza połowa `ow` jest różna od drugiej `wo`;
- wyraz `aa` jest dwusłowem o długości 1.

Trwa wyścig z czasem – ŻAKowie muszą jak najszybciej przebadać wszystkie stosowane przez Bajtocjan hasła by zidentyfikować występujące w nich dwusłowa i w porę ostrzec właścicieli zagrożonych kont mailowych. Pomóż im, opracowując odpowiedni program.

Mimo tego, że członkowie stowarzyszenia ŻAK nie są profesjonalistami, mają bardzo wysokie standardy. Jako osoba z zewnątrz dostaniesz dostęp jedynie do niewielkiej liczby $T \leqslant 10$ haseł Bajtocjan. Każde z nich masz przebadać pod kątem występowania dwusłów o długości nie większej niż pewna, zadana dla danego hasła, maksymalna długość j. Masz gwarancję, że j nie przekroczy $\frac{1}{3}$ długości całego hasła.

Zadanie

W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać rezultat swojego działania.

Zadanie polega na napisaniu programu, który – dla każdego z badanych przez Ciebie haseł – sporządzi stosowny

raport, zgodnie z wysokimi standardami narzuconymi przez ŻAKów.

Raport dotyczący i-tego hasła ma ustaloną strukturę i składa się z j podraportów zgodnie z formatem:

```
Raport Generalny Numer i
Podraport Numer 1
...
Podraport Numer 2
...
Podraport Numer 3
⋮
Podraport Numer j
...
```

W miejsce wielokropków oczywiście należy podać najwcześniej pojawiające się w badanym haśle dwusłowo o ustalonej długości, równej numerowi podraportu (lub wypisać tajny komunikat „`Konto jest bezpieczne`” w sytuacji, gdy takowe nie istnieje). Innymi słowy – podraport 1 opisuje wynik poszukiwań dwusłów o długości 1, podraport 2 opisuje analogiczny wynik dla dwusłów długości 2 i tak dalej.

Opis wejścia

Pierwszy element listy wejściowej `DANE` to pojedyncza liczba T ($1 \leqslant T \leqslant 10$), opisująca liczbę przypadków testowych, które masz przebadać.

Każda z kolejnych T par elementów na liście `DANE` opisuje pojedynczy przypadek testowy i zawiera, kolejno:

- pierwszy element z pary: pojedynczy ciąg wyrazowy składający się z s małych liter od „a” do „z”, bez polskich znaków (trzeba przyznać, że Bajtocjanie muszą wprowadzić solidniejsze reguły zarządzania hasłami). Wiadomo, że $4 \leqslant s \leqslant 1\,500$;
- drugi element z pary: liczba j opisująca zakres długości poszukiwanych w haśle dwusłów, przy czym $j \leqslant \frac{s}{3}$.

Opis wyjścia

Po zakończeniu Twojego programu na liście `WYNIKI` dla każdego przypadku testowego powinno się znaleźć $2j+1$ elementów. Pierwszym elementem jest wyrażenie „`Raport Generalny Numer i`”, gdzie i jest numerem przypadku testowego (uwaga, ten napis musi mieć dokładnie taką treść i formatowanie – w szczególności nie używaj żadnych dodatkowych spacji).

W kolejnych elementach powinno się znaleźć j podraportów, w których pojawi się tajny komunikat (gdy dwusłowo danej długości nie zostało znalezione) lub najwcześniej występujące w tekście dwusłowo rozpatrywanej długości.

Przykład

Na poniższym rysunku pokazano przykładowe dane wejściowe i wynik działania programu.

Rysunek 66.1: Przykładowe dane wejściowe i wynik działania programu

Wyjaśnienie przykładu

W przedstawionym przykładzie słowo `rabarbarbarar` nie zawiera dwusłów o długości 1 – żadna litera bowiem nie występuje w tym słowie dwa razy pod rząd.

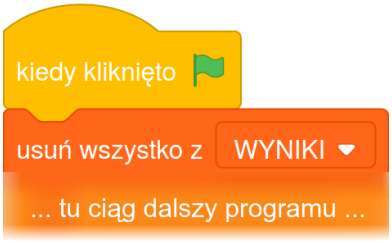
Rozpatrywane hasło zawiera jednak dokładnie jedno dwusłowo długości 2 – jest nim `arar`.

Hasło to posiada też kilka dwusłów długości 3. Zauważmy bowiem, że zarówno `barbar` jak i `arbarb` czy też `rbarba` są dwusłowami, które pojawiają się w badanym haśle. Z wymienionych tu przypadków jednak to `barbar` pojawia się najwcześniej, dlatego jest uwzględniony w raporcie.

Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie.

Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak jak pokazano poniżej:



66.2 Rozwiązanie

Rozwiązanie tego zadania w wersji dla Skrzatów jest uproszczone, bo nie wykorzystuje funkcji hashującej. Do poszukiwania dwusłów zaimplementowany został jedynie blok realizujący funkcjonalność `exactSearch` (z tego względu liczba liter w pojedynczym słowie została tu ograniczona do $1\,500$).

Rozwiązanie to znajdziesz w pliku:

./zadania/07_Algorytmy_tekstowe/By_bylo_bezpiecznie_Skrzaty/solution.sb3.

67 (VII) – By było bezpieczniej – Gremliny

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gremliny (II edycja – liga algorytmiczna)***
*Język programowania: **C++/Python***

67.1 Treść zadania

Cyberprzestrzeń Bajtocji nadal zmaga się ze zmasowanymi atakami cyberprzestępców. Do ataków przyznała się organizacja ZŁO (Złośliwi Łamacze Oprogramowania) podczas swojej oficjalnej konferencji prasowej. W swoim zadufaniu, rzecznik prasowy organizacji podał do wiadomości publicznej, że metoda łamania haseł Bajtocjan wykorzystuje występujące często w tych hasłach *dwusłowa*. W bajtockiej terminologii kryptograficznej *dwusłowo* o długości k, to nic innego jak ciąg znaków o długości $2k$, którego pierwsza połowa jest identyczna jak druga – przykładowo, wyraz `dumdum` jest dwusłowem o długości 3, ponieważ jego pierwsza połowa (czyli `dum`) jest identyczna jak druga.

Członkowie gildii ŻAK (Żartobliwych Amatorów Kryptografii) złapali się za głowy – przecież poinformowali użytkowników Bitonetu by wystrzegać się dwusłów w swoich hasłach! Niestety, nie wszyscy użytkownicy Bitonetu zastosowali się do tych zaleceń, jak widać... i teraz każde hasło trzeba skrupulatnie sprawdzić. Czasu jest bardzo mało, więc hasła należy badać metodami heurystycznymi. Zamiast przeszukiwać całą długość hasła w poszukiwaniu dwusłów dowolnej długości, testujemy hasła na obecność dwusłów o kilku z góry zadanych długościach.

Do tej procedury członkowie gildii ŻAK potrzebują wyspecjalizowanego oprogramowania. Zwrócili się więc z prośbą do Ciebie (jako eksperta!) – pomóż im, przygotowując odpowiedni program.

Zadanie

Mimo tego, że członkowie stowarzyszenia ŻAK nie są profesjonalistami, mają bardzo wysokie standardy. Jako osoba spoza organizacji dostaniesz dostęp tylko do jednego z haseł Bajtocjan, o długości nieprzekraczającej $10\,000$ znaków.

Hasło to należy przebadać pod kątem występowania dwusłów o zadanej długości. Precyzyjniej rzecz ujmując, dostaniesz T zapytań ($1 \leqslant T \leqslant 100\,000$), z których każde składać się będzie z pojedynczej liczby całkowitej k, nieprzekraczającej długości hasła. Dla każdego zapytania, Twój program powinien zwrócić odpowiedź na pytanie „*Ile dwusłów o długości k zawiera badane hasło?*”.

Opis wejścia

Pierwszy wiersz standardowego wejścia składa się z ciągu znakowego o długości s (przy czym $3 \leqslant s \leqslant 10\,000$), opisującego badane hasło. Każdy znak występujący w haśle stanowi małą literę alfabetu łacińskiego (trzeba przyznać, że Bajtocjanie muszą wprowadzić solidniejsze reguły zarządzania hasłami).

Drugi wiersz wejścia składa się z pojedynczej liczby T ($1 \leqslant T \leqslant 100\,000$), która oznacza liczbę zapytań dotyczących występowania w haśle dwusłów o zadanej długości.

Każda z kolejnych T linii wejścia opisuje pojedynczy przypadek testowy i zawiera dokładnie jedną liczbę k, definiującą długość poszukiwanych w haśle dwusłów ($1 \leqslant k \leqslant s$).

Opis wyjścia

Na wyjściu standardowym powinno pojawić się T linii. Każda z nich powinna zawierać pojedynczą liczbę – odpowiedź na zapytanie o liczbę występujących w haśle dwusłów zadanej długości.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
rabarbarbarar
4
3
2
1
6
```

prawidłową odpowiedzią jest:

```
4
1
0
0
```

Wyjaśnienie przykładu

W rozpatrywanym ciągu znaków jesteśmy w stanie znaleźć cztery dwusłowa o długości 3. Dwukrotnie można doszukać się dwusłowa `barbar` (pierwsze jego wystąpienie zaczyna się od trzeciego znaku, drugie od szóstego). Pozostałe dwa dwusłowa o długości 3 to kolejno: `arbarb` oraz `rbarba`. Stąd odpowiedzią na pierwsze z czterech zapytań w przykładzie jest 4.

Rozpatrywane hasło zawiera dokładnie jedno dwusłowo długości 2 – jest nim `arar`.

Hasło nie zawiera natomiast dwusłów o długości 1, ponieważ żadna litera nie występuje w słowie `rabarbarbarar` dwa razy pod rząd. Nie występują w tym haśle również żadne dwusłowa długości 6 – stąd odpowiedzią na każde z ostatnich dwóch zapytań jest 0.

67.2 Rozwiązanie

Rozwiązanie jest analogiczne do rozwiązania problemu ***By było bezpiecznie*** (dla Gnomów) – zamiast wypisywać pierwsze napotkane *dwusłowo*, musimy wyznaczyć liczbę *dwusłów* określonej długości występujących w danym haśle.

67.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/By_bylo_bezpieczniej/solution.py.

```
prime =  2015177
alphabet_size = 256

def exactSearch(s, pos, len):
    for j in range(pos,pos+len):
        if(s[j]!=s[j+len]):
            return False
    return True;

def lookForSafewords(j):
    number =0
    hashtable = []
    pol = pow(alphabet_size,j-1,prime)
    myHash = 0
    for k in range(j):
        myHash= (myHash*alphabet_size+ ord(s[k])) % prime
    hashtable.append(myHash)
    for k in range(len(s)-j):
        myHash = (alphabet_size*(myHash - ord(s[k])*pol) + ord(s[k+j]))%prime
        if(myHash<0):
            myHash+=prime
        hashtable.append(myHash);
    for k in range(len(hashtable)-j):
        if(hashtable[k]==hashtable[k+j]):
            if(exactSearch(s,k,j)):
                number+=1
    return number

def test(i):
    K=int(input().strip())
    if(K>len(V)/2):
        print("0")
    else:
        if(V[K-1]>-1):
            print(V[K-1])
        else:
            V[K-1]=lookForSafewords(K);
            print(V[K-1])

s=input().strip()
```

```
V=[-1]*len(s)
T=int(input().strip())
for i in range(0,T):
    test(i)
```

67.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/By_bylo_bezpieczniej/solution.cpp.

```
#include<iostream>
#include<vector>
#include<string>
#include<stdint.h>

#define prime 2015177
#define alphabet_size 256

using namespace std;

vector<int> V;
string S;

int64_t myPow(int64_t x, int64_t p)
{
  if (p == 0) return 1;
  if (p == 1) return x %prime;

  int64_t tmp = myPow(x, p/2) % prime;
  if (p%2 == 0) return (tmp * tmp) % prime;
  else return (x * ((tmp * tmp)%prime)) % prime;
}

bool exactSearch(string &s, int pos, int len){
  for(int j = pos; j<pos+len; j++){
    if(s[j]!=s[j+len]){
      return false;
    }
  }
  return true;
}

int lookForSafewords(string &s, int j){
  int number =0;
  vector<int64_t> hashtable = vector<int64_t>();
  int64_t pol = myPow(alphabet_size,j-1);
  int64_t myHash = 0;
  for(int k = 0; k<j; k++){
    myHash= (myHash*alphabet_size+ s[k]) % prime;
  }
  hashtable.push_back(myHash);
```

```
  for(int k=0; k<s.size()-j; k++){
    myHash = (alphabet_size*(myHash - s[k]*pol) + s[k+j])%prime;
    if(myHash<0){
      myHash+=prime;
    }
    hashtable.push_back(myHash);
  }
  for(int k = 0; k<hashtable.size()-j; k++){
    if(hashtable[k]==hashtable[k+j]){
      if(exactSearch(s,k,j)){
        number++;
      }
    }
  }
  return number;
}

void test(){
  int K;
  cin>>K;
  if(K>V.size()/2){
    cout<<0<<'\n';
  }
  else{
    if(V[K-1]>-1){
      cout<<V[K-1]<<'\n';
    }
    else{
      V[K-1]=lookForSafewords(S,K);
      cout<<V[K-1]<<'\n';
    }
  }
}

int main(){
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cin>>S;
  V=vector<int>(S.size(),-1);
  int T;
  cin>>T;
  while(T-->0){
    test();
  }
  return 0;
}
```

68 (VII) – Reverse code – Gremliny *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka
Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

68.1 Task description

During your recent trip to the beach, you found a glass bottle washed ashore with a slip of paper inside. At first the message inscribed on it seemed unreadable, but you quickly realized it was written in a rather primitive secret code – sections of text between square brackets ("`[]`") are simply meant to be read backwards, for example "`alg[tiro]hm`" actually means "`algorithm`".

You can decode enough to learn that the message contains information about hidden pirate treasure, but soon the process becomes too tiresome to perform manually, as the square brackets recursively nest within each other multiple times. You decide that this is clearly a task for a computer.

Task

Write a program that will decode the secret message by reversing text between square brackets. The message may contain nested brackets (that is, brackets within brackets, such as "`One[owT[Three[ruoF]]]`"). In this case, innermost brackets take precedence, similar to parentheses in mathematical expressions, e.g. you could decode the aforementioned example like this:

1. `One[owT[Three[ruoF]]]`
2. `One[owT[ThreeFour]]`
3. `One[owTruoFeerhT]`
4. `OneThreeFourTwo`

In order to make your own task slightly easier and less tricky, you have already replaced all whitespaces in the original text with underscores ("`_`") while copying it from the paper version.

Input description

The first and only line of the standard input consists of a non-empty string of up to $2 \cdot 10^6$ characters which may be letters, digits, basic punctuation ("`,.?!'-;:`"), underscores ("`_`") and square brackets ("`[]`"). You can safely assume that all square brackets are paired correctly, i.e. every opening bracket has exactly one closing bracket matching it and vice versa.

Output description

The standard output should contain one line – the decoded secret message without any square brackets.

Example

For sample input:

```
A[W_[y,[]]oh]o[dlr][!]
```

the correct output is:

```
Ahoy,_World!
```

Explanation

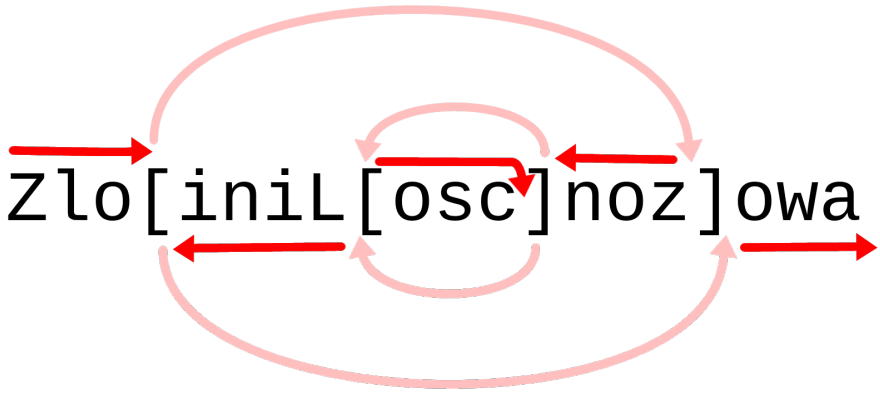
This example contains empty brackets. Of course, an empty string, when reversed, remains empty, so we can simply ignore them. Then, as previously, we can decode this example in stages, first reversing the innermost brackets to obtain "`A[W_,yoh]o[dlr][!]`". Afterwards, there are no longer any nested brackets, so the remainder of the task is trivial.

68.2 Rozwiązanie

W zadaniu należało odwrócić kolejność znaków w tekście między każdą parą nawiasów kwadratowych, przy czym nawiasy mogły być w sobie dowolnie zagnieżdżone.

Zacznijmy od odrzucenia najbardziej oczywistego podejścia, polegającego na faktycznym odwracaniu fragmentów tekstu zaczynając od najgłębszych nawiasów – każde odwrócenie jest kosztowne (złożoność liniowa względem długości tekstu), a i nawiasów może być dużo (również liniowo w długości tekstu), co skutkowałoby złożonością co najmniej kwadratową.

Zamiast tego rozważmy, w jakiej kolejności znaki powinny zostać wypisane na wyjście w przykładowym tekście, „`Zlo[iniL[osc]noz]owa`”.



Rysunek 68.1: Graf przedstawiający kolejność wypisywania znaków wiadomości

Powiedzmy, że chcemy przejść kursorem po łańcuchu według tej kolejności – możemy wówczas zauważyć ciekawe, regularne zachowanie kursora. Za każdym razem, gdy natrafia on na nawias kwadratowy, wykonuje skok do odpowiadającego mu nawiasu i odwraca kierunek iteracji. Nietrudno się przekonać, że jest tak dla każdego możliwego wejścia. To daje nam szkic prostego rozwiązania przy pomocy pętli. Jedyne, co wciąż jest potrzebne do jego wdrożenia, to zdolność wykonania takiego skoku między nawiasami w czasie stałym.

Aby to osiągnąć, możemy uprzednio utworzyć tablicę, która kojarzy ze sobą wzajemnie indeksy odpowiadających sobie nawiasów. Da się tego dokonać przy pomocy pętli po łańcuchu, która napotkawszy znak „`[`”, będzie odkładać jego indeks na stosie, zaś w reakcji na znak „`]`” – zdejmować ostatni indeks ze stosu i dokonywać skojarzenia (tj. po prostu zapisywać pozycję otwierającego nawiasu pod indeksem zamykającego i vice versa).

Taka tablica pozwala nam zaimplementować przedstawione powyżej graficznie rozwiązanie: iterujemy po łańcuchu wypisując znaki aż do napotkania nawiasu kwadratowego. Kiedy takowy znajdziemy, ustawiamy pozycję iteratora na skojarzony z obecną pozycją indeks i zmieniamy kierunek iteracji na przeciwny (podczas implementacji warto przy tym upewnić się, że w następnej kolejności wykonamy krok w odpowiednim kierunku, a nie natychmiastowy skok z powrotem).

Powyższe rozwiązanie osiąga złożoność liniową, ponieważ obie użyte pętle przechodzą po łańcuchu tylko raz, wykonując jedynie operacje w czasie stałym.

68.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/Reverse_code/solution.py.

```
s = input()
odpowiadajace_nawiasy = [0] * len(s)
stos = []
for i, znak in enumerate(s):
    if znak == '[':
        stos.append(i)
    elif znak == ']':
        ostatni = stos.pop()
        odpowiadajace_nawiasy[i] = ostatni
        odpowiadajace_nawiasy[ostatni] = i
kierunek = 1
i = 0
wynik = ""
while i < len(s):
    znak = s[i]
    if znak == '[' or znak == ']':
        i = odpowiadajace_nawiasy[i]
        kierunek = -kierunek
    else:
        wynik += znak
    i += kierunek
print(wynik)
```

68.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/Reverse_code/solution.cpp.

```cpp
#include <iostream>
#include <string>
#include <vector>

int main() {
  std::string s;
  std::cin >> s;
  std::vector<std::size_t> odpowiadajace_nawiasy(s.size());
  std::vector<std::size_t> stos;
  for (std::size_t i = 0; i < s.size(); i++) {
    char znak = s[i];
    if (znak == '[') {
      stos.push_back(i);
    } else if (znak == ']') {
      odpowiadajace_nawiasy[i] = stos.back();
      odpowiadajace_nawiasy[stos.back()] = i;
      stos.pop_back();
    }
  }
  int kierunek = 1;
  for (std::size_t i = 0; i < s.size(); i += kierunek) {
    char znak = s[i];
    if (znak == '[' || znak == ']') {
      i = odpowiadajace_nawiasy[i];
      kierunek = -kierunek;
    } else {
      std::cout << znak;
    }
  }
  std::cout << '\n';
  return 0;
}
```

69 (VII) – Wisielczy humor – Gnomy

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gnomy (IV edycja – zawody algorytmiczne – etap lokalny)***
*Język programowania: **C++/Python***

69.1 Treść zadania

Wybitny Bajtocki archeolog Bajtowit przebył wiele setek kilometrów podróżując po mrocznych jaskiniach i podziemnych korytarzach w poszukiwaniu Zaginionej Partycji. I dzisiaj w końcu nastał dzień, w którym skarb, pozornie utracony na wieki, znalazł się w zasięgu jego chudej ręki. Ostatnią przeszkodą, która dzieliła Bajtowita od sławy i bogactwa okazał się wredny Świnks, starożytna krzyżówka świni, kota i człowieka.

Rysunek 69.1: Obrazek przedstawia Świnksa (grafika: www.craiyon.com)

Świnks, zgodnie z opisem w „Mitach i legendach" Bajtandowskiego, nie tylko umiał mówić, ale był też obdarzony specyficznym poczuciem humoru i zamiłowaniem do zagadek. Na widok Bajtowita oznajmił głośno:

— *Kolego, ty też po skarb? Tu tylko dzisiaj już trzech podobnych typów się kręciło... daj sobie spokój!*

— *Czy zgodnie z regulaminem nie mam prawa przynajmniej spróbować rozwiązać twojej zagadki?* – zapytał Bajtowit.

— *No niby możesz, ale po co masz próbować, skoro i tak ci się nie uda?* — powiedział Świnks, ziewając ze znudzenia, po czym dodał — *Ba, nawet możesz sobie spróbować zgadnąć kilka razy, niech stracę! Jakie słowo mam na myśli?*

Bajtowita zamurowało. Próbował zgadnąć wiele, wiele razy, ale nawet znajomość długości słowa w niczym mu nie pomogła. W końcu Świnks machnął spiralną kitą, wyraźnie znużony.

— *Kolego, ostatnia próba i kończymy...* — ostrzegł.

Bajtowit zaczął gorączkowo przeglądać notatki ze swoich dotychczasowych odpowiedzi. Dla każdej próby odgadnięcia tajemniczego słowa wymyślonego przez Świnksa miał wynotowane dwie liczby. Pierwsza z nich opisywała liczbę poprawnie odgadniętych liter (znajdujących się na swoim miejscu) w słowie-zagadce, druga zaś – ile liter zostało podanych poprawnie, ale nie na swoim miejscu. Pomóż Bajtowitowi w rozwikłaniu zagadki Świnksa, a Bajtowit na pewno chętnie podzieli się z Tobą zyskiem z odnalezienia Zaginionej Partycji!

Zadanie

Świnks nie jest bezlitosny – daje Bajtowitowi wiele szans odgadnięcia pomyślanego (n – literowego) słowa (dokładniej: g takich prób). Po każdej próbie informuje też Bajtowita o tym, ile liter z jego odpowiedzi pokrywa się z literami pomyślanego przez niego słowa oraz ile znaków wymaga przestawienia, by znaleźć się na właściwym miejscu w rozpatrywanym słowie (oznaczmy te liczby dla wygody przez c i m).

Przykładowo, załóżmy że Świnks pomyślał o słowie *„burka”*, a Bajtowit poda słowo *„sroka”*. Wówczas litery *„ka”* pokrywają się, *„r”* jest nie na swoim miejscu, zaś *„s”* i *„o”* nie znajdują się w zgadywanym haśle. Zatem liczby c, m w tym przykładzie będą wynosiły odpowiednio 2 i 1.

Niestety, Świnks z racji bycia po części wieprzkiem ma słabą pamięć i zdarza mu się coś pomylić. Może się zdarzyć, że odpowiedzi Świnksa będą wzajemnie sprzeczne... Mimo to postaraj się zweryfikować, czy zagadka Świnksa i jego odpowiedzi na próby zgadnięcia hasła przez Bajtowita pozwalają jednoznacznie zidentyfikować pomyślane przez niego słowo.

Opis wejścia

W pierwszej linii wejścia otrzymasz pojedynczą liczbę naturalną $1 \leqslant T \leqslant 10$, opisującą liczbę zagadek, które starał się rozwiązać Bajtowit.

Opis pojedynczej zagadki zaczyna się od dwóch liczb całkowitych: n, g, odpowiadających kolejno: długości zgadywanego słowa ($3 \leqslant n \leqslant 5$) i liczbie prób odgadnięcia go, które podjął Bajtowit ($1 \leqslant g \leqslant 150$).

W każdym z kolejnych g wierszy znajdziesz po jednym n-literowym słowie X (stanowiącym próbę zgadnięcia hasła przez Bajtowita), po którym wypisane będą dwie liczby nieujemne całkowite: c, m. Pierwsza z nich, c odpowiada liczbie liter w słowie X, które pokrywają się z literami hasła wymyślonego przez Świnksa (tzn. już są na właściwych pozycjach).

Druga, m określa ile liter w X pokrywałoby się z literami wyrazu Świnksa pod warunkiem przestawienia ich w odpowiednie miejsce (oczywiście nie uwzględniamy tu już liter wliczonych w c, czyli tych na właściwych pozycjach).

Łatwo zauważyć, że $0 \leqslant c + m \leqslant n$.

Każde słowo X złożone jest wyłącznie z małych liter alfabetu łacińskiego (bez polskich znaków diakrytycznych).

Opis wyjścia

Dla każdej z zagadek Świnksa oczekiwane jest, że na wyjściu pojawi się jeden ciąg znakowy. W zależności od sytuacji ma to być:

- słowo pomyślane przez Świnksa, jeżeli daje się jednoznacznie je wyznaczyć na podstawie prób zgadnięcia hasła przez Bajtowita (i informacji o rezultatach tych prób);
- `WIELE POPRAWNYCH ODPOWIEDZI` – w sytuacji, gdy istnieje więcej niż jedno słowo, które mogłoby wygenerować takie liczby c, m jak te uzyskane dla wszystkich prób zgadnięcia hasła przez Bajtowita;
- `BRAK POPRAWNYCH ODPOWIEDZI` – w sytuacji, gdy notatki Bajtowita (lub Świnks) mylą się i nie istnieje żadne słowo o podanej długości, które miałoby takie liczby poprawnych dopasowań i liter nie na swoim miejscu, jak te zapisane w notatkach Bajtowita.

Przykład

Dla przykładowego wejścia:

```
3
4 7
baba 2 2
kuba 2 0
kuna 1 0
abku 2 0
kuku 0 0
jork 0 0
zywi 0 0
4 3
kora 2 0
arbu 0 1
kuba 1 0
4 2
ukos 4 0
skos 4 0
```

prawidłowym wyjściem jest:

```
abba
WIELE POPRAWNYCH ODPOWIEDZI
BRAK POPRAWNYCH ODPOWIEDZI
```

Wyjaśnienie przykładu

W pierwszym przypadku, już pierwsza próba zgadnięcia (*baba*) daje nam informację, że dwie litery są na swoim miejscu, a dwie pozostałe są zamienione miejscami. W drugiej próbie (*kuba*) widzimy, że tymi poprawnie zamieszczonymi literami są dwie końcowe litery, tj. *ba*. Prowadzi nas to do wniosku, że wystarczy jedynie zamienić dwie pierwsze litery, by uzyskać jedyną w tym wypadku poprawną odpowiedź, tj. *abba*. Weryfikujemy nasze próby odgadnięcia z pozostałymi podejściami Bajtowita i okazuje się, że *abba* jest jedynym pasującym hasłem.

W drugiej zagadce możemy zauważyć, że zarówno *khry* jak i *kgry* mogły być poprawnymi hasłami. Oznacza to, że zarówno jedno, jak i drugie mogło być poprawną odpowiedzią.

W trzecim wypadku Świnks musiał się pomylić – prawidłowym hasłem nie może być jednocześnie *skos* i *ukos*, a wskazówki Świnksa jednoznacznie to sugerują. Uzasadnia to nasz wybór trzeciej odpowiedzi.

69.2 Rozwiązanie

Zadanie to jest bardzo silnie inspirowane popularną niegdyś grą „Mastermind", polegającą na próbach odgadnięcia przez jednego z graczy ukrytego kodu, zaprojektowanego przez drugiego gracza. Znawcy gier planszowych z pewnością byli w stanie zauważyć podobieństwo między tymi tematami.

Rozwiązanie siłowe, polegające na weryfikacji wszystkich możliwych słów zadanej długości, wydaje się nam początkowo jedyną możliwą opcją i w kontekście złożoności obliczeniowej faktycznie nie jesteśmy w stanie zrobić dużo więcej. To, co jesteśmy w stanie zrobić to poprawa obserwowalnej szybkości rozwiązania poprzez eliminację z rozwiązania siłowego słów zawierających znaki, które nie mogą stanowić rozwiązania zagadki Świnksa.

Pozostaje zatem odpowiedzieć na pytanie – w jaki sposób należy to zrobić? Najprostszym sposobem jest zidentyfikowanie wszystkich prób odgadnięcia hasła, w których nie pojawia się ani jedna poprawna litera. Nic nie stoi na przeszkodzie, żeby przy podejściu polegającym na przejrzeniu wszystkich możliwych odpowiedzi pomijać litery występujące w takich próbach odgadnięcia rozwiązania.

Samo sprawdzanie, czy dany wyraz może być rozwiązaniem zagadki Świnksa jest minimalnie nieoczywiste od strony technicznej, chociaż bardzo łatwe od strony koncepcyjnej. Sprawdzenie, czy liczba poprawnie dopasowanych liter w pojedynczej odpowiedzi Bajtowita byłaby dobrze obliczona przy zaproponowanym rozwiązaniu zagadki jest dość proste i wymaga równoczesnego przeiterowania obydwu wyrazów. Weryfikacja liczby liter, które nie znajdują się na swoim miejscu jest odrobinę bardziej złożona. Najwygodniejszym sposobem, by ją przeprowadzić jest zliczenie ile razy każda z liter występuje w próbie zgadnięcia i w zaproponowanym haśle, a następnie zestawienie tych danych ze sobą. W Pythonie można to zrobić bardzo szybko z wykorzystaniem struktury `Counter` – w języku C++ możemy zrobić to przy pomocy struktury `map` lub stosując do tego celu zwykły wektor. Należy jedynie pamiętać, żeby od liczby liter powtarzających się w obydwu wyrazach odjąć liczbę znaków znajdujących się na poprawnych pozycjach.

Stosunkowo nietrywialny na pierwszy rzut oka jest też sposób, w jaki należy przeiterować wszystkie wyrazy o zadanej długości, jakie możemy ułożyć z dostępnych liter. Można zrobić to bardzo naiwnie – przygotować kilka funkcji, które będą w swoim ciele zawierały dostatecznie dużą liczbę zagnieżdżonych w sobie pętli. Jest to wykonalne z uwagi na to, że wejście ma stosunkowo niewielki rozmiar – ale czy na pewno jest to coś, co chcemy robić? Jeżeli nie, to zastanówmy się nad nieco bardziej wyrafinowanym podejściem. Jeżeli słowa o literach pochodzących z k-elementowego podzbioru alfabetu potraktować jako liczby, to wówczas wyrażane byłyby one w systemie o podstawie k. W takim razie konwersja kolejnych liczb naturalnych na słowa odbywa się w sposób relatywnie naturalny, analogiczny do konwersji liczb pomiędzy dwoma systemami liczbowymi o różnych podstawach. W proponowanych przez nas rozwiązaniach jest to dokonywane przez funkcję `get_next_item`.

Na koniec kilka uwag technicznych – prezentowane tutaj podejście jest prostą heurystyką, która jest daleka od ideału i nie wykorzystuje w pełni informacji, które stanowią wejście do tego zadania. Oczywiście możliwa jest poprawa zaproponowanego rozwiązania – można je rozwinąć na co najmniej kilka sposobów, zaczynając choćby od lepszego wykorzystania wiedzy związanej z liczbą poprawnie dopasowanych liter poprzez zestawianie ze sobą zbiorów o niezerowej liczbie podanych poprawnie liter. Szczegóły takiego podejścia pozostawiamy jednak do znalezienia i wykorzystania czytelnikowi.

69.2.1 Python

Źródło: ./zadania/07_Algorytmy_tekstowe/Wisielczy_humor/solution.py.

```
#Norek weź mu dostaw średniki na końcu linii. Brzydkie to tak, że się na to patrzeć nie da.
#Karol, tak nie wolno, to przecież pajton!
#Co nie wolno, co nie wolno?! Będzie działać ze średnikami szybciej, bo bardziej przypomina C++!

from collections import Counter;

indeterminate = "WIELE POPRAWNYCH ODPOWIEDZI";
incorrect = "BRAK POPRAWNYCH ODPOWIEDZI";

def matching(guess, verifier):
    score =0;
    for x in verifier.keys():
        if x in guess:
            score+= min(verifier[x],guess[x]);
    return score;

def get_next_item(L, item, index, available):
        if(index<0):
            return "";
        if(item[index]==available[len(available)-1]):
            item = item[:index]+available[0]+item[index+1:];
            return get_next_item(L,item, index-1,available);
        else:
            item = item[:index]+available[available.index(item[index])+1]+item[index+1:];
            return item;


def solve():
    length, guessnumber = list(map(int, input().split()))
    password = [];
    correct = [];
    misplaced = [];
    mappedpasswords = [];
    available = ['a','b','c','d','e','f','g','h','i','j','k','l',
                 'm','n','o','p','q','r','s','t','u','v','w','x','y','z']
    G = guessnumber;
    for _ in range(G):
        x,y,z = input().split(' ')
        y = int(y)
        z = int(z)
        if(y+z>0):
            password.append(x);
            correct.append(y);
            misplaced.append(z);
            mappedpasswords.append(Counter(x));
        else:
            guessnumber-=1;
            for j in range(len(x)):
```

```
                if x[j] in available:
                    available.remove(x[j]);
    found = 0;
    ans = "";
    SS="";
    if(len(available)>0):
        for i in range(length):
            SS+=available[0];
    while(SS!=""):
        s2map = Counter(SS);
        gud = True;
        for i in range(guessnumber):
            corr = 0;
            miss = 0;
            for q in range(length):
                if(password[i][q]==SS[q]):
                    corr+=1;
            miss = matching(mappedpasswords[i],s2map)-corr;
            if (corr!=correct[i] or miss!=misplaced[i]):
                gud = False;
                break;

        if(gud):
            if(ans==""):
                ans = SS;
            else:
#                print(ans,'\t',SS);
                print(indeterminate);
                return;
        SS = get_next_item(length,SS,length-1,available);
    #    print(SS);

    if(ans==""):
        print(incorrect);
        return;

    print(ans)


T = int(input())
for _ in range(T):
    solve()
```

69.2.2 C++

Źródło: ./zadania/07_Algorytmy_tekstowe/Wisielczy_humor/solution.cpp.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define indeterminate "WIELE POPRAWNYCH ODPOWIEDZI"
#define incorrect "BRAK POPRAWNYCH ODPOWIEDZI"

int pown(int x, int n)
    {
        int y = 1;
        // n = 2*d + r. x^n = (x^2)^d * x^r.
        int d = n >> 1;
        int r = n & 1;
        int x_2_d = d == 0? 1 : pown(x*x, d);
        int x_r = r == 0? 1 : x;
        return x_2_d*x_r;
    }

map<char,int> to_map(string &s){
  map<char,int> M = map<char,int>();
  for (auto const& x : s){
    if(M.find(x)!=M.end()){
      M[x]++;
    }
    else{
      M.insert({x,1});
    }
  }
  return M;
}

int matching(map<char,int> &guess, map<char,int> &verifier){
  int score =0;
  for (auto const& x : verifier){
    if(guess.find(x.first)!=guess.end()){
      score+= min(x.second,guess[x.first]);
    }
  }
  return score;
}

struct question{
  int length;
  int guessnumber;
  vector<string> password;
  vector<int> correct;
  vector<int> misplaced;
  vector<map<char,int>> mappedpasswords;
```

```
vector<char> available = vector<char>({'a','b','c','d','e','f','g','h','i','j','k','l',
  'm','n','o','p','q','r','s','t','u','v','w','x','y','z'});

map<char,int> to_map(string &s){
  map<char,int> M = map<char,int>();
  for(int i = 0; i < s.size(); i++){
    if(M.find(s[i])!=M.end()){
      M[s[i]]++;
    }
    else{
      M.insert({s[i],1});
    }
  }
  return M;
}

question(int n, int g){
  this->length = n;
  this->guessnumber = g;
  password = vector<string>();
  mappedpasswords = vector<map<char,int>>();
  correct = vector<int>();
  misplaced = vector<int>();
  string x;
  int y,z;
  for(int i = 0; i < g; i++){
    cin>>x>>y>>z;
    if(y+z>0){
      password.push_back(x);
      correct.push_back(y);
      misplaced.push_back(z);
      mappedpasswords.push_back(to_map(x));
    }
    else{//we delete all symbols used and worry not about the value here
      //no need to be painfully efficient in that though
      for(const auto& value: x){
        available.erase(remove(available.begin(), available.end(), value), available.end());
      }
      this->guessnumber--;
    }
  }

}

string get_first_item(int L){
  string S="";
  for(int i = 0; i<L;i++){
    S+=available[0];
  }
  return S;
}
```

```
string get_next_item(int L, string &item, int index){
  if(index<0){
    return "";
  }
  if(item[index]==available[available.size()-1]){
    item[index] = available[0];
    return get_next_item(L,item, index-1);
  }
  else{
    vector<char>::iterator it = find(this->available.begin(), this->available.end(), item[index]);
    it++;
    item[index]=*it;
    return item;
  }
}

string guessCheck(){
  int found = 0;
  string ans = "";

  string SS = get_first_item(this->length);
  while(SS!=""){
    map<char,int> s2map = to_map(SS);
    bool gud = true;
    for(int i =0; i<this->guessnumber; i++){
      int corr = 0;
      int miss = 0;
      for(int q = 0; q< this->length; q++){
        if(this->password[i][q]==SS[q]){
          corr++;
        }
      }

      miss = matching(s2map,mappedpasswords[i])-corr;
      if(corr!=correct[i] or miss!=misplaced[i]){
      //  if(this->length==3){cerr<<"Guess "<<SS<<" rejected at test
      // "<<password[i]<<" "<<correct[i]<<" "<<misplaced[i]<<'\n';}
        gud = false;
        break;
      }
    }
    if(gud){
        if(ans==""){
        ans = SS;
      }
    else{
        return indeterminate;
      }
    }
    SS = get_next_item(this->length,SS,this->length-1);
  }
if(ans==""){
```

```
    return incorrect;
  }
  return ans;
  }
};

int main(){
  int T;
  int n,g;
  cin>>T;
  while(T-->0){
    cin>>n>>g;
    question Q = question(n,g);
    cout<<Q.guessCheck()<<'\n';
  }
  return 0;
}
```

70 (VII) – A tu mamy mamuta! – Gnomy

Autor: Przemysław Gordinowicz, Politechnika Łódzka
Kategoria: Gnomy (III edycja – zawody algorytmiczne – etap regionalny)
Język programowania: C++/Python

70.1 Treść zadania

Palindromy (zdania lustrzane) są to wyrażenia (słowa, zdania, dłuższe utwory), które czytane wspak są takie same. Nazwa „palindrom" pochodzi z greckich słów „palin" (z powrotem, wracać po swoich śladach) i „dromos" (droga). Palindromem jest na przykład tytuł tego zadania.

Tworzenie palindromów ma wielowiekową tradycję. Anglicy twierdzą, że pierwsze zdanie wypowiedziane przez człowieka brzmiało „Madam, I'm Adam" (por. T. Morawski, strona `www.palindromy.pl`). Przypisywano im znaczenie magiczne. Współcześnie traktowane są jako żarty słowne i rozrywka umysłowa. W zabawy słowne związane z tworzeniem palindromów angażowali się znani poeci, np. Julian Tuwim (najdłuższy palindrom, jaki stworzył, liczył 92 litery: *A no, sam maj trapi; sanatora Kujaw martwi to, że ta sanacja baje; a nas? a też! ot, i w tramwaju karota! nasi! partia masona!*[1]), Stanisław Barańczak (235 liter) oraz hobbyści – prof. Tadeusz Morawski (elektronik) stworzył najdłuższy na świecie palindrom, liczący ponad 33 tys. liter.

Zadanie

Napisać program który w podanym tekście wyszuka i wskaże długość najdłuższego palindromu. Szukając palindromów należy przyjąć, że:

- wielkość liter nie ma znaczenia;
- pomija się odstępy (spacje) i znaki przestankowe;
- dla uproszczenia, tekst – oprócz znaków przestankowych (wskazanych poniżej) – zawiera wyłącznie małe i wielkie litery alfabetu łacińskiego, pomijając znaki specjalne, np. polskie litery.

Opis wejścia

Wejście składa się z pewnej liczby napisów (tekstów, w których należy szukać palindromów). W pierwszej linii wejścia pojawia się wartość $n \leq 100$ oznaczająca liczbę napisów do zbadania. Następnie w n kolejnych wierszach pojawiają się napisy. Napis może zawierać:

- małe litery alfabetu łacińskiego (od *a* do *z*);
- wielkie litery alfabetu łacińskiego (od *A* do *Z*);
- spacje (odstępy);
- wybrane znaki przestankowe:
  - wykrzyknik (!);

[1] Podany według Wikipedii; nie jest palindromem w ścisłym sensie, być może przez reformę ortografii w 1936 r.

- przecinek (,);
- kropkę (.);
- znak zapytania (?).

Można przyjąć, że napisy nie są dłuższe niż 30 000 znaków.

Opis wyjścia

Na wyjściu standardowym dla każdego zapytania powinna pojawić się stosowna odpowiedź, będąca długością najdłuższego palindromu znalezionego w tekście (spacji ani znaków przestankowych nie liczymy).

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
6
A tu mamy mamuta!
Centrum Mistrzostwa Informatycznego
Czy mama ma makaron?
!!!
A,???
Zakopane chce na pokaz.
```

prawidłową odpowiedzią jest:

```
13
2
7
0
1
19
```

Wyjaśnienie przykładów

Poniżej wytłuszczone zostały palindromy wskazanej długości:

- **A tu mamy mamuta**!
- Centru**m M**istrzostwa Informatycznego
- Czy m**ama ma ma**karon?
- (tekst „!!!” nie zawiera żadnej litery)
- **A**,???
- **Zakopane chce na pokaz**.

70.2 Rozwiązanie

Problematyka zadania sprowadza się do efektywnego wyszukiwania palindromów w tekście. W ramach wstępnej obróbki wymaga się usunięcia z tekstu zbędnych (wskazanych w treści zadania) znaków oraz ujednolicenia wielkości liter (zamiany wszystkich na wielkie lub małe, jak poniżej). Ponadto warto dodać znak specjalny (użyjemy *) pomiędzy każde dwa znaki w tekście (na początku i końcu też) – dzięki temu w tekście znajdują się tylko nieparzyste palindromy (a długość oryginalnego wyliczyć łatwo, jako $\frac{k-1}{2}$, gdzie k jest długością palindromu w przekształconym tekście). Realizujący to kod w języku Python może wyglądać tak, jak poniżej. Przykładowo, zamieni on napis mamma na *m*a*m*m*a*, zaś pustemu napisowi będą odpowiadać 2 znaki **.

```
1  n = int(input())
2  for i in range(n):
3      line = "".join(["*", "*".join(input().translate(str.maketrans(
4                              "", "",  " !,.?")).lower()), "*"])
5      print(palindrome(line))
```

Do opisania pozostaje funkcja wyznaczająca długość najdłuższego palindromu w tekście. Przedstawimy tu dwa podejścia:

1. Rozwiązanie naiwne (acz możliwie efektywne), które jednak nie zapewnia uzyskania maksimum punktów za to zadanie, o złożoności obliczeniowej $O(mk)$, gdzie m jest długością badanego tekstu, a k – długością najdłuższego palindromu;
2. Algorytm Manachera [14] o złożoności obliczeniowej $O(m)$.

Rozwiązanie naiwne

Dzięki wstawieniu dodatkowych znaków szukamy wyłącznie palindromów o nieparzystej długości. Testować, czy dany tekst jest palindromem najwygodniej jest od środka, porównując znaki w tej samej odległości od pozycji środkowej. Poniższy kod wskazuje rozmiar palindromu o środku na pozycji `pos` w napisie `text` o długości m:

```
1  def pal_size(text, m, pos):
2      if m-pos <= 1:
3          return 1
4      for j in range(1, min(pos+1, m-pos)):
5          if text[pos+j] != text[pos-j]:
6              return j-1
7      return min(pos, m-pos-1)
```

Warto zwrócić uwagę, że w linii 6 funkcja zwraca $j-1$ podczas, gdy rzeczywisty rozmiar palindromu to $2j-1$ – wynika to z pominięcia dodatkowych znaków *. Analogicznie jest w linii 7, wykonywanej, gdy fragment będący palindromem sięga do końca tekstu. Dla przykładu:

`pal_size("*m*a*m*m*a*", 11, 3)` zwróci wartość 3 (*m*a*m* → mam);

`pal_size("*m*a*m*m*a*", 11, 6)` zwróci wartość 4 (*a*m*m*a* → amma).

Wykorzystując powyższą funkcję wystarczy znaleźć najdłuższy palindrom sprawdzając palindromy o środku w każdej możliwej pozycji. Z uwagi na poszukiwanie najdłuższego palindromu warto jednak pamiętać, że ewentualne, długie palindromy mają swój środek daleko od końców badanego napisu. Warto więc rozpoczynać poszukiwania na środku tekstu przesuwając centralny punkt badanego palindromu na zewnątrz.

```
1  def palindrome(text):
2      m = len(text)
3      if m <= 2:  # krótkie słowa (tylko specjalne znaki)
4          return 0
5      c = m // 2
6      size = pal_size(text, m, c) # palindrom centralny
7      for i in range(1, c):       # palindromy przesunięte
8          if size > m - 2*i - 2:  # już nie będzie większych palindromów
9              break
10         s = max(pal_size(text, m, c-i),
11                 pal_size(text, m, c+i))
12         if s > size:
13             size = s
14     return size
```

Funkcja `pal_size` działa w czasie proporcjonalnym do długości znalezionego palindromu, a więc ograniczonym przez k (długość najdłuższego palindromu). Ponieważ będzie wywołana (linijki 10 i 11 powyżej) maksymalnie $2 * (m - k + 1)$ razy, stąd złożoność obliczeniową tego podejścia można oszacować przez $O((m - k + 1)k)$ lub – bardziej ogólnie – przez $O(mk)$.

Algorytm Manachera

Okazuje się, że lepiej wykorzystując własności palindromów, można znacząco zredukować czas obliczeń. Rozpocznijmy uproszczenia podanego powyżej kodu, przy okazji wyznaczając listę rozmiarów wszystkich maksymalnych palindromów w tekście (dla każdego możliwego środka palindromu).

```
1  def palindrome(text):
2      m = len(text)
3      palsizes = []
4      c = 0
5      s = 0
6      while c < m:
7          while c-s-1 >= 0 and c+s+1 < m and text[c-s-1] == text[c+s+1]:
8              s += 1
9          palsizes.append(s)
10         c += 1
11         s = 0
12     return max(palsizes)
```

Podany kod jest oczywiście mniej efektywny, ale jego złożoność obliczeniowa szacuje się (nadal) przez $O(mk)$.

Zauważmy jednak pewne potencjalne usprawnienie. Gdy znajdziemy długi palindrom, a następnie (linia 10) inkrementujemy c, czyli przesuwamy środek kolejnego poszukiwanego palindromu, to fragment tekstu wokół c mieści się wewnątrz tamtego długiego palindromu (będziemy go nazywać *poprzednim palindromem*) i można wykorzystać wyznaczone wcześniej informacje „z drugiej strony lustra", gdyż każdy znak w prawej części poprzedniego palindromu ma odbicie lustrzane we wcześniej przetworzonej lewej jego części.

Usprawniony kod od linijki 10 będzie wyglądać jak poniżej.

```
10          prevc = c
11          prevs = s
12          c += 1
13          s = 0
14          while c <= prevc + prevs:
15              mirroredc = prevc - (c - prevc)
16              maxsize = prevc + prevs - c
17              if palsizes[mirroredc] < maxsize:
18                  palsizes.append(palsizes[mirroredc])
19                  c += 1
20              elif palsizes[mirroredc] > maxsize:
21                  palsizes.append(maxsize)
22                  c += 1
23              else:   # palsize[mirroredc] == maxsize
24                  s = maxsize
25                  break
26      return max(palsizes)
```

Warto zwrócić uwagę, że:

linia 14: pętlę wykonujemy dopóki c jest wewnątrz poprzedniego palindromu;
linia 15: wyznaczamy środek podpalindromu dającego lustrzane odbicie wewnątrz poprzedniego palindromu;
linia 16: wyznaczamy maksymalny rozmiar podpalindromu o środku w c, będącego wewnątrz poprzedniego palindromu;
linia 17: *lustrzany podpalindrom* mieścił się wewnątrz poprzedniego palindromu – palindrom o środku w c będzie więc taki sam;
linia 20: lustrzany podpalindrom rozszerzał się poza poprzedni palindrom z lewej strony; z prawej się nie rozszerza, więc znamy rozmiar palindromu o środku w c;
linia 23: odbicie lustrzanego podpalindromu kończy się dokładnie tam, gdzie poprzedni palindrom, więc jest szansa na potencjalne dalsze rozszerzenie palindromu o środku w c poza poprzedni palindrom.

Aby oszacować złożoność obliczeniową algorytmu Manachera należy zauważyć, że albo przesuwamy prawy kraniec aktualnie rozważanego palindromu (linie 7–8 kodu), albo operując w ramach poprzedniego palindromu przesuwamy środek aktualnie rozważanego palindromu (linie 14–25). W każdym przebiegu jednej z pętli, któraś z tych wielkości jest powiększana, a obie są ograniczone przez m. Dlatego czas wykonania można oszacować przez $O(m)$.

70.2.1 Python

Wersja naiwna:

Źródło: ./zadania/07_Algorytmy_tekstowe/A_tu_mamy_mamuta/solutionnaive.py.

```
"""
A tu mamy mamuta! - rozwiązanie wzorcowe (naiwne)

Znajdujemy najdłuższy palindrom w tekście i podajemy jego długość
Ignorujemy znaki przestankowe i spacje.
"""

def pal_size(text, m, pos):
    """
    :param text:
    :param m:
    :param pos:
    :return: długość palindromu (nieparzystego) o środku w pos
    """
    if m-pos <= 1:
        return 1
    for j in range(1, min(pos+1, m-pos)):
        if text[pos+j] != text[pos-j]:
            return j-1
    return min(pos, m-pos-1) # palindrom do końca

def palindrome(text):
    """
    :param text: analizowany napis z dodanymi nawiasami (jako wartownikami)
    :return: długość najdłuższego palindromu
    """
    m = len(text)
    if m <= 2:  # krótkie słowa (tylko dodane znaki)
        return 0
    c = m // 2
    # palindrom centralny
    size = pal_size(text, m, c)
    #palindrom przesunięty
    for i in range(1, c):
        if size > m - 2*i - 2:      # już nie będzie większych palindromów
            break
        s = max(pal_size(text, m, c-i),
                pal_size(text, m, c+i))
        if s > size:
            size = s
    return size


n = int(input())
for i in range(n):
    # napis z usuniętymi znakami specjalnymi, konwersją na małe litery i gwiazdką pomiędzy każdymi
```

```
    # dwoma znakami (+ początek i koniec), dzięki temu szukamy tylko nieparzystych palindromów
    # do tego nawiasy jako wartownicy
    line = "".join(["*", "*".join(input().translate(str.maketrans(
       "", "",  " !,.?")).lower()), "*"])
    print(palindrome(line))
```

Rozwiązanie z algorytmem Manachera:

Źródło: ./zadania/07_Algorytmy_tekstowe/A_tu_mamy_mamuta/solution.py.

```
"""
A tu mamy mamuta! - rozwiązanie wzorcowe, algorytm Manachera (liniowy)

Znajdujemy najdłuższy palindrom w tekście i podajemy jego długość
Ignorujemy znaki przestankowe i spacje.
"""

def palindrome(text):
    """
    :param text: analizowany napis (z * pomiędzy znakami)
    :return: długość najdłuższego palindromu w napisie text
    """
    m = len(text)
    palsizes = []
    c = 0
    s = 0
    while c < m:
        # szukamy palindromu o środku w c
        while c-s-1 >= 0 and c+s+1 < m and text[c-s-1] == text[c+s+1]:
            s += 1
        palsizes.append(s) # zapamiętuje długość najdłuższego palindromu o środku w c
        # poniżej przeliczamy c i s do kolejnej iteracji korzystając ze wcześniejszych informacji
        prevc = c
        prevs = s
        c += 1
        s = 0      # domyślne wartości, ale może coś poprawimy
        while c <= prevc + prevs:
            # skorzystamy z faktu, że center leży w poprzednim palindromie a każdy znak w palindromie
            # ma odbicie lustrzane we wcześniejszej części. Użyjemy danych z tego odbicia.
            mirroredc = prevc - (c - prevc)
            maxsize = prevc + prevs - c
            if palsizes[mirroredc] < maxsize:
                palsizes.append(palsizes[mirroredc])
                c += 1
            elif palsizes[mirroredc] > maxsize:
                palsizes.append(maxsize)
                c += 1
            else:   # palsize[mirroredc] == maxsize
                s = maxsize
                break   # ten może da się powiększyć
    return max(palsizes)
```

```
n = int(input())
for i in range(n):
    # napis z usuniętymi znakami specjalnymi, konwersją na małe litery i gwiazdką pomiędzy każdymi
    # dwoma znakami (+ początek i koniec), dzięki temu szukamy tylko nieparzystych palindromów
    # do tego nawiasy jako wartownicy
    line = "".join(["*", "*".join(input().translate(str.maketrans(
       "", "",  " !,.?")).lower()), "*"])
    print(palindrome(line))
```

70.2.2 C++

Wersja naiwna:

Źródło: ./zadania/07_Algorytmy_tekstowe/A_tu_mamy_mamuta/solutionnaive.cpp.

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
#include <vector>

void rozwiaz_test() {
  std::string oryginalny_tekst;
  std::getline(std::cin, oryginalny_tekst);
  std::string uproszczony_tekst = "(*";
  for (char znak : oryginalny_tekst) {
    if (std::isalpha(znak)) {
      uproszczony_tekst += std::tolower(znak);
      uproszczony_tekst += "*";    // dokładam "*", żeby szukać nieparzystego palindromu
    }
  }
  uproszczony_tekst += ")"; // wartownik
  int dlugosc = uproszczony_tekst.size();
  int wynik = 0;
  for (int i = 0; i < dlugosc / 2; i++) { // szukam palindromu o środku w  dlugosc/2 +/- i
    int centr = dlugosc/2 - i;
    int rad = 1;
    while (uproszczony_tekst[centr + rad] == uproszczony_tekst[centr - rad])
      rad ++;
    if (rad > wynik)
      wynik = rad;
      if (centr - wynik == 0)
        break;
    centr = dlugosc/2 + i;
    rad = 1;
    while (uproszczony_tekst[centr + rad] == uproszczony_tekst[centr - rad])
      rad ++;
    if (rad > wynik)
      wynik = rad;
```

```
    if (centr + wynik >= dlugosc - 2) {// dłuższego palindromu już nie znajdę
      break;
    }
  }
  std::cout << wynik-1 << '\n';
}

int main() {
  int liczba_testow;
  std::cin >> liczba_testow;
  std::ws(std::cin);
  for (int i = 0; i < liczba_testow; i++) {
    rozwiaz_test();
  }
  return 0;
}
```

Rozwiązanie z algorytmem Manachera:

Źródło: ./zadania/07_Algorytmy_tekstowe/A_tu_mamy_mamuta/solution.cpp.

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
#include <vector>
void rozwiaz_test() {
  std::string oryginalny_tekst;
  std::getline(std::cin, oryginalny_tekst);
  std::string uproszczony_tekst = "\1";
  for (char znak : oryginalny_tekst) {
    if (std::isalpha(znak))
      uproszczony_tekst += std::tolower(znak);
  }
  int dlugosc = uproszczony_tekst.size();
  int wynik = 0;
  for (int parzystosc = 0; parzystosc < 2; parzystosc++) {
    std::vector<int> promienie(dlugosc);
    int a = 0;
    int b = 0;
    for (int i = 1; i < dlugosc; i++) {
      if (i <= b)
        promienie[i] = std::min(promienie[a + b - i + parzystosc], b - i);
      while (uproszczony_tekst[i - promienie[i] - 1]
        == uproszczony_tekst[i + promienie[i] + 1 - parzystosc]) {
        promienie[i]++;
      }
      if (b < i + promienie[i]) {
        a = i - promienie[i] - parzystosc;
        b = i + promienie[i];
      }
```

```
    }
    for (int j = 1; j < dlugosc; j++) {
      int dlugosc_palindromu = promienie[j] * 2 - parzystosc + 1;
      wynik = std::max(wynik, dlugosc_palindromu);
    }
  }
  std::cout << wynik << '\n';
}
int main() {
  int liczba_testow;
  std::cin >> liczba_testow;
  std::ws(std::cin);
  for (int i = 0; i < liczba_testow; i++) {
    rozwiaz_test();
  }
  return 0;
}
```

Centrum Mistrzostwa Informatycznego

Dział VIII

Grafy

71 (VIII) – Wprowadzenie

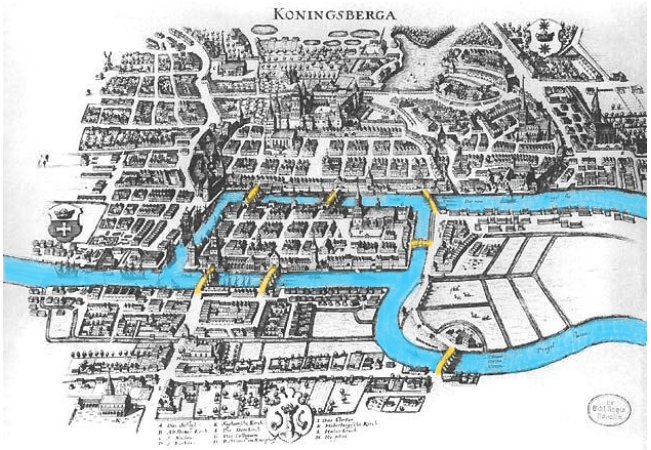
Autor: Paweł Koszałka, Akademia Górniczo-Hutnicza w Krakowie

Pod pojęciem *grafu* rozumiemy w informatyce pewien sposób organizacji danych, znacząco różny i bardziej elastyczny niż tablica lub lista przechowująca po prostu ciąg kolejnych elementów [2][4][7][1][21]. Z uwagi na tę elastyczność i wszechstronność, a także na istnienie wielu praktycznych zastosowań oraz nietrywialnych algorytmów wykorzystujących grafy, pojawiają się one również często w konkursowych zadaniach algorytmicznych [17]. Z tego też względu niniejszy dział jest najobszerniejszy ze wszystkich, przy czym zdecydowana większość omówionych tu zadań przeznaczona jest dla uczniów szkół ponadpodstawowych (czyli dla kategorii Gremliny). Zanim jednak przejdziemy do zadań, musimy wprowadzić pewne pojęcia i podstawy teoretyczne, a rozpoczniemy od krótkiego rysu historycznego.

71.1 Zagadnienie mostów królewieckich

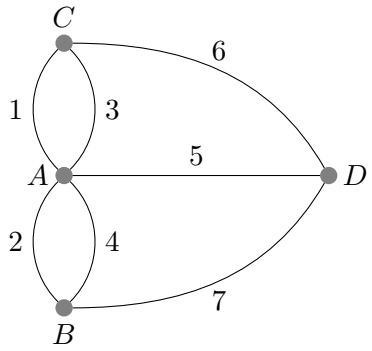
Teoria grafów, powszechnie stosowana w informatyce, jak i poza nią, stanowi gałąź matematyki, która swój początek wzięła od zadania rozwiązanego dla zabawy przez jednego z najwybitniejszych matematyków w historii – Leonharda Eulera.

Wydarzyło się to w 1736 roku w Królewcu, gdzie Euler zatrzymał się podczas jednej ze swoich podróży po Europie, a problem dotyczył topografii miasta. W centrum Królewca, zbudowanego nad rzeką Pregołą, znajdował się wówczas układ siedmiu mostów, który łączył leżącą tam wyspę zwaną Knipawa z innymi częściami miasta oddzielonymi przez dwie odnogi rzeki. Pytanie brzmiało następująco: „Czy da się przespacerować przez



Rysunek 71.1: Rozmieszczenie mostów w Królewcu w 1652 roku

wszystkie mosty w taki sposób, aby przez każdy z nich przejść tylko raz i powrócić do punktu wyjścia?” Euler udowodnił, że jest to niemożliwe i posłużył się przy tym rysunkiem złożonym z kropek i kresek połączonych ze sobą tak, żeby tworzyły model tej szczególnej architektury. Uproszczony schemat pomijał wszystkie zbędne szczegóły zachowując przy tym istotę problemu. Dzięki zastosowaniu takiego narzędzia, udało się utożsamić



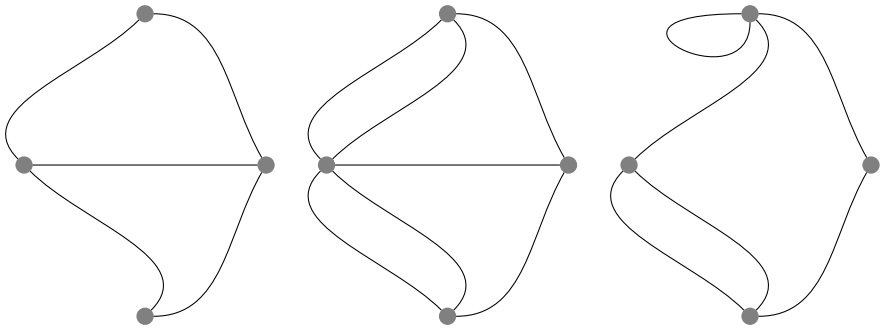
Rysunek 71.2: Układ mostów wyrażony grafem

problem wyjściowy z pytaniem: czy można narysować drogę zamkniętą[1], która przez każdą krzywą przechodzi dokładnie raz? Euler wykazał, że da się to zrobić tylko wtedy, gdy liczba linii wychodzących z każdego punktu jest parzysta.

Był to właśnie *graf*, a podobnego narzędzia używali później badacze z różnych dziedzin: Kirchhoff, Cayley, Jordan, Hamilton, Lewin, Uhlenbeck, Lee, Young i in. Mosty zostały zniszczone podczas II wojny światowej, ale dały początek teorii grafów, bez której zastosowań w architekturze, inżynierii, informatyce i telekomunikacji, a także naukach społecznych, czy dziedzinach związanych ze znajdowaniem optymalnych rozwiązań problemów planowania trudno byłoby dzisiaj funkcjonować.

71.2 Wierzchołki, krawędzie i ściany

Piękno grafów polega przede wszystkim na ich prostocie, dzięki której możemy przedstawić nawet najbardziej skomplikowane problemy. Albowiem *graf ogólny* G lub po prostu *graf*, składa się z niepustego i skończonego zbioru $V(G)$, którego elementy nazywamy *wierzchołkami* lub *węzłami* oraz skończonej rodziny $E(G)$ nieuporządkowanych par elementów zbioru $V(G)$ nazywanych *krawędziami*. Słowo *rodzina* oznacza, że mogą istnieć powtórzenia, czyli *krawędzie wielokrotne*. Jeśli tak jest, wtedy graf nazywamy *multigrafem*. Jeśli powtórzeń nie ma, a każdą parę wierzchołków łączy co najwyżej jedna krawędź, to graf taki nazywamy *prostym*. *Pętla* to krawędź łącząca wierzchołek sam ze sobą, a graf posiadający pętlę nazywamy *pseudografem*.

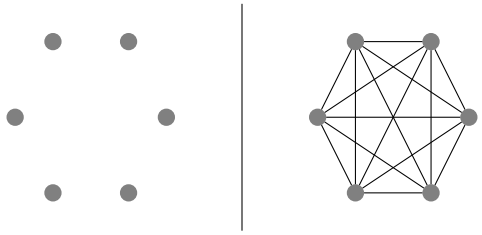


Rysunek 71.3: Graf prosty, multigraf i pseudograf

Zbiory $V(G)$ i $E(G)$ będziemy oznaczać w skrócie jako V i E, zaś sam graf – jako $G = (V, E)$. Wierzchołki z reguły oznaczamy liczbami naturalnymi $1, 2, 3, \ldots$ lub literami $A, B, C, \ldots$ itd. Mówimy także, że krawędź

[1]Droga zamknięta, to taka, której początek i koniec znajdują się w tym samym punkcie.

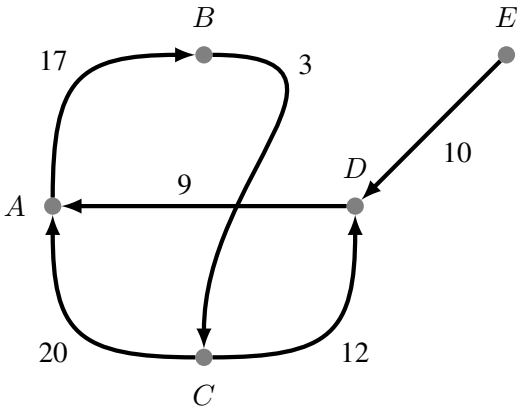
$\{v, w\}$ *łączy* wierzchołki v i w, co krócej oznaczamy przez vw. Dwa wierzchołki połączone jedną krawędzią nazywamy *sąsiednimi* – mówimy wówczas, że wierzchołki v i w są *incydentne* z tą krawędzią. Kiedy dwie krawędzie mają wspólny wierzchołek mówimy o *sąsiedztwie krawędzi*. *Graf pusty* składa się tylko z wierzchołków i nie posiada żadnych krawędzi, natomiast w *grafie pełnym* znajdują się wszystkie możliwe kombinacje krawędzi pomiędzy wierzchołkami. Grafy puste o n wierzchołkach będziemy oznaczać N_n, a grafy pełne K_n. Każdy graf K_n posiada $\frac{n(n-1)}{2}$ krawędzi, czyli $\binom{n}{2}$.



Rysunek 71.4: Przykład grafu pustego (po lewej) i pełnego (po prawej)

Jeżeli krawędziom lub wierzchołkom grafu przypiszemy jakieś dane, to mówimy o *grafie etykietowanym*. Jeśli krawędzie mają etykiety w postaci liczbowej, to liczby te nazywamy *wagami*, a taki graf – *grafem ważonym*.

Krawędź zdefiniowaliśmy wyżej jako nieuporządkowaną parę wierzchołków. Jeśli jednak ją uporządkujemy, to nazwiemy taką krawędź *skierowaną* (lub *łukiem*), co możemy zaznaczyć na rysunku za pomocą strzałki. Jeśli wszystkie krawędzie są łukami, to graf taki nazywamy *skierowanym* lub *digrafem*. Analogicznie możemy powiedzieć o *grafie nieskierowanym* wtedy, gdy wszystkie jego krawędzie nie mają strzałek.



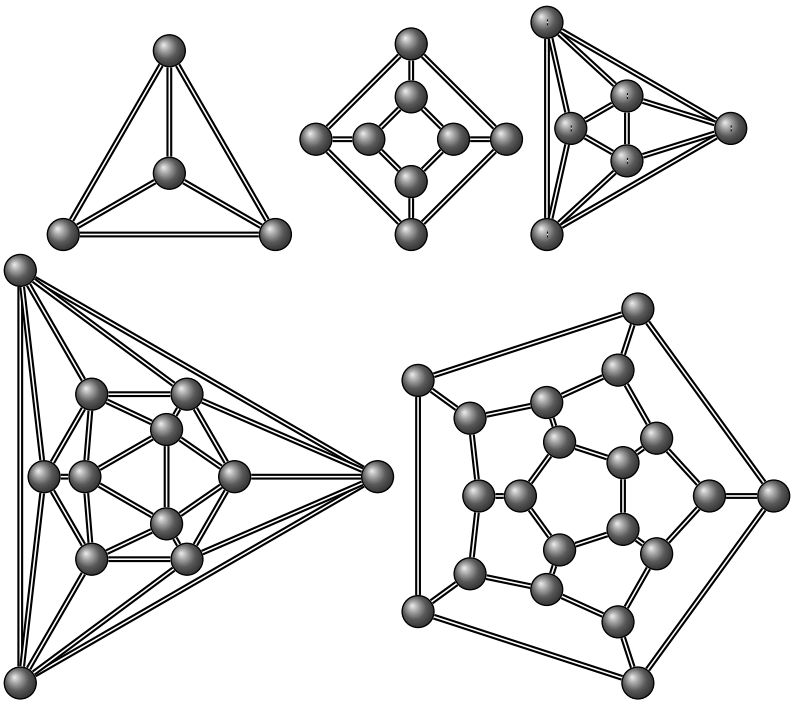
Rysunek 71.5: Graf skierowany i ważony

Stopniem wierzchołka v nazywamy liczbę krawędzi do niego dochodzących (w grafie nieskierowanym) albo sumę wchodzących i wychodzących (dla grafu skierowanego) i oznaczamy jako $deg(v)$ (od ang. *degree*). Inaczej mówiąc, jest to liczba krawędzi incydentnych z v. Wierzchołek stopnia 0 nazywamy *izolowanym*, a wierzchołek stopnia 1: *końcowym*. Graf, w którym każdy wierzchołek ma ten sam stopień r nazywamy *grafem regularnym stopnia* $\mathbf{r}$ lub *grafem* $\mathbf{r}$*-regularnym*. Graf pusty N_n jest zatem grafem regularnym stopnia 0. Szczególnymi przykładami grafów regularnych są *grafy platońskie*, utworzone z wierzchołków i krawędzi pięciu wielościanów foremnych: czworościanu, sześcianu, ośmiościanu, dwunastościanu i dwudziestościanu zwanych platońskimi. Grafy platońskie spełniają tzw. *formułę Eulera* sformułowaną przez niego w 1750 roku:

$$n - m + f = 2\ ,$$

gdzie n oznacza liczbę wierzchołków grafu, m liczbę krawędzi, zaś f liczbę ścian grafu.

Przez *ściany* rozumiemy obszary, czyli zbiory punktów płaszczyzny, na które dzielą ją krawędzie grafu. Do ścian zaliczamy również „obszar nieskończony”, czyli *zewnętrze* grafu.



Rysunek 71.6: Grafy platońskie

W roku 1736 Leonhard Euler wykazał tzw. *Lemat o uściskach dłoni*, który mówi, że:

Lemat 71.1 (Euler, 1736)

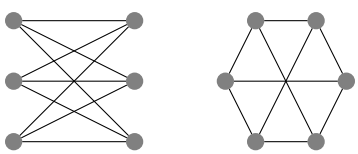
Jeśli pewne osoby witają się podając sobie dłonie, to łączna liczba uściśniętych dłoni jest parzysta, ponieważ w każdym uścisku uczestniczą dokładnie dwie osoby.

Wynika z tego fakt, że *w każdym grafie suma stopni wszystkich wierzchołków jest liczbą parzystą*, a co za tym idzie: *w dowolnym grafie liczba wierzchołków o nieparzystych stopniach jest parzysta*.

Istnieje również pojęcie *grafu nieskończonego*, w którym zarówno zbiór wierzchołków V, jak i rodzina krawędzi E są zbiorami nieskończonymi.

71.3 Izomorfizm

Ponieważ nie rozpatrujemy grafów w kategoriach geometrii planarnej, reprezentacja graficzna tego samego grafu może być różna. Ważne zatem jest, aby umieć stwierdzić czy dane dwa grafy są *równoważne* – inaczej mówiąc: *izomorficzne*. Aby tak było, muszą one mieć taką samą liczbę wierzchołków oraz odpowiadających im krawędzi incydentnych, aby zachowany był stopień każdego wierzchołka.

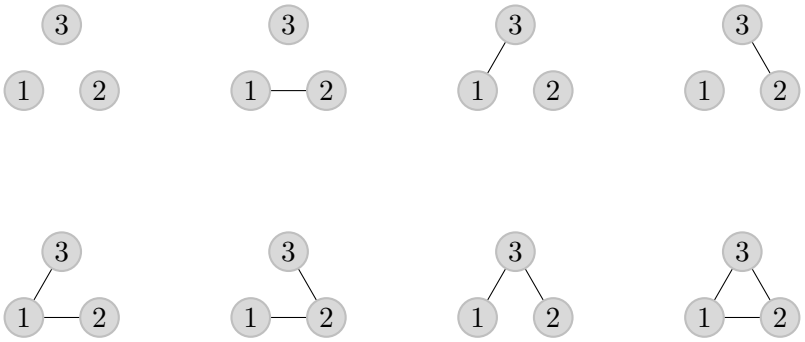


Rysunek 71.7: Grafy izomorficzne (zauważ, że w pierwszym grafie wystarczy zamienić miejscami oba środkowe wierzchołki, aby uzyskać układ analogiczny jak w drugim)

W wielu problemach pomijamy oznaczenia (etykiety) wierzchołków, co ma wpływ na to, jakie grafy możemy nazwać izomorficznymi oraz ile różnych grafów (z dokładnością do izomorfizmu) da się skonstruować. Przykładowo, dla grafów „nieoznaczonych" o trzech wierzchołkach, istnieją cztery różne (nieizomorficzne) grafy pokazane poniżej:

Rysunek 71.8: Różne (z dokładnością do izomorfizmu) grafy 3-wierzchołkowe nieoznaczone

Jeśli jednak nadamy wierzchołkom unikatowe etykiety, to liczba wszystkich możliwych nieizomorficznych grafów wzrośnie (w tym przypadku – do ośmiu):



Rysunek 71.9: Różne (z dokładnością do izomorfizmu) grafy 3-wierzchołkowe oznaczone

71.4 Trasa, ścieżka, droga, cykl

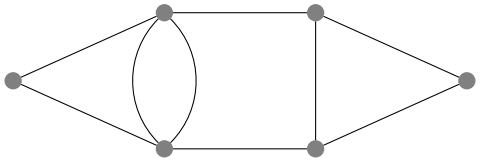
Trasą (lub *marszrutą*) w danym grafie G nazywamy skończony ciąg krawędzi postaci:

$$v_0v_1, v_1v_2, \ldots, v_{m-1}v_m,$$

w którym każde dwie kolejne krawędzie są sąsiednie albo identyczne. Wierzchołek v_0 nazywamy wtedy *wierzchołkiem początkowym*, a wierzchołek v_m *wierzchołkiem końcowym* trasy. Liczbę krawędzi na trasie nazywamy *długością trasy*. Trasę zapisujemy również jako:

$$v_0 \to v_1 \to \cdots \to v_m.$$

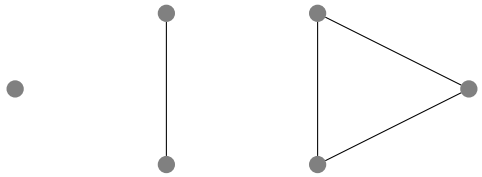
Trasę, w której wszystkie krawędzie są różne nazywamy *ścieżką*. Jeżeli w ścieżce wszystkie wierzchołki z wyjątkiem początku i końca są różne, to nazywamy ją *drogą*. Ścieżka jest *zamknięta* wtedy, gdy $v_0 = v_m$, a jeśli zawiera co najmniej jedną krawędź nazywamy ją *cyklem*. Kiedy mamy do czynienia z zamkniętą drogą, to mówimy o *cyklu prostym*. Graf nie posiadający cykli określamy jako *graf acykliczny*.



Rysunek 71.10: Graf zawierający cykle

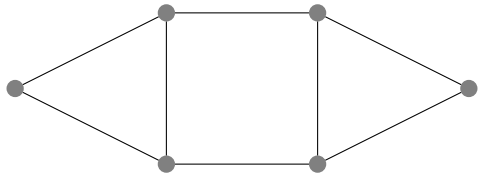
71.5 Spójność

Grafy G_1 i G_2 można ze sobą połączyć. Jeśli zbiory $V(G_1)$ i $V(G_2)$ są rozłączne, wtedy *sumą grafów* $G_1 \cup G_2$ nazywamy graf, którego zbiorem wierzchołków jest $V_1 \cup V_2$ a zbiorem krawędzi zbiór $E1 \cup E2$. Graf, którego nie da się przedstawić w postaci sumy dwóch grafów nazywamy *spójnym*. Równoważnie możemy to opisać, iż w grafie spójnym między każdą parą wierzchołków istnieje łączące je ścieżka. W przeciwnym wypadku mamy do czynienia z *grafem niespójnym*. Każdy graf niespójny G możemy przedstawić w postaci sumy grafów spójnych, które nazywamy *spójnymi składowymi* grafu.



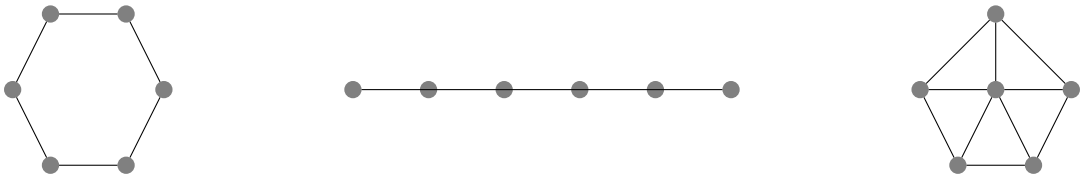
Rysunek 71.11: Spójne składowe grafu

W grafie spójnym możemy mówić o *odległości* pomiędzy dwoma dowolnymi wierzchołkami – jest to minimalna liczba krawędzi potrzebna, aby połączyć te dwa wierzchołki drogą. Krawędź, której usunięcie rozspójnia graf nazywamy *mostem*. Wierzchołek, którego usunięcie rozspójnia graf nazywamy *punktem artykulacji* [7].



Rysunek 71.12: Graf spójny

Grafy spójne, regularne stopnia 2 nazywamy *cyklicznymi* i oznaczamy C_n, gdzie n oznacza liczbę wierzchołków. Jeżeli z grafu C_n usuniemy jedną krawędź, to otrzymamy *graf liniowy* o n wierzchołkach, który oznaczamy symbolem P_n, natomiast *kołem* W_n nazywamy graf powstający z C_{n-1} poprzez dodanie nowego wierzchołka i połączenie z nim wszystkich dotychczasowych.



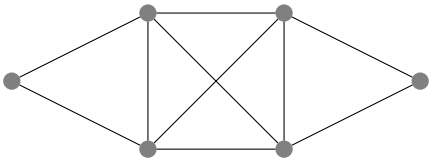
Rysunek 71.13: Graf cykliczny, liniowy i koło

71.6 Graf eulerowski, hamiltonowski

Graf spójny G nazywamy *eulerowskim*, jeśli istnieje zamknięta ścieżka zawierająca każdą krawędź G, a ścieżkę taką nazywamy *cyklem Eulera*. Jeśli istnieje ścieżka (nie będąca ścieżką zamkniętą) zawierająca każdą krawędź G, to mówimy, że G jest *grafem półeulerowskim*. Zauważmy, że „problem mostów królewieckich” jest równoważny pytaniu czy graf pokazany na rysunku 71.2 ma cykl Eulera. W tym celu Euler sformułował i udowodnił następujące twierdzenie:

Twierdzenie 71.1 (Euler, 1736)

Graf spójny jest grafem eulerowskim wtedy i tylko wtedy, gdy stopień każdego jego wierzchołka jest liczbą parzystą.



Rysunek 71.14: Przykład grafu eulerowskiego

Istnieje algorytm, który pozwala wyznaczyć cykl Eulera w grafie eulerowskim, zwany *algorytmem Fleury'ego*:

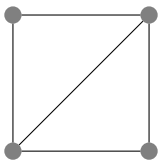
Twierdzenie 71.2

Jeżeli graf jest eulerowski, to następująca procedura jest wykonalna i tworzy w grafie cykl Eulera:
Zacznij cykl w dowolnym wierzchołku i przechodź krawędzie w dowolnej kolejności oraz:

1. *Usuwaj z grafu przechodzone krawędzie i wierzchołki izolowane powstające w wyniku usuwania tych krawędzi;*
2. *Przechodź przez most*[a] *tylko wtedy, gdy nie masz innej możliwości.*

[a] Mowa tu o *moście* w znaczeniu wprowadzonym w poprzedniej sekcji (krawędź rozspójniająca graf).

Dla grafu spójnego można sformułować pytanie analogiczne, jak w przypadku cyklu Eulera, ale w kontekście wierzchołków: czy istnieje ścieżka przechodząca dokładnie raz przez każdy jego wierzchołek? Taka ścieżka musi być cyklem, chyba że graf jest N_1. Cykl taki nazywamy *cyklem Hamiltona*, a sam graf *grafem hamiltonowskim*. Analogicznie, *grafem półhamiltonowskim* nazwiemy graf, w którym istnieje droga przechodząca przez wszystkie wierzchołki.

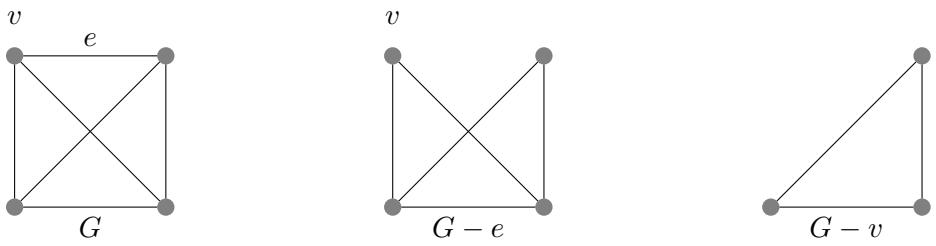


Rysunek 71.15: Przykład grafu hamiltonowskiego

Znalezienie warunku koniecznego i wystarczającego na istnienie cyklu Hamiltona, podobnie jak to jest dla cyklu Eulera, jest jednym z najważniejszych i nierozwiązanych problemów teorii grafów. Prawdopodobnie warunek taki nie istnieje, podobnie jak algorytm, który wyznaczałby cykl Hamiltona w czasie wielomianowym.

71.7 Podgrafy

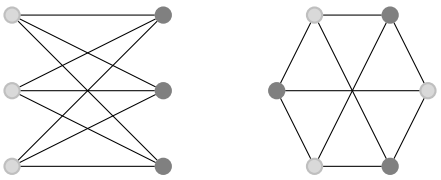
Podgrafem grafu G nazywamy graf, którego wszystkie wierzchołki należą do $V(G)$, a krawędzie należą do rodziny $E(G)$. Jeśli usuniemy niektóre krawędzie i wierzchołki grafu G, to otrzymamy jego podgraf. Oznaczamy to jako $G-e$, gdzie $e \in E(G)$ lub ogólnie $G-F$ dla $F \subset E(G)$. Podobnie, jeśli v jest wierzchołkiem grafu G, to $G-v$ oznacza graf powstający z G poprzez usunięcie wierzchołka v i wszystkich krawędzi z nim incydentnych z v. Ogólnie, $G-S$ oznacza graf G, z którego usunięto wszystkie wierzchołki należące do zbioru $S \subset V(G)$ oraz wszystkie krawędzie z nimi incydentne.



Rysunek 71.16: Graf i jego podgrafy

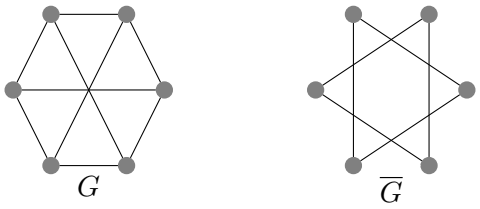
71.8 Grafy dwudzielne

Niech zbiory A i B będą zbiorami rozłącznymi oraz $V = A \cup B$ zbiorem wierzchołków grafu G. Jeśli każda krawędź G łączy wierzchołek ze zbioru A z wierzchołkiem ze zbioru B, to taki graf nazywamy *dwudzielnym*. Możemy tę własność równoważnie wyrazić za pomocą kolorów: jeżeli wierzchołki grafu da się pokolorować dwoma kolorami, np. białym i czarnym w taki sposób, że każda krawędź grafu łączy wierzchołki o przeciwnym kolorze, to jest to graf dwudzielny. Jeśli każdy wierzchołek zbioru A jest połączony dokładnie jedną krawędzią z każdym wierzchołkiem zbioru B, to jest to *graf pełny dwudzielny*. Taki graf mający r czarnych i s białych wierzchołkach oznaczamy $K_{r,s}$. Każdy graf $K_{r,s}$ posiada $r + s$ wierzchołków oraz rs krawędzi.



Rysunek 71.17: Grafy dwudzielne

Jeżeli G jest grafem prostym, to jego *dopełnieniem* jest graf prosty $\overline{G}$ taki, że jego zbiorem wierzchołków jest również $V(G)$, w którym dwa wierzchołki są sąsiednie wtedy, gdy nie są sąsiednie w grafie G. Dopełnieniem grafu pełnego jest graf pusty, a dopełnieniem grafu pełnego dwudzielnego jest suma dwóch grafów pełnych.

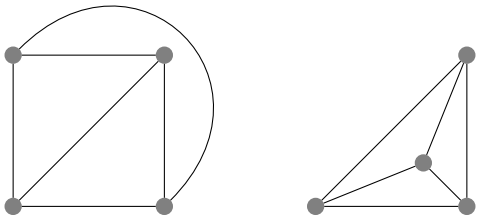


Rysunek 71.18: Graf prosty i jego dopełnienie

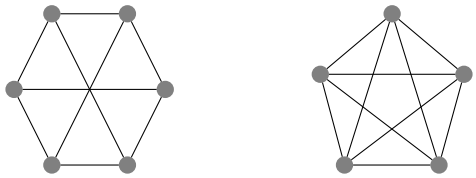
71.9 Grafy planarne

Graf nazywamy *planarnym*, gdy można go narysować na płaszczyźnie w taki sposób, aby krawędzie nie przecinały się (poza wierzchołkami). Każdy taki rysunek nazywamy *rysunkiem płaskim* lub po prostu *grafem płaskim*. Aby zatem sprawdzić czy dany graf jest planarny, musimy sprawdzić czy jest on izomorficzny z jakimś grafem płaskim. Wszystkie grafy planarne spełniają przedstawioną wcześniej formułę Eulera (Lemat 71.2), a w szczególności są nimi grafy platońskie (Rys. 71.6).

Wiadomo, że grafy $K_{3,3}$ i K_5 są nieplanarne:



Rysunek 71.19: Rysunki płaskie grafu planarnego K_4



Rysunek 71.20: Grafy $K_{3,3}$ i K_5

Mówimy, że graf G jest rozszerzeniem grafu H, jeżeli powstał poprzez wstawienie pewnej liczby dodatkowych wierzchołków „w środek krawędzi" grafu H. Graf G można zatem sprowadzić do H poprzez usunięcie pewnej liczby wierzchołków stopnia 2. Z kolei dwa grafy nazwiemy *homeomorficznymi*, jeśli można je otrzymać z tego samego grafu poprzez wstawienie wierzchołków stopnia 2 wewnątrz jego krawędzi. Wstawienie lub usunięcie wierzchołków stopnia 2 nie ma wpływu na planarność grafu. Dzięki temu możemy jednak sformułować zaskakujące twierdzenie, autorstwa polskiego matematyka *Kazimierza Kuratowskiego*:

Twierdzenie 71.3 (Kuratowski, 1930)

Dany graf jest planarny wtedy i tylko wtedy, gdy nie zawiera podgrafu homeomorficznego z grafem K_5 lub z grafem $K_{3,3}$.

♣

Jeśli zamiast pojęcia homeomorfizmu użyjemy rozszerzenia, wtedy twierdzenie Kuratowskiego przyjmuje postać:

Twierdzenie 71.4

Skończony graf jest planarny wtedy i tylko wtedy, gdy nie zawiera podgrafu, który jest rozszerzeniem grafu $K_{3,3}$ lub grafu K_5.

♣

Aby zatem sprawdzić planarność grafu, należy usunąć z niego wszystkie wierzchołki stopnia 2 i sprawdzić czy w nowym grafie nie ma podgrafu $K_{3,3}$ lub K_5.

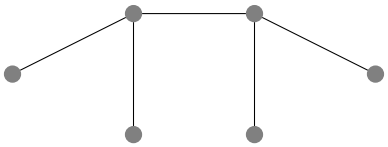
Własność planarności grafu ma zastosowanie m.in. w architekturze przy planowaniu dostępności pomieszczeń. Jeżeli odpowiedni graf jest planarny, to można wszystkie pomieszczenia rozplanować na jednym piętrze (w przeciwnym wypadku, konieczne będzie wprowadzenie dodatkowych pięter i schodów).

71.10 Drzewa

Grafy spójne, w których każdą parę wierzchołków można połączyć dokładnie jedną drogą nazywamy *drzewami*. A zatem drzewo, to spójny graf acykliczny. Jak można zauważyć, dodanie dowolnej krawędzi do drzewa utworzy

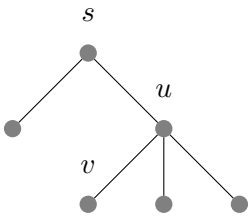
cykl, a usunięcie dowolnej krawędzi z drzewa je rozspójni – w obu przypadkach przestanie być drzewem.

Lasem nazywamy graf nie zawierający cykli. Każda jego spójna składowa jest zatem drzewem (a jeśli las składa się z jednej spójnej składowej, to po prostu jest drzewem).



Rysunek 71.21: Przykład drzewa

Drzewa mogą mieć strukturę hierarchiczną, z jednym wyróżnionym wierzchołkiem s zwanym *korzeniem*, umieszczonym na górze, i pozostałymi *potomkami* „rozgałęziającymi" się ku dołowi (p. omówienie zadania „Kto stoi przede mną" dla Gremlinów w dziale III). Wówczas wierzchołek u, z którego „doszliśmy" do wierzchołka v, nazywamy jego *poprzednikiem* lub *rodzicem*, zaś v jego *potomkiem* lub *synem*. Wierzchołki pozbawione potomków nazywamy *liśćmi*.



Rysunek 71.22: Drzewo ukorzenione

Zauważmy, że dowolny wierzchołek drzewa może być wybrany jako korzeń, a wybór ten będzie decydować o relacjach rodzic-potomek między wszystkimi wierzchołkami.

Drzewo o n wierzchołkach ma zawsze $n-1$ krawędzi, skąd otrzymujemy następujący wniosek:

Podsumowanie 71.1

Jeśli graf G jest lasem, który ma n wierzchołków i k składowych, to G ma $n-k$ krawędzi. ♡

Rozważmy spójny graf G, który zawiera cykle. Jeśli usuniemy z niego krawędź należącą do cyklu, to nadal pozostanie spójny. Możemy powtarzać tę operację, dopóki w G nie będzie już cykli. Powstanie wówczas drzewo, które „spina" wszystkie krawędzie grafu – nazywamy je *drzewem rozpinającym* (lub *spinającym*) graf G:

Rysunek 71.23: Graf i jego drzewo rozpinające (jedno z możliwych)

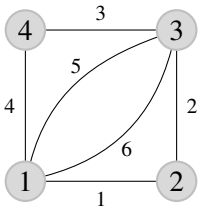
Liczba możliwych drzew rozpinających grafu może być bardzo duża. W przypadku grafu pełnego o n wierzchołkach (oznaczonych) istnieje n^{n-2} różnych drzew rozpinających, co wynika z *Twierdzenia Cayleya* (mówiącego, że liczba różnych n-wierzchołkowych drzew oznaczonych wynosi n^{n-2}) [21].

71.11 Reprezentacja grafu w komputerze

Istnieją różne sposoby na przechowywanie grafu w pamięci komputera, a najpopularniejsze z nich to: *macierz sąsiedztwa*, *macierz incydencji* oraz *lista sąsiedztwa*. Wszystkie trzy można stosować zarówno do grafów skierowanych, jak i nieskierowanych.

Macierz sąsiedztwa $\mathbf{A}$ grafu o wierzchołkach oznaczonych liczbami $\{1, 2, ..., n\}$, jest dwuwymiarową tablicą o wymiarach $n \times n$, w której na przecięciu i-tego wiersza oraz j-tej kolumny została umieszczona wartość $k \in \mathbb{N} \cup \{0\}$ oznaczająca liczbę krawędzi pomiędzy wierzchołkami o numerach i, j. Zauważmy, że jeśli graf jest nieskierowany, to macierz sąsiedztwa jest macierzą symetryczną (bo jeśli węzeł A jest połączony krawędzią z węzłem B, to tym samym węzeł B jest połączony z węzłem A).

Jeżeli w grafie o n wierzchołkach, krawędzie również oznaczymy liczbami $\{1, 2, ..., m\}$, to macierz $\mathbf{M}$ o wymiarach $n \times m$, w której wartości 1 albo 0 na pozycji (i, j) oznaczają odpowiednio: incydencję wierzchołka i z krawędzią j lub jej brak — nazywamy *macierzą incydencji*. W przypadku grafu skierowanego będziemy dodatkowo używać liczby -1, która oznaczać będzie, że krawędź j wychodzi z wierzchołka i, zaś 1, gdy do niego wchodzi.



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Rysunek 71.24: Przykładowy graf nieskierowany oraz jego macierz sąsiedztwa (A) i macierz incydencji (B)

Reprezentację polegającą na podaniu listy wierzchołków sąsiednich dla każdego wierzchołka grafu nazywamy *listą sąsiedztwa* grafu:

1:	2	3	3	4
2:	1	3		
3:	1	1	2	4
4:	1	3		

Tabela 71.1: Lista sąsiedztwa grafu z poprzedniego rysunku

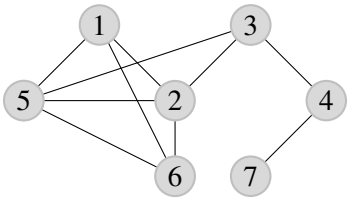
Wybór reprezentacji grafu może mieć istotny wpływ na złożoność obliczeniową i czas działania naszego algorytmu. Należy tu wziąć pod uwagę zarówno jakie operacje w nim dominują oraz jaka jest charakterystyka samego grafu. Przykładowo, sprawdzenie czy dwa konkretne wierzchołki są połączone krawędzią jest natychmiastowe w przypadku macierzy sąsiedztwa (złożoność $O(1)$, czyli w czasie stałym), natomiast lista sąsiedztwa wymaga w tym celu przeszukania wszystkich sąsiadów danego wierzchołka. W przypadku *grafów gęstych*, może

to prowadzić do złożoności $O(|V|)$, gdzie $|V|$ oznacza liczbę wierzchołków[2]. W przypadku *grafów rzadkich*[3], to jednak lista sąsiedztwa okazuje się zwykle bardziej przydatna (dla grafów rzadkich o bardzo dużej liczbie wierzchołków, macierz sąsiedztwa może nawet nie zmieścić się w pamięci operacyjnej).

71.12 Przeszukiwanie grafu

Zazwyczaj kiedy potrzebujemy uzyskać z grafu jakąś konkretną informację, musimy umieć przeszukać go w sposób systematyczny. Służą do tego dwa równoważne co do złożoności czasowej i pamięciowej algorytmy: *przeszukiwanie w głąb* (*depth-first search*, w skrócie DFS) oraz *przeszukiwanie wszerz* (*breadth-first search*, czyli BFS) [4][2][7][17]. Obie metody pozwalają na odwiedzenie wszystkich wierzchołków, choć w różnej kolejności, dlatego często można je stosować zamiennie. Złożoności: obliczeniowa i pamięciowa są równe i takie same dla algorytmów DFS i BFS oraz zależne od sposobu jaki wybierzemy dla reprezentacji grafu. W przypadku macierzy sąsiedztwa będzie to $O(n^2)$, a dla listy sąsiedztwa $O(n+m)$, gdzie n oznacza liczbę wierzchołków grafu, a m liczbę jego krawędzi. Jedyna różnica w implementacji tych algorytmów polega na zastosowaniu innych struktur danych do przechowywania kolejnych wierzchołków do „odwiedzenia". DFS wykorzystuje *stos*, tj. strukturę typu LIFO, zaś BFS *kolejkę*, czyli FIFO (p. wprowadzenie do działu III). Przeszukiwanie grafu $G = (V, E)$ zaczynamy od wyróżnionego wierzchołka s zwanego dalej *źródłem*, a każdy nowo napotkany wierzchołek staje się *odwiedzony*.

Algorytmy BFS i DFS omówimy na przykładzie poniższego grafu:



1: 2 5 6
2: 1 3 5 6
3: 2 4 5
4: 3 7
5: 1 2 3 6
6: 1 2 5
7: 4

Rysunek 71.25: Przykładowy graf oraz jego lista sąsiedztwa

71.12.1 BFS

Podczas przeszukiwania *wszerz*, zanim bardziej zagłębimy się w grafie, odwiedzamy tak wiele wierzchołków, jak to jest możliwe. To znaczy, że sprawdzamy wszystkie wierzchołki sąsiadujące z danym wierzchołkiem, zanim przejdziemy do jakiegokolwiek wierzchołka położonego dalej. Wynikiem działania algorytmu jest również

[2]Przez *graf gęsty* rozumiemy graf o bardzo dużej liczbie krawędzi – rzędu $|V|^2$.

[3]W *grafie rzadkim* liczba krawędzi jest podobnego rzędu, co liczba wierzchołków.

drzewo przeszukiwania wszerz o korzeniu w s zawierające wszystkie wierzchołki, do których można dotrzeć. Droga z korzenia do dowolnego wierzchołka v w tym drzewie odpowiada *najkrótszej ścieżce* między tymi wierzchołkami w grafie G. Bariera pomiędzy wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana „na całej szerokości", dlatego zanim odwiedzimy wierzchołki w odległości $k+1$ od źródła, najpierw odwiedzimy wszystkie w odległości k.

Przykład implementacji algorytmu BFS w języku C++:

```
vector<int> G[10001];   // graf jako lista sasiedztwa (tu: tablica wektorów)
int visited[10001]={0};

void BFS(int n) {
        queue<int> Q;
        visited[n]=1;
        Q.push(n);
        while(!Q.empty()) {
                int v = Q.front();
                cout << "Przetwarzam wierzchołek nr " << v << endl;
                Q.pop();
                for(int i = 0; i < (int)G[v].size(); i++) {
                        if(!visited[G[v][i]]) {
                                visited[G[v][i]] = 1;
                                Q.push(G[v][i]);
                        }
                }
        }
}
```

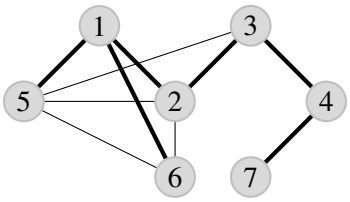
W przypadku naszego przykładowego grafu, algorytm BFS po przetworzeniu węzła startowego $s = 1$ dodaje do kolejki jego sąsiadów (2, 5, 6). Pierwszym węzłem w kolejce staje się wówczas węzeł 2, więc w kolejnej iteracji pobieramy go i przetwarzamy, a do kolejki dodajemy jego sąsiadów, a w zasadzie tylko sąsiada 3 (bo sąsiedzi 1, 5, 6 są już zaznaczeni jako odwiedzeni). Następnie z kolejki pobieramy i przetwarzamy węzeł 5 oraz 6, a dalej 3 (dodając do kolejki węzeł 4), potem 4 (dodając 7) i na końcu 7.

Podsumowując, zawartość kolejki w kolejnych przebiegach pętli `while` wygląda następująco:

1		
2	5	6
5	6	3
6	3	
3		
4		
7		

Ponieważ elementy pobierane są z początku kolejki, więc zawartość pierwszej kolumny pokazuje nam, w jakiej kolejności przetwarzane były węzły grafu (1, 2, 5, 6, 3, 4, 7). Poniżej przedstawiono nasz przykładowy graf z za-

znaczonym drzewem przeszukiwania wszerz (pogrubione krawędzie), uzyskanym dla wierzchołka startowego w węźle numer 1.



Rysunek 71.26: Drzewo przeszukiwania wszerz uzyskane dla grafu przykładowego

71.12.2 DFS

Przeszukując *w głąb* zagłębiamy się tak daleko, jak to tylko możliwe, zanim odwiedzimy wierzchołki sąsiednie. Badane są krawędzie ostatnio odwiedzonego wierzchołka, dopóki wychodzą z niego niezbadane dotychczas krawędzie. Wierzchołek uznajemy za *przetworzony*, gdy jego lista sąsiedztwa zostanie całkowicie zbadana. Krawędzie i wierzchołki, po których „poruszał się" algorytm DFS tworzą drzewo przeszukiwania w głąb.

Poniżej iteracyjna implementacja przeszukiwania w głąb w języku C++:

```
vector<int> G[10001];   // graf jako lista sasiedztwa (tu: tablica wektorów)
int visited[10001]={0};
void DFS(int n) {
        stack<int> S;
        visited[n]=1;
        S.push(n);
        while(!S.empty()) {
                int v = S.top();
                S.pop();
                cout << "Przetwarzam wierzchołek nr " << v << endl;
                for(int i = (int)G[v].size() - 1; i >= 0; i--)
                        if(!visited[G[v][i]]) {
                                visited[G[v][i]] = 1;
                                S.push(G[v][i]);
                        }
        }
}
```

Kod ten jest, jak widać, bardzo podobny jak w przypadku BFS. Główna różnica, to użycie stosu zamiast kolejki[4]. Oczywiście, zamiast jawnego użycia stosu można wykorzystać (niejawnie) stos systemowy, stosując rekurencję:

```
void DFS_rec(int v) {
        cout << "Przetwarzam wierzchołek nr " << v << endl;
        visited[v] = 1;
        for(int i = 0; i < (int)G[v].size(); i++)
                if (!visited[G[v][i]]) DFS_rec(G[v][i]); // tu wywołanie rekurencyjne
}
```

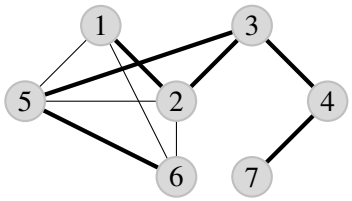
[4]Zauważmy też, że sąsiedzi dodawani są do stosu w pętli for „od końca", gdyż samo użycie stosu odwraca kolejność elementów.

Podczas przeszukiwania w głąb, wierzchołki naszego przykładowego grafu są przetwarzane w następującej kolejności: po przetworzeniu węzła startowego $s = 1$ kładziemy na stosie jego sąsiadów (6, 5, 2), po czym zdejmujemy leżący na wierzchu węzeł 2, przetwarzamy go i kładziemy na stosie jego sąsiadów, a w zasadzie tylko sąsiada 3 (bo sąsiedzi 1, 5, 6 byli już zaznaczeni jako odwiedzeni). Teraz widać różnicę w stosunku do BFS, bo kolejnym węzłem do przetworzenia jest leżący na wierzchu (dodany przed chwilą) węzeł 3[5]. Jedynym nieodwiedzonym sąsiadem 3, którego możemy położyć na stosie jest węzeł 4, po czym od razu go zdejmujemy i kładziemy na stos jego sąsiada 7. Dopiero po przetworzeniu węzła 7 zdejmujemy wreszcie leżące na samym dnie elementy 5 i 6 (położone w pierwszym przebiegu pętli `while`).

Reasumując, zawartość stosu w kolejnych przebiegach pętli `while` wygląda następująco:

	2	3	4	7		
	5	5	5	5	5	
1	6	6	6	6	6	6

Elementy pobierane są zawsze z wierzchu stosu, więc kolejność przetwarzania węzłów określimy patrząc w każdym kroku na element leżący najwyżej (tu węzły przetwarzane były w kolejności: 1, 2, 3, 4, 7, 5, 6). Poniżej przedstawiono nasz przykładowy graf z zaznaczonym drzewem przeszukiwania w głąb (pogrubione krawędzie), uzyskanym dla wierzchołka startowego w węźle numer 1.



Rysunek 71.27: Drzewo przeszukiwania w głąb uzyskane dla grafu przykładowego

Na koniec warto zauważyć, że obie przedstawione metody można wykorzystać do uzyskania (różnych) drzew rozpinających grafu. Jeśli graf byłby niespójny, otrzymalibyśmy – zamiast drzewa – *las przeszukiwania* w głąb lub wszerz. Oczywiście, w praktyce należałoby w tym celu „uruchomić" nasz algorytm BFS lub DFS wielokrotnie – startując kolejno w każdym nieodwiedzonym jeszcze węźle w grafie. Niejako przy okazji uzyskalibyśmy w ten sposób informację, z ilu spójnych składowych składa się graf.

71.13 Najkrótsze ścieżki

Zadanie szukania najkrótszych ścieżek w grafie jest jednym z najbardziej typowych problemów o dużym znaczeniu praktycznym [2][4][7]. Zagadnienie to można podzielić na cztery warianty, w których interesują nas:

1. najkrótsze ścieżki z pojedynczego źródła do wszystkich wierzchołków;
2. najkrótsze ścieżki z jednym wierzchołkiem docelowym;
3. najkrótsze ścieżki między jedną parą wierzchołków;
4. najkrótsze ścieżki między wszystkimi parami wierzchołków.

[5] W algorytmie BFS węzeł 3 był dodany na końcu kolejki, więc musiał czekać dłużej, aż się nim zajmiemy.

Skupimy się tylko na przypadku pierwszym ponieważ, aby rozwiązać pozostałe, zazwyczaj trzeba wyznaczyć najkrótsze ścieżki z pojedynczego źródła do wszystkich innych wierzchołków w grafie. W poniższych rozważaniach przyjmujemy, że wszystkie grafy reprezentowane są za pomocą list sąsiedztwa. Zbiór sąsiadów danego wierzchołka u będziemy oznaczać jako $Adj[u]$.

71.13.1 Grafy bez wag

Weźmy pod uwagę dowolny graf $G = (V, E)$ bez wag albo taki, w którym waga każdej krawędzi wynosi tyle samo – skierowany lub nieskierowany. Dla każdego wierzchołka $v \in V$ osiągalnego ze źródła $s \in V$ istnieje ścieżka łącząca s i v:

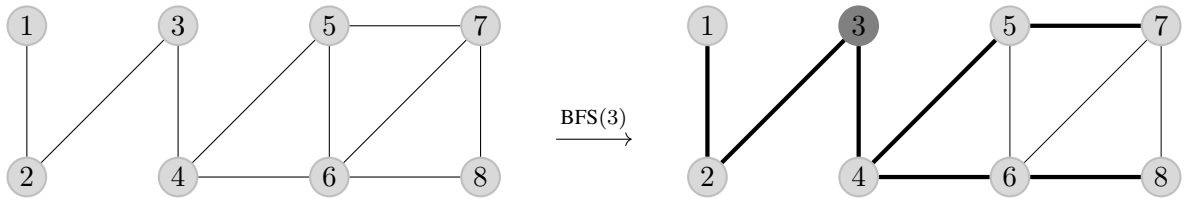
$$p = (sv_1, v_1v_2, ..., v_kv), \text{ gdzie } \forall_{1 \leq i \leq k}\ v_i \in V \text{ dla } k \in \mathbb{N}\,.$$

Jako *odległość* v od s będziemy rozumieć *długość* $|p|$ w sensie definicji ścieżki (p. sekcja 71.4), czyli liczbę krawędzi ją tworzących (leżących na ścieżce p). Zatem:

$$|p| = k + 1\,.$$

Najkrótszą ścieżką od źródła s do wierzchołka v będzie każda ścieżka, która wyznacza najmniejszą odległość między s i v, czyli zawiera minimalną liczbę krawędzi.

Jak można zauważyć, do znalezienia najkrótszych ścieżek z pojedynczego źródła w przypadku grafu bez wag wystarczy wykorzystać przedstawiony wcześniej algorytm przeszukiwania wszerz (algorytm BFS – sekcja 71.12.1). Podczas jego wykonania niejako automatycznie znajdziemy najmniejsze odległości między wierzchołkiem startowym, a pozostałymi wierzchołkami, do których można dotrzeć:



Rysunek 71.28: Najkrótsze ścieżki z węzła nr 3 do wszystkich pozostałych w grafie nieważonym

71.13.2 Grafy z wagami

W przypadku grafów ważonych, pojęcia długości ścieżki nie będziemy utożsamiać z liczbą krawędzi leżących na ścieżce, ale z *wagą ścieżki*, która jest sumą wag tworzących ją krawędzi. Dla uproszczenia oznaczmy ścieżkę p w grafie ważonym $G = (V, E)$ z przypisaną mu funkcją wagową $w : E \longrightarrow \mathbb{R}$ jako ciąg wierzchołków incydentnych z krawędziami ją tworzącymi:

$$p = (v_0, v_1, ..., v_k), \text{ gdzie } \forall_{0 \leq i \leq k}\ v_i \in V \text{ dla } k \in \mathbb{N}\,.$$

Wtedy waga ścieżki p będzie równa:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)\,,$$

gdzie $w(v_{i-1}, v_i)$ oznacza wagę krawędzi łączącej wierzchołki v_{i-1} oraz v_i.

Zatem, najkrótszą ścieżką od źródła $s \in V$ do wierzchołka $v \in V$ będzie teraz każda ścieżka, której waga jest najmniejsza, a jej wartość oznaczamy jako $\delta(s, v)$. Jeżeli taka ścieżka nie istnieje, to $\delta(s, v) = \infty$. Natomiast, jeśli na pewnej ścieżce z s do v znajduje się cykl o ujemnej wadze, to $\delta(s, v) = -\infty$, ponieważ zawsze da się znaleźć krótszą ścieżkę od już znalezionej – wystarczy jeszcze raz przejść przez ujemny cykl. Gdy istnieją ujemne wagi w grafie, ale nie zawiera on ujemnych cykli, to wartości $\delta(s, v_i)$ są określone dla każdego $v_i \in V$ osiągalnego z s.

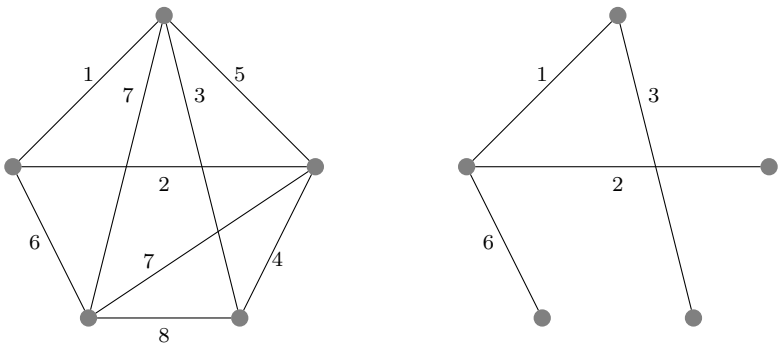
71.13.2.1 Minimalne drzewo spinające grafu

Mówiąc o najkrótszych ścieżkach w grafie, należy wspomnieć również o tzw. *minimalnym drzewie spinającym* (ang. *minimum spanning tree*, MST) [4][2], które omówimy tu na podstawie następującego przykładu:

Załóżmy, że w danym państwie istnieje sieć głównych dróg, która łączy ze sobą wszystkie miasta-stolice dystryktów administracyjnych oraz znamy długość każdej drogi między sąsiednimi miastami. Państwowe przedsiębiorstwo transportowe chce wyznaczyć siatkę połączeń komunikacyjnych między nimi tak, aby jej sumaryczna długość była jak najkrótsza. Z punktu widzenia teorii grafów, sieć dróg to graf nieskierowany, gdzie każda krawędź $e = (u, v) \in E$ oznacza drogę pomiędzy parą miast $u, v \in V$, której przyporządkowujemy wagę $w(e)$ równą odległości między nimi. Wyznaczenie siatki połączeń między miastami można wówczas utożsamić z problemem znalezienia drzewa spinającego $T = (V, F \subseteq E)$, którego waga:

$$W(T) = \sum_{e \in F} w(e)\,,$$

jest najmniejsza spośród wag wszystkich możliwych drzew łączących stolice (jak pamiętamy, drzew takich istnieje maksymalnie n^{n-2}, w przypadku sieci dróg będącej grafem pełnym). Efektywne rozwiązanie tego problemu polega na zastosowaniu *algorytmu zachłannego*, w którym zawsze wybieramy krawędź o najmniejszej wadze tak, aby nie utworzył się cykl. Do klasycznych algorytmów konstrukcji minimalnego drzewa spinającego należą algorytmy Prima oraz Kruskala (zainteresowanego czytelnika odsyłamy do literatury [4]).



Rysunek 71.29: Graf i jego minimalne („najlżejsze”) drzewo spinające

Można tu zauważyć, że minimalne drzewo spinające nie stanowi uniwersalnego rozwiązania problemu wyznaczania najkrótszych ścieżek z dowolnego węzła do wszystkich innych. Przykładowo, drzewo z powyższego rysunku zawiera wprawdzie najkrótsze ścieżki prowadzące ze skrajnie lewego węzła do pozostałych, ale nie zawiera np. najkrótszej ścieżki łączącej dwa dolne węzły (o wadze 8), zmuszając nas do „podróży dookoła” trasą o łącznej wadze $6 + 1 + 3 = 10$. Drzewo to nie mogłoby zatem reprezentować rozwiązania problemu najkrótszych ścieżek, jeśli za węzeł źródłowy przyjęlibyśmy którykolwiek z dolnych węzłów.

W kolejnych sekcjach przedstawimy dwa klasyczne algorytmy znajdowania najkrótszej ścieżki z wybranego

węzła do wszystkich pozostałych w grafie ważonym, zakładając, że jest to graf skierowany (digraf). W przypadku grafu nieskierowanego wystarczy najpierw przekształcić go do postaci digrafu, zastępując każdą krawędź *parą krawędzi* o przeciwnych zwrotach i takiej samej wadze.

71.13.2.2 Inicjalizacja węzłów

Przyjmijmy, że każdy wierzchołek $v \in V$ w grafie ważonym $G = (V, E)$ z funkcją wagową $w : E \longrightarrow \mathbb{R}$, będzie „pamiętał" swojego *poprzednika*, tj. sąsiedni wierzchołek, z którego dotarliśmy do v, konstruując ścieżkę od źródła $s \in V$. Ponieważ naszym celem jest zbudowanie pewnego drzewa spinającego o korzeniu w s, to zgodnie z terminologią dotyczącą drzew, tym sąsiednim wierzchołkiem będzie rodzic wierzchołka v, więc oznaczać go będziemy jako $v.parent$. Jeżeli wierzchołek v nie posiada poprzednika, wtedy $v.parent = NIL$. Kolejną informacją przechowywaną przez każdy wierzchołek, będzie wartość aktualnie znanej najmniejszej jego odległości od źródła. Oznaczmy ją przez $v.dist$. Przed rozpoczęciem procesu wyszukiwania najkrótszych ścieżek od źródła s, przypisujemy każdemu wierzchołkowi (z wyjątkiem samego źródła) nieskończoną odległość od źródła oraz informację o braku poprzednika.

Algorithm 1 Inicjalizacja pojedynczego źródła

Require: Initialize(G, s)
1: **for** $v_i \in V$ **do**
2: $v_i.dist = \infty$;
3: $v_i.parent = NIL$;
4: **end for**
5: $s.dist = 0$

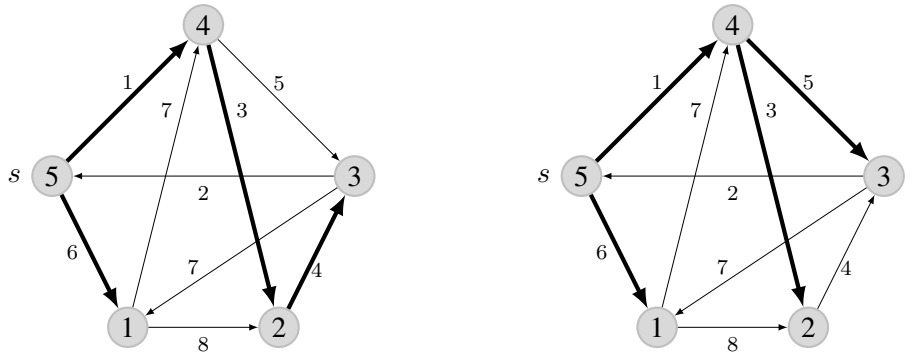
71.13.2.3 Relaksacja krawędzi

Podstawą znanych algorytmów do wyszukiwania najkrótszych ścieżek jest procedura *relaksacji krawędzi*, która polega na zmniejszeniu znanej odległości $v.dist$ danego wierzchołka v od źródła, jeśli pojawi się nowa ścieżka od s do v o mniejszej wadze. Zmianie ulega wówczas poprzednik v, którym staje się jego sąsiad $u \in V$ leżący na nowej ścieżce, a nową odległość obliczamy sumując odległość u od źródła oraz wagę krawędzi łączącej u z v:

Algorithm 2 Relaksacja krawędzi

Require: Relax(u, v, w)
1: **if** $v.dist > u.dist + w(u, v)$ **then**
2: $v.dist := u.dist + w(u, v)$;
3: $v.parent := u$;
4: **end if**

Na kolejnym rysunku pogrubionymi strzałkami przedstawiono przykładowe, dotychczas poznane (jeszcze nie ostateczne) najkrótsze ścieżki między wierzchołkiem źródłowym (nr 5), a wszystkimi innymi w pewnym grafie skierowanym. Wykonanie procedury relaksacji dla wierzchołka v nr 3, doprowadziło do zmiany jego rodzica u (z wierzchołka 2 na 4) i do zmniejszenia jego odległości od źródła z $1 + 3 + 4 = 8$ na $1 + 5 = 6$.



v	1	2	3	4	5
$dist$	6	4	8	1	0
$parent$	5	4	2	5	NIL

Relax$(4, 3, w)$ $\longrightarrow$

v	1	2	3	4	5
$dist$	6	4	6	1	0
$parent$	5	4	4	5	NIL

Rysunek 71.30: Relaksacja krawędzi

71.13.2.4 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda [4] służy do obliczania najkrótszych ścieżek z pojedynczego źródła s do wszystkich pozostałych wierzchołków w digrafie ważonym, który nie posiada ujemnych cykli. Jeśli takowe istnieją, to algorytm je wykryje. Zasada jego działania opiera się na powtórzeniu relaksacji wszystkich krawędzi $|V| - 1$ razy. Poprawność tej procedury możemy uzasadnić następująco: po pierwszym wykonaniu relaksacji wszystkich krawędzi mamy policzone najkrótsze ścieżki od źródła s do wierzchołków oddalonych od niego o 1 krawędź, po drugiej iteracji – dla wierzchołków odległych o 2, po trzeciej – o 3, ... itd. Najdalszy „krąg" wierzchołków będzie oddalony od źródła o $|V| - 1$ krawędzi, dlatego po wykonaniu tylu powtórzeń będziemy mieli policzone najkrótsze ścieżki do wszystkich wierzchołków. Natomiast wykonanie dodatkowego, $|V|$-tego powtórzenia będzie testem na istnienie ujemnych cykli w grafie. Jeśli istnieją – to po wykonaniu tej iteracji uda się jeszcze coś zrelaksować (`return FALSE`); w przeciwnym wypadku, każde kolejne powtórzenie relaksacji wszystkich krawędzi niczego już nie zmieni (`return TRUE`). Algorytm Bellmana-Forda działa w czasie $O(|V||E|)$.

Algorithm 3 Algorytm Bellmana-Forda

Require: Bellman-Ford(G, w, s)
1: Initialize(G, s)
2: **for** $k := 1$ **to** $|V| - 1$ **do**
3: **for** $(u_i, v_j) \in E$ **do**
4: Relax(u_i, v_j, w)
5: **end for**
6: **end for**
7: **for** $(u_i, v_j) \in E$ **do**
8: **if** $v_j.dist > u_i.dist + w(u_i, v_j)$ **then**
9: **return** FALSE
10: **end if**
11: **end for**
12: **return** TRUE

71.13.2.5 Algorytm Dijkstry

Podobnie jak algorytm Bellmana-Forda, *algorytm Dijkstry* [4][23] służy do rozwiązania problemu najkrótszych ścieżek z pojedynczego źródła s do wszystkich wierzchołków w digrafie ważonym, lecz którego funkcja wagowa posiada tylko wartości nieujemne $w : E \longrightarrow \mathbb{R}^+ \cup \{0\}$. Ten algorytm również opiera się na wielokrotnym zastosowaniu operacji relaksacji krawędzi, ale jest ona wykonywana zawsze tylko raz dla każdego v_i ze zbioru wszystkich sąsiadów $Adj[u]$ danego wierzchołka u, po czym zasila on zbiór S wierzchołków, dla których najkrótsza ścieżka została obliczona. Wierzchołki do sprawdzenia przechowywane są w kolejce priorytetowej Q, skąd po przetworzeniu są usuwane. Po zakończeniu tej procedury kolejka będzie pusta, zaś zbiór $S = V$.

Algorithm 4 Algorytm Dijkstry

Require: Dijkstra(G, w, s)
1: Initialize(G, s)
2: $Q := V$
3: **while** $Q \neq \emptyset$ **do**
4: $u :=$ min(Q) {Pobierz pierwszy wierzchołek – o najmniejszej odległości od źródła}
5: $Q \setminus \{u\}$ {Usuń go z kolejki}
6: **for** $v_i \in Adj[u]$ **do**
7: Relax(u, v_i, w)
8: **end for**
9: **end while**

Kolejkę priorytetową stosujemy tu, aby zapewnić, że do przetworzenia zawsze brany będzie „najlżejszy" z wierzchołków czekających w kolejce. Złożoność obliczeniowa algorytmu Dijkstry uzależniona jest od sposobu implementacji kolejki priorytetowej i waha się od $O(|V|^2 + |E|) = O(|V|^2)$ do $O(|V|\log|V| + |E|)$ w przypadku zastosowania kopca Fibonacciego [4]. Algorytm Dijkstry jest przykładem algorytmu zachłannego.

71.14 Przepływy

Rozważmy sytuację, w której w centrum pewnego miasta znajdują się wyłącznie drogi jednokierunkowe, z których każda posiada pewne maksymalne dopuszczalne natężenie ruchu liczone jako liczba pojazdów na godzinę. Chcielibyśmy wiedzieć ile aut maksymalnie może przejechać przez całe centrum w ciągu jednej godziny. Podobne pytanie możemy formułować w stosunku do sieci elektrycznej i przepływającego przez nią prądu, cieczy płynącej w rurociągach od określonego źródła do ujścia, czy też przewozu towarów z fabryki do magazynu (lub wielu magazynów).

71.14.1 Sieci przepływowe

Powyższe problemy, sprowadzające się do zadania określenia tzw. *maksymalnego przepływu*, możemy zobrazować za pomocą grafów, które nazywać będziemy *sieciami przepływowymi* [4][21][17][23]. Zdefiniujemy zatem *sieć* jako digraf $G = (V, E)$, gdzie do każdego łuku $e = (u, v) \in E$, łączącego wierzchołki $u, v \in V$, przypisana jest wartość $\psi(e)$, którą nazywać będziemy *przepustowością* tego łuku. Sumę przepustowości łuków, których końcem jest wierzchołek x nazwiemy jego *stopniem wejściowym*:

$$\text{indeg}(x) = \sum_{u \in V} \psi(u, x)\,,$$

zaś sumę przepustowości łuków, dla których x jest początkiem nazwiemy jego *stopniem wyjściowym*:

$$\text{outdeg}(x) = \sum_{v \in V} \psi(x, v)\,.$$

Zauważmy, że suma stopni wyjściowych wszystkich wierzchołków musi być równa sumie stopni wyjściowych. Własność ta jest w istocie pewną modyfikacją omówionego wcześniej lematu o uściskach dłoni.

W sieci wyróżniamy dwa szczególne wierzchołki: *źródło* i *ujście*. W przedstawionym dalej przykładowym grafie są nimi odpowiednio wierzchołki s i t. W ogólności, możliwych jest wiele źródeł i wiele ujść.

Definicja 71.1

Przepływem w sieci $G = (V, E)$ ze źródłem s i ujściem t nazywamy funkcję ϕ, przypisującą każdemu łukowi $e \in E$ nieujemną liczbę rzeczywistą, którą nazywamy przepływem wzdłuż tego łuku taką, że:

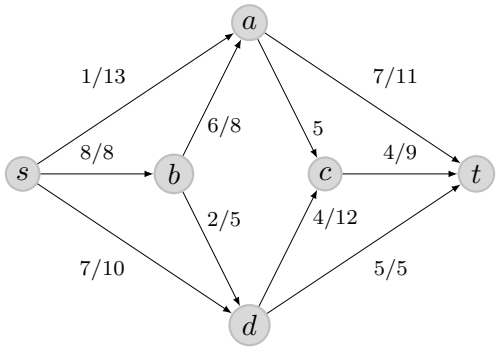
1. $\phi(e) \leq \psi(e)$
2. $\forall x \in V \setminus \{s, t\} \quad \sum_{e_{in}(x)} \phi(e_{in}(x)) = \sum_{e_{out}(x)} \phi(e_{out}(x))$

gdzie $e_{in}(x)$ oraz $e_{out}(x)$ oznaczają odpowiednio: krawędzie wchodzące i wychodzące z wierzchołka x. ♣

A zatem przepływ wzdłuż dowolnego łuku nie może przekroczyć jego przepustowości, a „łączny przepływ" wchodzący do dowolnego wierzchołka (z wyjątkiem źródła i ujścia) musi być równy „łącznemu przepływowi", który z niego wychodzi.

Wartością przepływu ϕ, oznaczaną jako $|\phi|$, nazywamy sumę przepływów wzdłuż łuków wychodzących ze źródła lub – równoważnie – sumę przepływów wzdłuż łuków wchodzących do ujścia.

Przepływ zerowy to taki, w którym przepływ wzdłuż każdego łuku wynosi zero. Zaprzeczeniem jego jest *przepływ niezerowy*. Natomiast, jeśli dla łuku $e \in E$ zachodzi równość $\phi(e) = \psi(e)$, to mówimy, że jest on *nasycony*; w przeciwnym wypadku — *nienasycony*. Przepływ ϕ *jest całkowitoliczbowy*, jeżeli dla każdego łuku $e \in E$ wartość $\phi(e)$ jest liczbą całkowitą.



Rysunek 71.31: Sieć przepływowa G. Oznaczenie przy każdym łuku $e \in E$ reprezentuje stosunek przepływu do przepustowości tego łuku, czyli $\phi(e)/\psi(e)$. W przypadku przepływu zerowego, podawana jest tylko przepustowość danej krawędzi

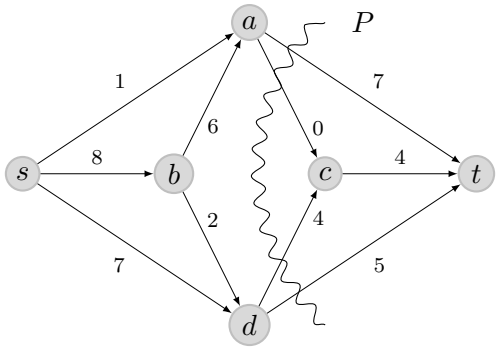
Przekrojem sieci przepływowej nazwiemy z kolei podzbiór P jej łuków taki, że każda droga ze źródła do ujścia musi prowadzić przez jeden z łuków należących do P. *Przepustowością przekroju* nazywamy sumę przepustowości łuków do niego należących: $\sum_{p \in P} \psi(p)$.

Jak wspomnieliśmy na początku, w sieciach przepływowych interesować nas będą głównie *przepływy maksy-*

malne, czyli o największej wartości lub – co równoważne – przekroje o jak najmniejszej przepustowości, tj. *przekroje minimalne*. Równoważność ta wynika z ważnego twierdzenia udowodnionego przez Forda i Fulkersona w 1955 roku:

Twierdzenie 71.5 (Ford-Fulkerson)

W każdej sieci wartość dowolnego przepływu maksymalnego jest równa przepustowości dowolnego przekroju minimalnego.



Rysunek 71.32: Przekrój P sieci przepływowej G z rysunku 71.31 (o przepustowości 16)

Zauważmy tu, że przypadek sieci G, w której mielibyśmy wiele źródeł $\{s_1, s_2, ..., s_m\}$ i wiele ujść $\{t_1, t_2, ..., t_n\}$ nie jest trudniejszy w kontekście problemu maksymalnego przepływu, gdyż taką sieć można sprowadzić do równoważnej jej sieci G' z pojedynczym źródłem i ujściem poprzez dodanie tzw. *superźródła* s oraz *superujścia* t. Dodajemy wówczas do sieci krawędzie skierowane (s, s_i) o przepustowości $c(s, s_i) = \infty$ dla każdego $i = 1, 2, ..., m$ oraz krawędzie skierowane (t_j, t) o przepustowości $c(t_j, t) = \infty$ dla każdego $j = 1, 2, ..., n$. Wówczas każdy przepływ sieci G odpowiada przepływowi w sieci G' i na odwrót.

71.14.1.1 Metoda Forda-Fulkersona

W przypadku sieci skomplikowanych i rozległych, skuteczne metody znajdowania przepływu maksymalnego oparte są na konstruowaniu *dróg powiększających przepływ* ze źródła do ujścia. Jeśli wiemy, że dany łuk sieci $e = (u, v) \in E$ jest nienasycony i znamy jego przepustowość $\psi(u, v)$ oraz bieżący przepływ $\phi(u, v)$, to łatwo zauważyć, że przepływ ten możemy jeszcze powiększyć, maksymalnie o wartość $\psi(u, v) - \phi(u, v)$; wtedy łuk ten stanie się nasycony. Oczywiście możemy także zmniejszyć ten przepływ, maksymalnie o $\phi(u, v)$, czyli do zera[6]. Niech obie te wartości staną się wagami dwóch nowych krawędzi: $e' = (u, v)$ oraz $e'' = (v, u)$, zastępujących oryginalną krawędź e, przy czym wagą krawędzi e' będzie $\psi(u, v) - \phi(u, v)$, a wagą przeciwnie skierowanej krawędzi e'' będzie $\phi(u, v)$. Dokonując takiej zamiany wszystkich krawędzi w sieci G otrzymamy nową sieć przepływową – tzw. *sieć residualną*, $G_\phi = (V, E_\phi)$, w której wagi łuków oznaczają o ile można zmienić przepływ w odpowiadających im łukach w sieci G. Wartości te nazywamy *przepustowością residualną*, zdefiniowaną formalnie jako:

$$\psi_\phi(u, v) = \begin{cases} \psi(u, v) - \phi(u, v) & \text{dla } (u, v) \in E \\ \phi(v, u) & \text{dla } (v, u) \in E \\ 0 & \text{dla } (u, v) \notin E \end{cases}$$

[6] Zmniejszanie przepływu w danym łuku może też być potrzebne do maksymalizacji przepływu w całej sieci, jeśli łuk ten prowadzi „wstecz", tzn. od ujścia w stronę źródła.

Każdy element zbioru *krawędzi residualnych* spełnia warunek dodatniego przepływu:

$$E_\phi = \{(u,v) \in V \times V : \psi_\phi(u,v) > 0\}\,.$$

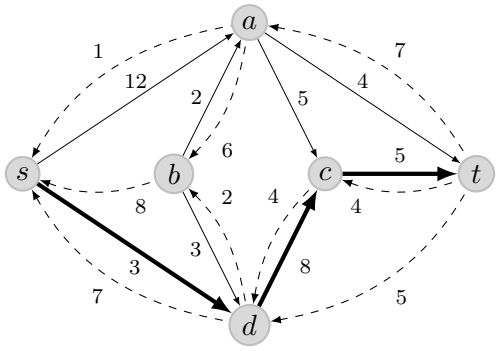
Przepływ ϕ' w sieci residualnej G_ϕ pokazuje nam w jaki sposób można powiększyć przepływ ϕ w pierwotnej sieci przepływowej G. Dla każdego łuku $(u,v) \in E$, wartość takiego *rozszerzenia* wynosi:

$$\phi(u,v) + \phi'(u,v) - \phi'(v,u)\,.$$

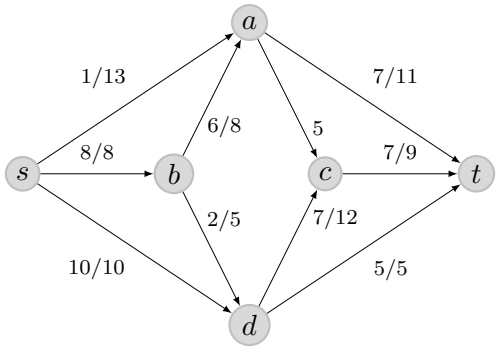
W przeciwnym wypadku (tj. gdy $(u,v) \notin E$), rozszerzenie to jest równe 0.

Każdą ścieżkę p ze źródła s do ujścia t w sieci residualnej G_ϕ nazywamy *ścieżką powiększającą* (dla sieci G i przepływu ϕ). *Przepustowością residualną ścieżki powiększającej* p będzie najmniejsza przepustowość residualna spośród tworzących ją krawędzi („wąskie gardło”):

$$\psi_\phi(p) = \min\{\psi_\phi(u,v) : (u,v) \in p\}\,.$$



Rysunek 71.33: Sieć residualna G_ϕ sieci przepływowej G (z rysunku 71.31) i ścieżka powiększająca p mająca przepustowość residualną $\psi_\phi(p) = 3$, o którą można powiększyć przepływ ϕ w sieci G wzdłuż ścieżki odpowiadającej p. Krawędzie o zerowej wartości przepływu zostały pominięte



Rysunek 71.34: Nowy przepływ w sieci z rysunku 71.31 po uwzględnieniu ścieżki powiększającej z rysunku 71.33

Możemy teraz sformułować metodę Forda-Fulkersona rozwiązywania problemu maksymalnego przepływu w sieci G, w której wierzchołek s jest źródłem, t ujściem, a ϕ przepływem. W każdej kolejnej iteracji zwiększamy wartość przepływu poprzez konstruowanie nowej sieci residualnej i wybieranie ścieżki powiększającej. Na niektórych krawędziach przepływ może się zmniejszać, ale powtarzanie tego procesu dopóki nie da się już znaleźć żadnej ścieżki powiększającej wygeneruje nam ostatecznie przepływ maksymalny. Można to uzasadnić, korzystając z podanego wcześniej twierdzenia o maksymalnym przepływie i minimalnym przekroju.

Ściślej rzecz biorąc, nie jest to gotowy algorytm, jednak stanowi podstawę dla kilku znanych implementacji o róż-

Algorithm 5 Metoda Forda-Fulkersona

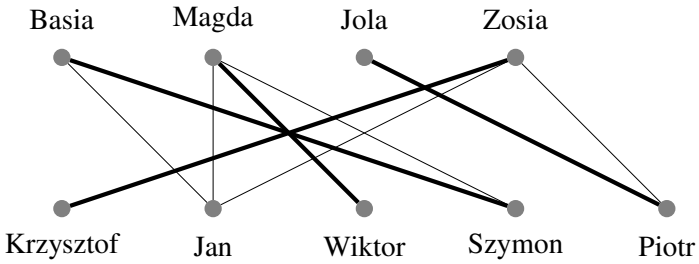
Require: Ford-Fulkerson-Method(G, s, t)
1: $\phi := 0$;
2: **while** istnieje ścieżka powiększająca p w sieci residualnej G_ϕ **do**
3: powiększ ϕ wzdłuż p;
4: **end while**
5: **return** ϕ

nym czasie działania, różniących się sposobem konstrukcji kolejnych ścieżek powiększających. Przykładowo, jeżeli do znalezienia ścieżki powiększającej zastosujemy przeszukiwanie grafu wszerz (BFS – p. sekcja 71.12.1), wybierając dzięki temu zawsze najkrótszą (w sensie liczby krawędzi) ścieżkę z s do t, to uzyskamy w ten sposób *algorytm Edmondsa-Karpa* [4][23] o złożoności obliczeniowej $O(|V||E|^2)$.

71.14.2 Skojarzenia

Załóżmy, że mamy do czynienia z taką oto sytuacją: każda dziewczyna ze skończonego zbioru dziewcząt zna pewną liczbę chłopców. Zadajmy sobie teraz pytanie: „Przy jakich warunkach każda z nich będzie mogła poślubić któregoś ze znanych sobie chłopców?" Problem ten nazywamy *problemem kojarzenia małżeństw* i można go przedstawić za pomocą grafu dwudzielnego $G = (V1 \cup V2, E)$, w którym zbiory $V1, V2$ odpowiadają dziewczętom i chłopcom, a każda krawędź $e \in E$ łączy pewną dziewczynę z pewnym chłopcem, którego zna. Możemy zauważyć, że aby problem ten miał rozwiązanie, to każde k dziewcząt – podzbiór zbioru wszystkich m dziewcząt – musi łącznie znać co najmniej k chłopców. Warunek ten nazywamy *warunkiem kojarzenia małżeństw* i okazuje się, że jest on – jak wykazał w 1933 roku Philip Hall – również warunkiem wystarczającym.

Skojarzeniem w grafie $G = (V, E)$ nazywamy każdy podzbiór $M \subseteq E$ krawędzi tego grafu, które nie posiadają wspólnych wierzchołków [4][17][23]. Innymi słowy, każdy wierzchołek w G jest końcem co najwyżej jednej krawędzi z M. Oczywiście w grafie może istnieć wiele skojarzeń – zauważmy, że dowolna pojedyncza krawędź jest już pewnym skojarzeniem. Nas jednak interesować będzie przede wszystkim tzw. *najliczniejsze (największe) skojarzenie* N w grafie G, czyli takie, które składa się z największej możliwej liczby krawędzi (takich najliczniejszych skojarzeń również może być w danym grafie wiele).

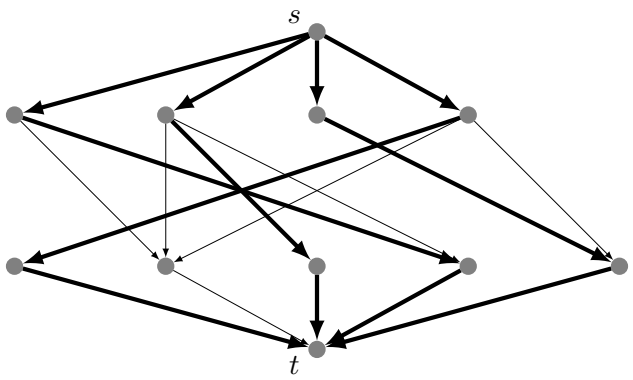


Rysunek 71.35: Najliczniejsze skojarzenie (pogrubione krawędzie) w grafie dwudzielnym G

Problem znajdowania najliczniejszego skojarzenia, w szczególności w grafach dwudzielnych, ma odzwierciedlenie w wielu zastosowaniach praktycznych, np. przy ustalaniu przydziału zadań dla pracowników w firmie, które będą oni wykonywać niezależnie od siebie.

Aby znaleźć najliczniejsze skojarzenie w grafie dwudzielnym, możemy posłużyć się sieciami przepływowymi. Rozwiązanie polega na skonstruowaniu [4] sieci przepływowej $G' = (V', E')$, w której przepływy odpowiadają

skojarzeniom w grafie G. W tym celu dodajemy do G dodatkowy węzeł s, stanowiący „superźródło” oraz dodatkowy węzeł t, stanowiący „superujście”, zgodnie z rysunkiem poniżej.



Rysunek 71.36: Sieć przepływowa G' dla grafu G z rysunku 71.35

Ustalamy również, że każda krawędź ma przepustowość wynoszącą dokładnie jeden. Najliczniejsze skojarzenie otrzymamy poprzez znalezienie całkowitoliczbowego, maksymalnego przepływu w G'. Wykorzystując do tego przedstawioną wyżej metodę Forda-Fulkersona, w której przepływ o takiej własności jest gwarantowany [4], otrzymamy algorytm o wielomianowej złożoności obliczeniowej $O(|V||E|)$.

72 (VIII) – Among the Stars – Gremliny *(zad. w jęz. angielskim)*

Autor: Jan Filipowicz, Politechnika Łódzka

Kategoria: Gremliny (IV edycja – zawody algorytmiczne – etap regionalny)

Język programowania: C++/Python

72.1 Task description

Your friend Jeremy hosts his own website where all registered users can share their own fantasy constellations (a constellation is any arrangement of stars – real or imaginary – connected with lines to make a shape). He built a small but proactive community focused around it and many people happily use it to show off their creations.

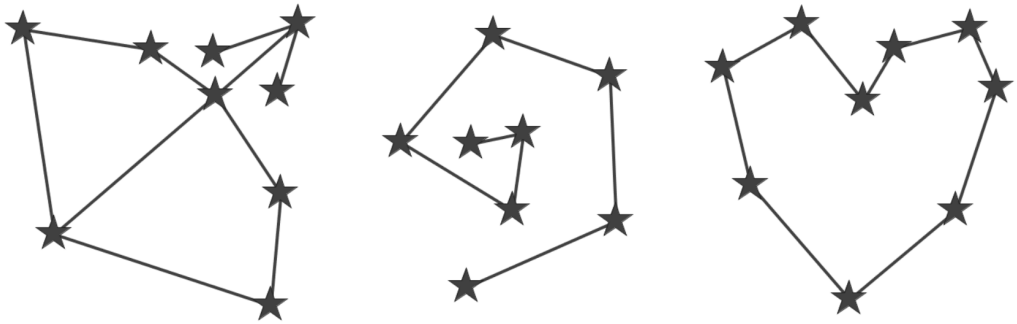


Figure 72.1: Some of the many fictional constellations designed by the website's users: three sets of stars connected by lines, shaped like a drawn bow with an arrow, a spiral, and a heart

As the website grew more popular, Jeremy noticed an unfortunate trend – some new users began to flood it with very similar, repetitive and uncreative constellations. For a reason he cannot understand, they always seem to outline the same crude shape of an astronaut.

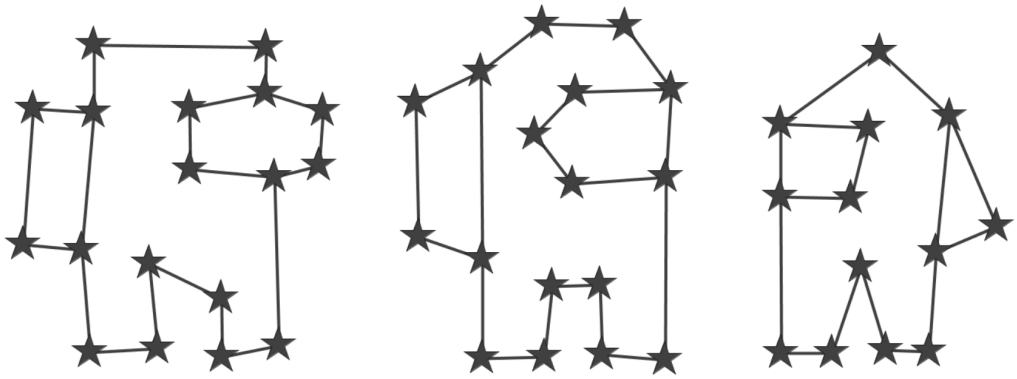


Figure 72.2: Some of the repetitive astronaut-shaped creations

Jeremy considers these creations to be spam and has been manually removing them from the site, but they only keep returning in greater numbers. He would like to automate detecting them, but they come from different accounts and IP addresses, and differ slightly in shape, so he is completely out of ideas.

As he continues venting to you about his struggles, you come up with a plan that might just help your friend overcome this tremendous problem.

Task

You notice that every spam constellation uploaded to the website has the same overall shape. If you ignore the positions of the stars but preserve the connections between them, it can be reduced to what looks like 3 different cycles, such that the first shares at least one line with the second, and the second shares at least one different line with the third. The first and the third cycle have no stars or lines in common.

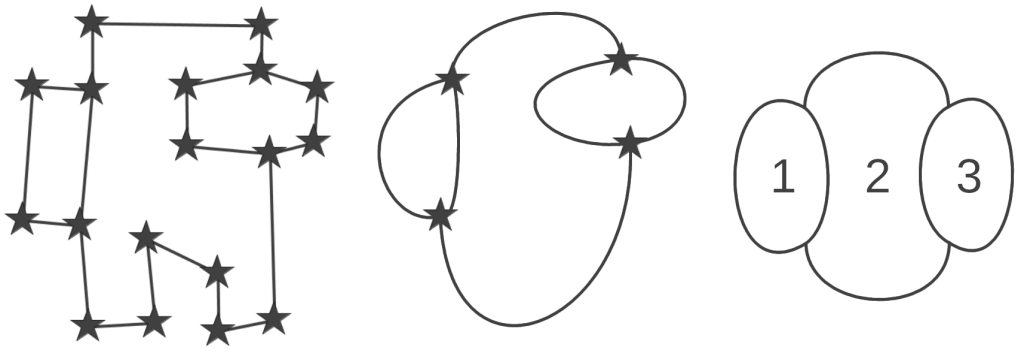


Figure 72.3: Simplification and generalization of the astronaut shape

Write a program that detects constellations exactly matching this structure.

Input description

The first line of standard input contains one natural number C ($1 \leq C \leq 100$) – the number of constellations to test. After that, C constellation descriptions follow.

Each constellation description begins with a line containing two integers S, L ($3 \leq S \leq L \leq 10^4$) – the number of stars and the number of lines between them. The stars are numbered from 1 to S. Each i-th of the next L lines contains two integers a_i, b_i ($1 \leq a_i < b_i \leq S$), meaning that the i-th line connects stars a_i and b_i. There will never be more than one line connecting the same pair of stars, and every star is guaranteed to be reachable from every other star through some series of lines.

Output description

For each of the C constellations your program should print one line – "`SUSPICIOUS`" if the constellation is detected as potential spam or "`NOT SUSPICIOUS`" if it does not match the structure of the undesired constellations.

Example

For sample input presented on the next page:

```
5
7 9
1 2
2 3
3 4
1 5
5 6
6 7
2 5
3 6
4 7
8 10
1 2
2 3
3 4
1 5
5 6
6 7
2 5
3 6
4 7
7 8
7 9
1 2
2 3
1 4
4 5
2 4
3 5
5 6
6 7
5 7
4 6
1 2
1 3
1 4
2 3
2 4
3 4
5 7
1 2
1 3
1 4
1 5
2 3
3 4
4 5
```

the correct output is:

```
SUSPICIOUS
NOT SUSPICIOUS
NOT SUSPICIOUS
NOT SUSPICIOUS
NOT SUSPICIOUS
```

Explanation

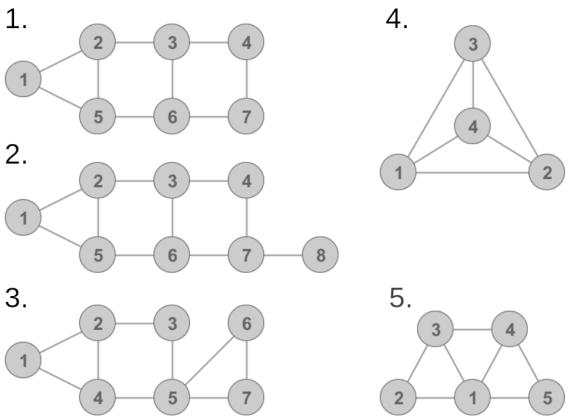


Figure 72.4: Example graphs

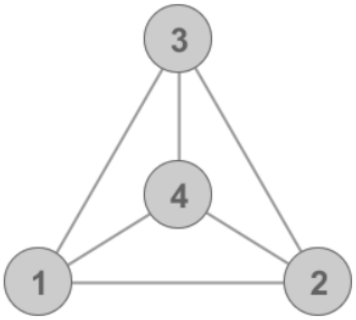
There are 5 constellations to test. The first one matches the structure we expect from spam, so we correctly report it as "`SUSPICIOUS`". The second one is almost identical, except that it has an extra star that does not belong to any of the 3 cycles we expect from spam, which is enough to declare it "`NOT SUSPICIOUS`". The third constellation has 3 cycles, but the second and the third one do not have any line in common. Sharing lines was one of our requirements, so we also declare it "`NOT SUSPICIOUS`". The fourth one does not remind our expected structure at all – it could be said to have 3 cycles too, but all of them share stars and lines – so it receives the same "`NOT SUSPICIOUS`" verdict. The final, fifth constellation also has 3 cycles but all of them have a star in common – star 1 – making the constellation "`NOT SUSPICIOUS`".

72.2 Rozwiązanie

Celem zadania było zaprojektowanie i implementacja algorytmu rozpoznającego szczególne grafy składające się z 3 cykli takich, że pierwszy ma krawędź wspólną z drugim a drugi z trzecim, podczas gdy pierwszy nie współdzieli z trzecim żadnych wierzchołków ani krawędzi.

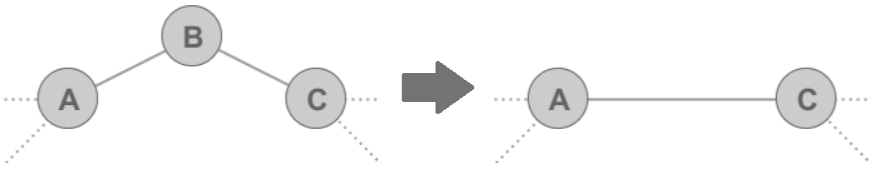
Przy omówieniu zadania przyda się pojęcie *stopnia* wierzchołka (p. definicja we wprowadzeniu do tego działu – sekcja 71.2). Stopniem nazywamy liczbę krawędzi wychodzących z wierzchołka. Łatwo zaobserwować, że graf może mieć pożądaną postać tylko wtedy, gdy wszystkie wierzchołki mają stopień równy 2 lub 3, przy czym dokładnie cztery wierzchołki muszą mieć stopień równy 3. Po wczytaniu grafu do *listy sąsiedztwa* (p. sekcja 71.11), tę własność bardzo łatwo sprawdzić i warto właśnie od tego zacząć algorytm.

Oczywiście nie każdy graf, który spełnia powyższy warunek, ma pożądaną postać – nawet w przykładach do zadania znalazł się graf tego typu przedstawiony poniżej.



Rysunek 72.5: Graf pełny o 4 wierzchołkach

Musimy zatem rozważać dalej. Zauważmy, że wierzchołki o stopniu 2 nie mają wpływu na odpowiedź, tj. jeżeli w zadanym grafie mamy pewne wierzchołki A, B, C połączone krawędziami od A do B i od B do C, oraz B ma stopień 2, to otrzymamy równoważny graf, jeżeli usuniemy B i połączymy A i C bezpośrednią krawędzią.



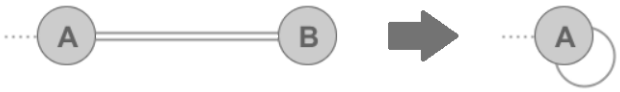
Rysunek 72.6: Operacja usunięcia wierzchołka z grafu

W ten sposób możemy odpowiednią pętlą uprościć graf do jedynie czterech wierzchołków (tych o stopniu 3), co powinno znacznie ułatwić jego analizę (por. pojęcie grafów homeomorficznych wprowadzone w sekcji 71.9). Podczas tej operacji możemy jednak natrafić na pewne niespodziewane sytuacje.

Mogłoby się np. tak zdarzyć, że wierzchołki A i C z powyższego przykładu byłyby dodatkowo połączone krawędzią bezpośrednią – wówczas w klasycznym ujęciu grafów nie można wstawić między nimi nowej krawędzi, a jest ona konieczna do zachowania kształtu grafu (po usunięciu wierzchołka B). Z pomocą przychodzi nam pojęcie *multigrafu* (p. sekcja 71.2), w którym między parą wierzchołków może znajdować się dowolnie wiele krawędzi. Szczęśliwie, postać listy sąsiedztwa grafu jest równie dobrze przystosowana do reprezentacji multigrafów, więc z punktu widzenia implementacji algorytmu nie musimy nawet dokonywać żadnych zmian.

Pojawia się tu jeszcze trzeci możliwy przypadek – jeżeli usunęliśmy już kilka wierzchołków i operujemy na

multigrafie, to możemy natrafić na wierzchołek B, który ma stopień 2, ale obie jego krawędzie kończą się w tym samym drugim wierzchołku, tj. A i C są tym samym wierzchołkiem. O dziwo, nawet wtedy nie musimy wprowadzać żadnych modyfikacji – uznajemy, że nowa krawędź jest wtedy *pętlą*, tj. prowadzi od A do A. Aby stopień wierzchołka A nie uległ zmianie, uznajemy, że pętla liczy się dla niego jako 2 krawędzie, ponieważ wierzchołek A jest na obu jej końcach. Implementacyjnie to również okazuje się zwykle wygodne i nie wymaga jawnego zapisu przypadków szczególnych, ale warto pisząc kod mieć na uwadze, że może się to zdarzyć.

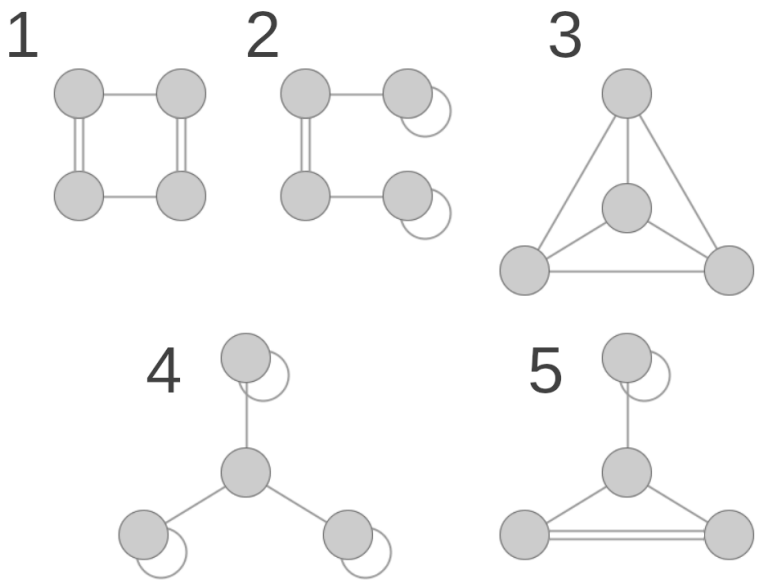


Rysunek 72.7: Usunięcie wierzchołka tworzące pętlę

Podczas implementacji może okazać się, że zamiast dosłownie usuwać wierzchołek z grafu, łatwiej jest usunąć prowadzące z niego i do niego krawędzie i ignorować go w przyszłości – wówczas nie trzeba przeprowadzać skomplikowanej aktualizacji indeksów. Warto też zwrócić uwagę, że w celu usunięcia krawędzi możemy bez zmartwień o wydajność dokonywać prostego liniowego przeszukiwania listy sąsiedztwa dla danego wierzchołka, ponieważ na wstępie zapewniliśmy, że jej długość – równa stopniowi wierzchołka – nigdy nie przekracza 3, a zatem jest nieistotnie mała.

Po sprowadzeniu multigrafu do czterech wierzchołków pozostaje tylko sprawdzić, czy otrzymaliśmy ten wzorcowy z nich. Szczęśliwie dla nas, istnieje tylko 5 różnych spójnych multigrafów, w których wszystkie wierzchołki mają stopień równy 3, o czym można stosunkowo szybko przekonać się rozpisując możliwe konfiguracje krawędzi na kartce.

Na poniższej ilustracji znajdują się wszystkie takie multigrafy, przy czym ten, który usiłujemy identyfikować, posiada numer 1. Nietrudno zaprojektować jakąś metodę, która taką identyfikację umożliwi – np. możemy zauważyć, że trzy z niepożądanych multigrafów posiadają pętlę (2, 4, 5) a pozostały nie posiada żadnych podwójnych krawędzi (3) i na tej podstawie je odrzucić.



Rysunek 72.8: Wszystkie możliwe do otrzymania multigrafy

72.2.1 Python

Źródło: ./zadania/08_Grafy/Among_the_Stars/solution.py.

```python
def rozwiaz_przypadek():
    s, l = [int(x) for x in input().split()]
    konstelacja = [[] for _ in range(s)]
    for _ in range(l):
        a, b = [int(x) for x in input().split()]
        a -= 1
        b -= 1
        konstelacja[a].append(b)
        konstelacja[b].append(a)
    rozgalezienia = 0
    for gwiazda in konstelacja:
        if len(gwiazda) == 3:
            rozgalezienia += 1
        elif len(gwiazda) != 2:
            return False
    if rozgalezienia != 4:
        return False
    for i, gwiazda in enumerate(konstelacja):
        if len(gwiazda) == 2:
            sasiad_1, sasiad_2 = gwiazda
            konstelacja[sasiad_1][konstelacja[sasiad_1].index(i)] = sasiad_2
            konstelacja[sasiad_2][konstelacja[sasiad_2].index(i)] = sasiad_1
            gwiazda.clear()
    for i, gwiazda in enumerate(konstelacja):
        if gwiazda:
            if i in gwiazda:
                return False
            if len(gwiazda) == len(set(gwiazda)):
                return False
    return True

c = int(input())
for _ in range(c):
    if rozwiaz_przypadek():
        print("SUSPICIOUS")
    else:
        print("NOT SUSPICIOUS")
```

72.2.2 C++

Źródło: ./zadania/08_Grafy/Among_the_Stars/solution.cpp.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
bool rozwiaz_przypadek() {
  int s, l;
  std::cin >> s >> l;
  std::vector<std::vector<int>> konstelacja(s);
  for (int i = 0; i < l; i++) {
    int a, b;
    std::cin >> a >> b;
    a--;
    b--;
    konstelacja[a].push_back(b);
    konstelacja[b].push_back(a);
  }
  int rozgalezienia = 0;
  for (const auto& gwiazda : konstelacja) {
    if (gwiazda.size() == 3)
      rozgalezienia++;
    else if (gwiazda.size() != 2)
      return false;
  }
  if (rozgalezienia != 4)
    return false;
  for (int i = 0; i < s; i++) {
    auto& gwiazda = konstelacja[i];
    if (gwiazda.size() == 2) {
      int sasiad_1 = gwiazda[0];
      int sasiad_2 = gwiazda[1];
      *std::find(konstelacja[sasiad_1].begin(),
        konstelacja[sasiad_1].end(), i) = sasiad_2;
      *std::find(konstelacja[sasiad_2].begin(),
        konstelacja[sasiad_2].end(), i) = sasiad_1;
      gwiazda.clear();
    }
  }
  for (int i = 0; i < s; i++) {
    const auto& gwiazda = konstelacja[i];
    if (!gwiazda.empty()) {
      if (std::find(gwiazda.begin(), gwiazda.end(), i) != gwiazda.end())
        return false;
      if (gwiazda[0] != gwiazda[1]
        && gwiazda[1] != gwiazda[2] && gwiazda[0] != gwiazda[2])
        return false;
    }
  }
  return true;
}
```

```
int main() {
  int c;
  std::cin >> c;
  for (int i = 0; i < c; i++) {
    if (rozwiaz_przypadek())
      std::cout << "SUSPICIOUS\n";
    else
      std::cout << "NOT SUSPICIOUS\n";
  }
  return 0;
}
```

73 (VIII) – Derby – Gremliny

Autor: Paweł Koszałka, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap regionalny)
Język programowania: C++/Python

73.1 Treść zadania

W Bit Jorku – jednym z głównych miast Bajtocji – istnieją dwie drużyny piłkarskie: FC Int oraz Mega Bajt. Kluby od zawsze rywalizują za sobą. Kibice FC Intu znani są z tego, że sukcesy swojej drużyny świętują w niezwykle huczny sposób, kiedy tylko spotykają się ze sobą, podobnie zresztą jak kibice Mega Bajtu. Fani przeciwnych drużyn natomiast unikają siebie nawzajem i nie utrzymują żadnych kontaktów. Fakt ten w sprytny sposób postanowili wykorzystać włodarze miasta, aby zapewnić spokój podczas nadchodzących derbów, które – jak co roku – organizowane są na (położonym w pewnej odległości od Bit Jorku) Bajtockim Stadionie Narodowym.

Kibice obu drużyn, przyjeżdżający na miejsce w przeddzień meczu, zostaną zakwaterowani w jednym z dostępnych ośrodków noclegowych. Ośrodków tych jest wiele, ale każdy wygląda dość podobnie i składa się z domków połączonych ze sobą alejkami lub pomostami, które nie tworzą skrzyżowań. W każdym z domków może zamieszkać tylko jeden kibic.

W celu zapewnienia spokoju, władze Bit Jorku chciałyby wybrać taki ośrodek, w którym domek kibica jednej drużyny sąsiadowałby wyłącznie z domkami kibiców tej drugiej, aby nie mogło dochodzić do bezpośrednich spotkań między Bitjorczykami kibicującymi tej samej drużynie.

Zadanie

Pomóż organizatorom imprezy i napisz program sprawdzający, które ośrodki spełniają powyższy warunek. Należy przy tym założyć, że niezależnie od liczby domków w danym ośrodku, wszystkie domki w ośrodku powinny zostać zasiedlone, a liczba kibiców (obu drużyn) jest nieograniczona.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się liczba z $\left(3 \leqslant z \leqslant 10^2\right)$ oznaczająca liczbę ośrodków do sprawdzenia. W pierwszym wierszu opisu każdego ośrodka znajdują się dwie liczby: n $\left(2 \leqslant n \leqslant 10^3\right)$ oraz s $\left(1 \leqslant s \leqslant 10^6\right)$ oddzielone pojedynczym odstępem (spacją). Wartość n, to liczba domków w ośrodku (ponumerowanych od 1 do n), zaś s oznacza liczbę alejek lub pomostów łączących domki. Kolejne s wierszy opisu ośrodka zawiera po jednej parze liczb naturalnych, oddzielonych pojedynczym odstępem, oznaczających numery połączonych domków (przy czym żaden domek nie jest połączony alejką bezpośrednio z samym sobą). Możesz założyć, że łączna suma wartości s we wszystkich ośrodkach nie przekracza $5 \cdot 10^6$.

Opis wyjścia

W każdym z z wierszy wyjścia powinno znaleźć się słowo TAK, jeśli dany ośrodek spełnia warunek postawiony przez władze Bit Jorku lub NIE – w przeciwnym wypadku.

Przykład

Wejście:

```
3
4 4
1 2
2 3
3 4
4 1
4 5
1 2
1 3
2 3
3 4
4 1
4 5
2 1
1 2
3 2
4 3
4 3
```

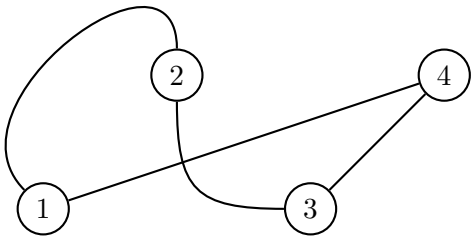
Wyjście:

```
TAK
NIE
TAK
```

Wyjaśnienie przykładu

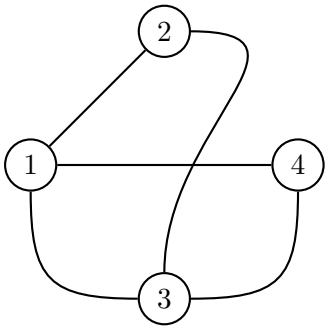
Ogólnie rzecz biorąc, topologię każdego ośrodka możemy zilustrować za pomocą grafu nieskierowanego, którego węzłami są numery domków, zaś krawędzie oznaczają alejki lub pomosty między nimi (p. rysunki na następnej stronie). Ścieżki łączące domki nigdy nie krzyżują się ze sobą – należy założyć, że w miejscu gdzie mogłyby się przeciąć znajduje się zawsze pomost.

W naszym przykładzie mamy trzy ośrodki. Układ domków i alejek w pierwszym z nich możemy przedstawić następująco:



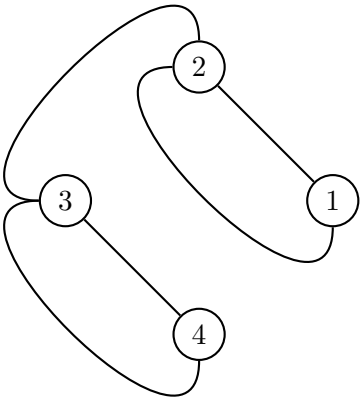
W tym przypadku można na przykład rozmieścić kibiców FC Int w domkach 1 i 3, a kibiców Mega Bajt w domkach 2 i 4. Wówczas fani tej samej drużyny nie będą sąsiadować ze sobą.

Plan drugiego ośrodka opisuje poniższy rysunek:



Przy takim układzie domków, nie da się rozdzielić Bitjorczyków w oczekiwany sposób. Niezależnie od tego, której drużynie kibicowałby mieszkaniec domku nr 1, jego sąsiedzi musieliby kibicować tej drugiej, ale sami są również swoimi bezpośrednimi sąsiadami, co przeczy warunkom zadania.

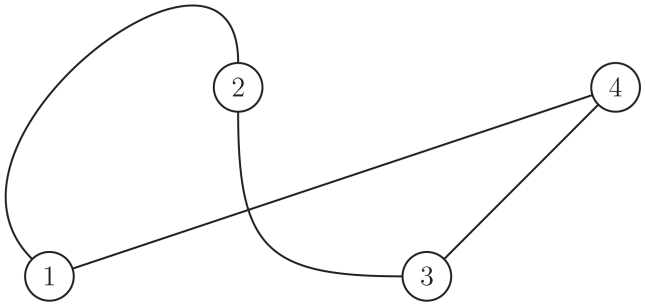
Trzeci przypadek pozwala osiągnąć zamierzony efekt. Kibice jednej z drużyn mogliby zostać ulokowani w domkach 1 i 3, a drugiej – w domkach 2 i 4:



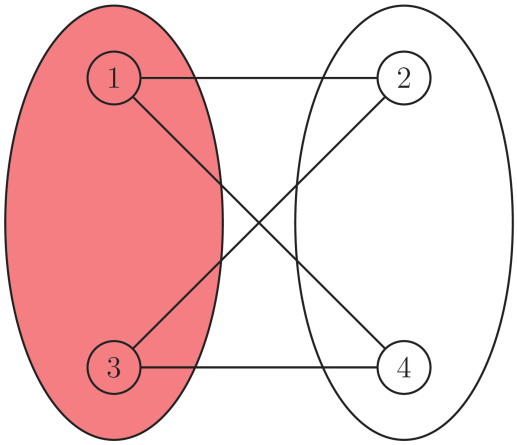
73.2 Rozwiązanie

Opisany w zadaniu problem dotyczy zagadnienia grafów dwudzielnych (p. sekcja 71.8), to znaczy takich, w których wierzchołki da się podzielić na dwa rozłączne zbiory względem ich krawędzi. Chodzi o to, aby wewnątrz każdego z tych zbiorów wierzchołki nie miały połączeń między sobą. Mogą natomiast posiadać (choć nie muszą) krawędzie z wierzchołkami z drugiego zbioru. W naszym zadaniu jednym zbiorem będą kibice drużyny FC Int, zaś drugim – fani Mega Bajtu (ponieważ zgodnie z założeniami zadania, kibice tej samej drużyny nie mogą sąsiadować ze sobą w ośrodku).

Przykład pierwszy w zadaniu wygląda następująco:



Możemy jednak popatrzeć na niego w poniższy sposób. Wtedy widać, że wierzchołki 1 i 3 tworzą jeden zbiór, a wierzchołki 2 i 4 – drugi. Między nimi istnieją krawędzie, ale wewnątrz zbiorów elementy nie mają połączeń, czyli nie sąsiadują ze sobą. Graf w tym przypadku jest zatem dwudzielny. Przekładając to na realia naszego zadania, można zatem stwierdzić, że da się tu rozdzielić kibiców FC Int i Mega Bajtu tak, aby żadnych dwóch kibiców tej samej drużyny nie sąsiadowało ze sobą.



Aby sprawdzić czy dany graf jest dwudzielny wystarczy „przejść po nim” algorytmem BFS lub DFS (p. sekcja 71.12) i kolorować jego wierzchołki. Jako informację o kolorze można tu wykorzystać tablicę „odwiedzonych” oraz wartości: $-1, 0, 1$. Na początku wszystkie wierzchołki otrzymują kolor 0, który oznacza również, że nie były odwiedzone. Rozpoczynamy przechodzenie grafu z dowolnego wierzchołka i kolorujemy go wartością 1 oznaczając tym samym, że jest odwiedzony. Następnie wrzucamy go na koniec kolejki w przypadku algorytmu BFS lub na stos, jeśli wykorzystujemy DFS. W głównej pętli algorytmu pobieramy wierzchołek z początku kolejki (lub z wierzchu stosu), przeglądamy wszystkich jego nieodwiedzonych (pokolorowanych na 0) sąsiadów i kolorujemy ich na kolor przeciwny. To znaczy, że do tablicy „odwiedzonych” sąsiadów musimy wstawić

wartość koloru wierzchołka, w którym aktualnie jesteśmy pomnożoną przez -1. Następnie taki wierzchołek trafia na koniec kolejki (lub na stos). Tę procedurę powtarzamy, aż do wyczerpania kolejki (lub stosu). Kiedy tak się stanie, możemy stwierdzić, że graf jest dwudzielny. Zawsze, gdy któryś z sąsiadów był już odwiedzony (kolor ma wartość inną niż 0), to porównujemy jego kolor z kolorem wierzchołka, w którym aktualnie jesteśmy. Jeśli są takie same, to graf nie jest dwudzielny i nie da się go pokolorować tak, aby żadne dwa wierzchołki o tym samym kolorze nie były połączone krawędzią. W tym momencie przerywamy kolorowanie grafu stwierdzając, że nie jest dwudzielny.

73.2.1 Python

Źródło: ./zadania/08_Grafy/Derby/solution.py.

```
import sys

from collections import defaultdict
from queue import Queue

#alorytm BFS

def BFS(start,graph,color):
    global tak
    tak=True
    Q = Queue()
    color[start]=1
    Q.put(start)

    while not(Q.empty()):
        v=Q.get()

        for i in range(len(graph[v])):
            if not(color[graph[v][i]]):
                color[graph[v][i]]=(-1)*color[v]
                Q.put(graph[v][i])
            elif (color[graph[v][i]] == color[v]):
                print("NIE")
                tak=False
                return

#wczytanie grafu jako lista sąsiedztwa

z=int(input())
for j in range(1,z+1):

    n, m = input().split()
    n = int(n)
    m = int(m)

    graph = defaultdict(list)
    color = [0]*(n+1)
```

```
    for i in range(m):
        a, b = input().split()
        a = int(a)
        b = int(b)

        graph[a].append(b)
        graph[b].append(a)

    #głóuma część

    for w in range(1,n+1):
        if not(color[w]):
            BFS(w,graph,color)
        if not(tak):
            break

    if tak:
        print("TAK")
```

73.2.2 C++

Źródło: ./zadania/08_Grafy/Derby/solution.cpp.

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

bool tak;

void BFS(int n,vector<int> G[], int kolor[]){
    tak=true;
  queue<int> Q;
  kolor[n]=1;
  Q.push(n);
  while(!Q.empty()){
  int k=Q.front();Q.pop();
  for(int i=0;i<(int)G[k].size();i++)
    if(kolor[G[k][i]]==0){
      kolor[G[k][i]]=(-1)*kolor[k];
      Q.push(G[k][i]);
    }else if(kolor[G[k][i]]==kolor[k]){
      cout<<"NIE"<<endl;
      tak=false;
      return;
    }
  }
```

```
}

int main()
{
  int z;
  cin>>z;
  for(int j=1;j<=z;j++)
    {
        vector<int> G[10001];
        int N,V;
        int kolor[10001]={0};

        cin>>N>>V;
        int k,l;
        for(int i=1;i<=V;i++){
            cin>>k>>l;
            G[k].push_back(l);
            G[l].push_back(k);
        }
        for(int n=1;n<=N;n++)
        {
            if(kolor[n]==0)BFS(n,G,kolor);
            if(!tak) break;
        }

        if(tak) cout<<"TAK"<<endl;
    }

  return 0;
}
```

74 (VIII) – Transmutacja obiektów nieożywionych – Gnomy

Autor: Łukasz Skonieczny, Politechnika Warszawska
Kategoria: Gnomy (IV edycja – zawody algorytmiczne – etap regionalny)
Język programowania: C++/Python

74.1 Treść zadania

W szkole Czarów, Magii i Inkantacji (CMI) naucza się między innymi zaklęć pozwalających zmieniać lub inaczej: *transmutować* obiekty nieożywione w inne obiekty nieożywione (istnieją też zaklęcia do zamiany obiektów ożywionych, ale należą one do magii dużo wyższego poziomu, którą nie będziemy się teraz zajmować). Zaklęcie transmutacyjne działa symetrycznie, czyli jeśli „abrakadabra" zamienia widelec w szklankę, wtedy to samo zaklęcie „abrakadabra" rzucone na szklankę zamienia ją w widelec. Znane zaklęcia spisane są w Wielkiej Księdze Transmutacji Zwyczajnej. Wiadomo, że w księdze nie ma powtórzeń, czyli dla danej pary obiektów istnieje co najwyżej jedno zaklęcie pozwalające zmieniać te obiekty między sobą. Księga jest kompletna w tym sensie, że jeśli tylko jakiś obiekt A da się zamienić w obiekt Z, to w księdze można znaleźć zaklęcie zmieniające bezpośrednio A w Z lub serię zaklęć $A \leftrightarrow B$, $B \leftrightarrow C$, ..., $X \leftrightarrow Z$ zmieniającą A w Z w kilku krokach. I rzeczywiście, w wielu przypadkach księga nie zawiera bezpośredniego zaklęcia do zamiany A w Z, ale możliwe jest znalezienie w niej serii zaklęć (czasem bardzo długiej), której wykonanie do takiej zamiany ostatecznie doprowadzi.

Wielcy magowie uznali, że z tego powodu księga jest niewygodna w użyciu, więc postanowili uzupełnić ją o wszystkie brakujące zaklęcia.

Zadanie

Znajdź liczbę brakujących zaklęć w Wielkiej Księdze Transmutacji Zwyczajnej. Brakującym zaklęciem jest takie zaklęcie, które nie zostało zapisane w księdze, ale jego efekt można uzyskać za pomocą serii innych zaklęć.

Opis wejścia

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite: n oraz m (gdzie $1 \leq n \leq 100\,000$, $1 \leq m \leq 3\,200\,000$), oznaczające odpowiednio liczbę obiektów nieżywionych oraz liczbę zaklęć. W kolejnych m wierszach znajdują się pary liczb z przedziału $[1, n]$. Każda taka para oznacza zaklęcie transmutujące obiekty z tej pary między sobą.

Opis wyjścia

Na standardowe wyjście należy wypisać jedną liczbę całkowitą oznaczającą liczbę brakujących zaklęć.

Przykłady

Dla przykładowego, podanego poniżej wejścia:

```
4 3
1 2
2 3
2 4
```

prawidłową odpowiedzią jest:

```
3
```

Z kolei dla innego wejścia:

```
6 4
1 2
2 3
5 4
6 4
```

prawidłową odpowiedzią jest:

```
2
```

Wyjaśnienie przykładów

W pierwszym przypadku księga zawiera 3 zaklęcia dotyczące czterech obiektów. Są to: $1 \leftrightarrow 2$, $2 \leftrightarrow 3$, $2 \leftrightarrow 4$.

Można zauważyć, że każdy obiekt może być tu zamieniony na każdy inny, ale niekoniecznie bezpośrednio. Przykładowo, obiekt 1 może zostać zamieniony na obiekt 3 za pomocą dwóch zaklęć: $1 \leftrightarrow 2$, $2 \leftrightarrow 3$. Brakuje zatem zaklęcia $1 \leftrightarrow 3$. Podobnie łatwo zauważyć, że brakuje też zaklęć $1 \leftrightarrow 4$ oraz $3 \leftrightarrow 4$. Całkowita liczba brakujących zaklęć wynosi zatem trzy.

W drugim przypadku księga zawiera 4 zaklęcia dotyczące sześciu obiektów: $1 \leftrightarrow 2$, $2 \leftrightarrow 3$, $4 \leftrightarrow 5$, $4 \leftrightarrow 6$. Brakuje w niej tylko dwóch zaklęć: $1 \leftrightarrow 3$, $5 \leftrightarrow 6$.

Zauważmy, że faktycznie więcej zaklęć dodać tu już nie można. Przykładowo, zaklęcie $1 \leftrightarrow 4$ nie znajduje się w księdze, ale nie może zostać do niej dodane, gdyż nie istnieje żadna seria zaklęć pozwalająca zmienić obiekt 1 w obiekt 4.

74.2 Rozwiązanie

To zadanie ma drugą, trudniejszą wersję („Transmutacja obiektów ożywionych” dla Gremlinów), przedstawioną dalej wraz z opisem rozwiązania obu wersji. Tu prezentujemy tylko sam kod wzorcowy rozwiązania dla Gnomów.

74.2.1 Python

Źródło: ./zadania/08_Grafy/Transmutacja_Gnomy/solution-undirected.py.

```
from collections import defaultdict
from queue import Queue

def main():
    ns,ms = input().strip().split()
    n,m = int(ns), int(ms)
    graph = defaultdict(list)

    for i in range(0,m):
        a,b = input().strip().split()
        graph[int(a)-1].append(int(b)-1)
        graph[int(b)-1].append(int(a)-1)

    visited = set()
    components = defaultdict(list)
    current_component = 0
    for vertice in graph:
        if vertice not in visited:
            visited.add(vertice)
            queue = Queue()
            queue.put(vertice)

            while not queue.empty():
                v = queue.get()
                components[current_component].append(v)
                for neighbour in graph[v]:
                    if neighbour not in visited:
                        visited.add(neighbour)
                        queue.put(neighbour)

            current_component += 1

    r = 0
    for c in components:
        for v in components[c]:
            r += len(components[c]) - len(graph[v]) - 1

    print(r//2) # print(current_component)
if __name__ == '__main__':
    main()
```

74.2.2 C++

Źródło: ./zadania/08_Grafy/Transmutacja_Gnomy/solution-undirected.cpp.

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <queue>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >>n>>m;
    vector<vector<int>> graph(n);
    for (int i=0; i<m; i++) {
        int a,b;
        cin >>a >>b;
        graph[a-1].push_back(b-1);
        graph[b-1].push_back(a-1);
    }
    set<int> visited;
    map<int, vector<int>> components;
    int current_component = 0;
    for (int vertice=0; vertice<n; vertice++) {
        if (visited.count(vertice)==0) {
            visited.insert(vertice);
            queue<int> queue;
            queue.push(vertice);
            while (!queue.empty()) {
                int v = queue.front();
                queue.pop();
                components[current_component].push_back(v);
                for (int neighbour : graph[v]) {
                    if (visited.count(neighbour)==0) {
                        visited.insert(neighbour);
                        queue.push(neighbour);
                    }
                }
            }
        }
        current_component++;
    }
    long r = 0;
    for (auto c : components)
         for (int v : c.second)
             r += c.second.size() - graph[v].size() - 1;
    cout <<r/2<<endl;
}
```

75 (VIII) – Transmutacja obiektów ożywionych – Gremliny

Autor: ***Łukasz Skonieczny***, ***Politechnika Warszawska***
Kategoria: ***Gremliny (IV edycja – zawody algorytmiczne – etap regionalny)***
Język programowania: ***C++/Python***

75.1 Treść zadania

W szkole Czarów, Magii i Inkantacji (CMI) naucza się między innymi zaklęć pozwalających zmieniać lub inaczej: *transmutować* obiekty ożywione w inne obiekty ożywione (istnieją też zaklęcia do zamiany obiektów nieożywionych, ale należą one do magii niższego poziomu, którą nie będziemy się teraz zajmować). Zaklęcie transmutacyjne dla obiektów ożywionych działa tylko w jedną stronę, czyli jeśli „abrakadabra" zamienia kota w wiewiórkę, to nie można go użyć do zamiany wiewiórki w kota. Do tego jest potrzebne inne zaklęcie, o ile w ogóle takie istnieje. Znane zaklęcia spisane są w Wielkiej Księdze Transmutacji Nadzwyczajnej. Wiadomo, że w księdze nie ma powtórzeń, czyli dla danej pary obiektów (A, B) istnieje co najwyżej jedno zaklęcie pozwalające zmienić obiekt A w obiekt B. Księga jest kompletna w tym sensie, że jeśli tylko jakiś obiekt A da się zamienić w obiekt Z, to w księdze można znaleźć zaklęcie zmieniające bezpośrednio A w Z lub serię zaklęć $A \to B$, $B \to C$, ..., $X \to Z$ zmieniającą A w Z w kilku krokach. I rzeczywiście, w wielu przypadkach księga nie zawiera bezpośredniego zaklęcia do zamiany A w Z, ale możliwe jest znalezienie w niej serii zaklęć (czasem bardzo długiej), której wykonanie do takiej zamiany ostatecznie doprowadzi.

Wielcy magowie uznali, że z tego powodu księga jest niewygodna w użyciu, więc postanowili uzupełnić ją o wszystkie brakujące zaklęcia.

Zadanie

Znajdź liczbę brakujących zaklęć w Wielkiej Księdze Transmutacji Nadzwyczajnej. Brakującym zaklęciem jest takie zaklęcie, które nie zostało zapisane w księdze, ale jego efekt można uzyskać za pomocą serii innych zaklęć.

Opis wejścia

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite: n oraz m ($1 \leq n \leq 1\,000$, $1 \leq m \leq 65\,000$), oznaczające odpowiednio liczbę obiektów ożywionych oraz liczbę zaklęć. W kolejnych m wierszach znajdują się pary liczb z przedziału $[1, n]$. Każda taka para oznacza zaklęcie transmutujące pierwszy obiekt z pary w drugi obiekt z pary.

Opis wyjścia

Na standardowe wyjście należy wypisać jedną liczbę całkowitą oznaczającą liczbę brakujących zaklęć.

Przykłady

Dla przykładowego, podanego poniżej wejścia:

```
4 3
1 2
2 3
2 4
```

prawidłową odpowiedzią jest:

```
2
```

Z kolei dla innego wejścia:

```
6 5
1 2
2 3
3 4
4 1
5 6
```

prawidłową odpowiedzią jest:

```
8
```

Wyjaśnienie przykładów

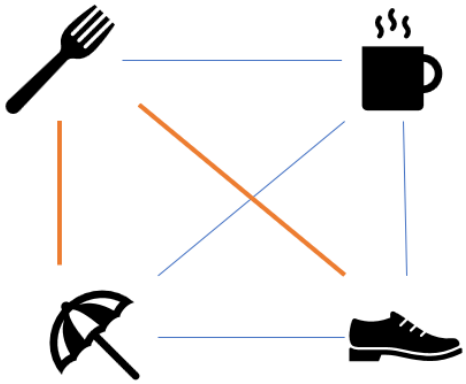
W pierwszym przypadku księga zawiera 3 zaklęcia dotyczące czterech obiektów: $1 \to 2$, $2 \to 3$, $2 \to 4$. Brakuje w niej dwóch zaklęć: $1 \to 3, 1 \to 4$. Pozostałe kandydujące zaklęcia $3 \to 4$, $4 \to 3$, $2 \to 1$, $3 \to 2$, $4 \to 2$, $3 \to 1$, $4 \to 1$ nie mogą zostać dodane do księgi, gdyż nie wynikają z żadnej serii transmutacji zapisanych w księdze.

W drugim przypadku księga zawiera 5 zaklęć dotyczących sześciu obiektów: $1 \to 2$, $2 \to 3$, $3 \to 4$, $4 \to 1$, $5 \to 6$. Brakuje w niej ośmiu zaklęć: $1 \to 3, 1 \to 4, 2 \to 4, 2 \to 1, 3 \to 1, 3 \to 2, 4 \to 2, 4 \to 3$. Zauważmy tu, że w niektórych przypadkach zostały dodane zaklęcia symetryczne, np. $2 \to 1$, gdyż obiekt 2 można zamienić w obiekt 1 za pomocą serii zaklęć $2 \to 3 \to 4 \to 1$.

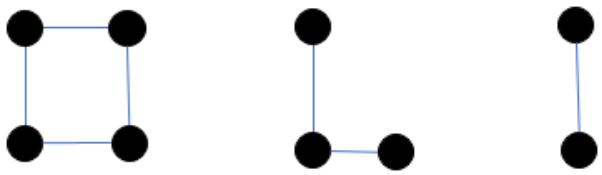
75.2 Rozwiązanie

W obu wersjach zadania, zaklęcia do przemiany obiektów tworzą graf, przy czym w przypadku transmutacji obiektów nieożywionych jest to graf nieskierowany, a w przypadku transmutacji obiektów ożywionych – graf skierowany. Stanowi to istotną różnicę, gdyż pojęcie osiągalności między dwoma wierzchołkami w grafie nieskierowanym jest symetryczne (jeśli istnieje ścieżka z A do B, to istnieje także ścieżka z B do A), a w przypadku grafów skierowanych ta własność niekoniecznie jest spełniona.

W obu wersjach zadania należy znaleźć liczbę brakujących krawędzi w grafie. Przez brakujące krawędzie rozumiemy krawędzie łączące wierzchołki osiągalne, które nie są połączone bezpośrednią krawędzią. Łatwo zauważyć, że w przypadku grafów nieskierowanych spójnych wszystkie wierzchołki są wzajemnie osiągalnie zatem rozwiązaniem jest po prostu różnica między liczbą krawędzi w grafie pełnym (jest ich $n(n-1)/2$), a liczbą krawędzi w grafie zaklęć. Jeśli graf nie jest spójny, to wierzchołki z różnych składowych spójnych nie są wzajemnie osiągalnie zatem powyższą różnicę należy policzyć niezależnie dla każdej składowej spójnej grafu, a następnie zsumować. Zadanie sprowadza się więc do wyznaczenia składowych spójnych, co można łatwo zrealizować algorytmem BFS lub DFS (p. sekcja 71.12 we wprowadzeniu do tego działu).

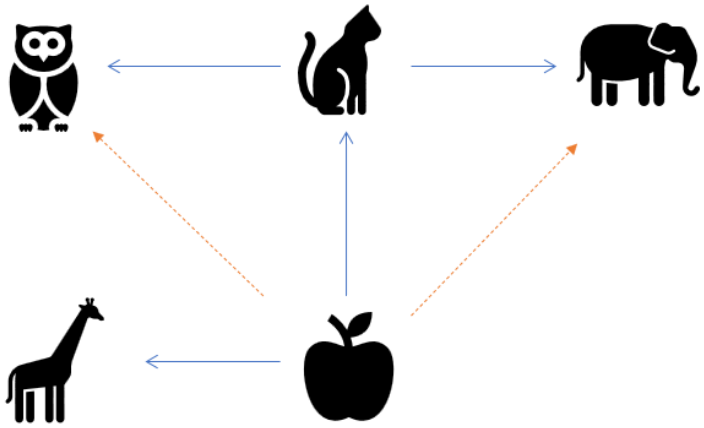


Rysunek 75.1: Graf reprezentuje cztery zaklęcia transmutacji obiektów nieożywionych – krawędzie niebieskie. Graf jest spójny, czyli każdy wierzchołek jest osiągalny z każdego innego. Do pełności grafu brakuje dwóch zaklęć – krawędzi pomarańczowych (grafika: www.vecteezy.com)

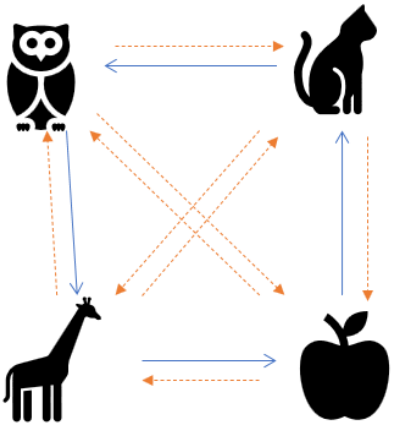


Rysunek 75.2: Graf transmutacji jest niespójny. W pierwszej składowej spójnej brakuje dwóch krawędzi, w drugiej składowej – jednej, a trzecia składowa jest grafem pełnym o dwóch wierzchołkach, więc nie brakuje w niej żadnej krawędzi. W całym grafie brakuje więc trzech zaklęć

W przypadku grafu skierowanego brakujące krawędzie wynikają z relacji przechodniości. Jeśli $a \to b$ oraz $b \to c$, to $a \to c$. Problemem do rozwiązania jest tu tak zwane *domknięcie przechodnie grafu*, czyli uzupełnienie grafu o krawędzie, których istnienie można wywnioskować z relacji przechodniości.



Rysunek 75.3: Domknięcie przechodnie grafu skierowanego, który nie zawiera cyklu (grafika: www.vecteezy.com)



Rysunek 75.4: Domknięcie przechodnie grafu skierowanego zawierającego cykl (grafika: www.vecteezy.com)

Istnieje kilka algorytmów do znajdowania domknięcia przechodniego grafu, w tym zaawansowane polegające na potęgowaniu macierzy sąsiedztwa lub szukaniu najkrótszych ścieżek pomiędzy wszystkimi możliwymi parami wierzchołków (algorytm Floyda-Warshalla [4]). W rozwiązaniu wzorcowym użyto prostszego algorytmu DFS odnajdującego wierzchołki osiągalne dla każdego wierzchołka grafu z osobna.

75.2.1 Python

Źródło: ./zadania/08_Grafy/Transmutacja_Gremliny/solution-directed.py.

```
from collections import deque

n, m = [int(x) for x in input().split()]
graf = []
for _ in range(n):
    graf.append([])
for _ in range(m):
    a, b = [int(x) for x in input().split()]
    graf[a - 1].append(b - 1)
wynik = -m
for i in range(n):
    stos = deque()
    widziano = n * [False]
    stos.append(i)
    widziano[i] = True
    while stos:
        v = stos.pop()
        for w in graf[v]:
            if not widziano[w]:
                widziano[w] = True
                stos.append(w)
                wynik += 1
print(wynik)
```

75.2.2 C++

Źródło: ./zadania/08_Grafy/Transmutacja_Gremliny/solution-directed.cpp.

```cpp
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n, m;
  std::cin >> n >> m;
  std::vector<std::vector<int>> graf(n);
  for (int i = 0; i < m; i++) {
    int a, b;
    std::cin >> a >> b;
    graf[a - 1].push_back(b - 1);
  }
  long long wynik = -m;
  for (int i = 0; i < n; i++) {
    std::vector<int> stos;
    std::vector<bool> widziano(n);
    stos.push_back(i);
    widziano[i] = true;
    while (!stos.empty()) {
      int v = stos.back();
      stos.pop_back();
      for (const auto& w : graf[v]) {
        if (!widziano[w]) {
          widziano[w] = true;
          stos.push_back(w);
          wynik++;
        }
      }
    }
  }
  std::cout << wynik << '\n';
  return 0;
}
```

76 (VIII) – Bit Chata – Gremliny

Autor: Paweł Koszałka, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gremliny (III edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

76.1 Treść zadania

W Górach Bajtockich znajduje się tylko jedno schronisko turystyczne, o nazwie „Bit Chata". Niestety sieć szlaków trekkingowych nie pozwala do niego dotrzeć z każdego miejsca w kompleksie turystycznym. Punkty węzłowe szlaków ponumerowane są od 1 do n, z czego 1 należy do Bit Chaty.

Ze względu na wymagający teren w górach, istnieje tylko m jednokierunkowych przejść pomiędzy węzłami, a czasy potrzebne na ich pokonanie są znane. Nie wszystkie punkty węzłowe są ze sobą połączone szlakami, zaś pomiędzy niektórymi dwoma z nich można przejść na więcej niż jeden sposób.

Służby Ratownictwa Górskiego Bajtocji (SRGB) testują nowy system ostrzegania turystów. Na przykład, w przypadku zbliżającego się załamania pogody, każdy kto znajduje się na szlaku otrzyma powiadomienie SMS z informacją o zagrożeniu oraz minimalnym czasie potrzebnym, aby z najbliższego punktu węzłowego dojść do schroniska lub o braku możliwości dotarcia do Bit Chaty.

Zadanie

Pomóż SRGB i napisz program, który wyznaczy dane potrzebne do komunikatu.

Opis wejścia

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby: n $(2 \leqslant n \leqslant 10^5)$ oraz m $(1 \leqslant m \leqslant 10^6)$.

W i-tym z kolejnych m wierszy znajduje się opis i-tego przejścia między punktami węzłowymi. Opis ten składa się z trzech liczb całkowitych oddzielonych spacją. Są to: a_i, b_i — numery punktów węzłowych $(1 \leqslant a_i, b_i \leqslant n)$ oraz c_i — czas i-tego przejścia z punktu a do punktu b wyrażony w bajtockich jednostkach czasu $(1 \leqslant c_i \leqslant 10^9)$.

Opis wyjścia

W i-tym z $n-1$ wierszy wyjścia powinna znaleźć się jedna liczba: -1, jeśli z węzła $i+1$ nie da się dojść do Bit Chaty lub minimalny czas przejścia, jeśli taka możliwość istnieje.

Przykłady

Przykład 1

Wejście:

```
4 4
2 1 100
4 1 20
2 4 10
3 2 50
```

Wyjście:

```
30
80
20
```

Przykład 2

Wejście:

```
5 3
2 1 10
3 2 30
4 5 70
```

Wyjście:

```
10
40
-1
-1
```

Przykład 3

Wejście:

```
4 5
2 1 20
1 2 10
3 2 50
4 3 10
4 3 20
```

Wyjście:

```
20
70
80
```

Przykład 4

Wejście:

```
10 10
1 1 43
2 6 92
3 5 54
4 4 22
5 3 96
1 2 72
2 1 13
3 3 36
4 2 12
5 1 74
```

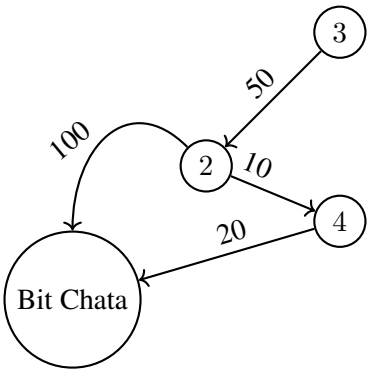
Wyjście:

```
13
128
25
74
-1
-1
-1
-1
-1
```

Wyjaśnienie przykładów

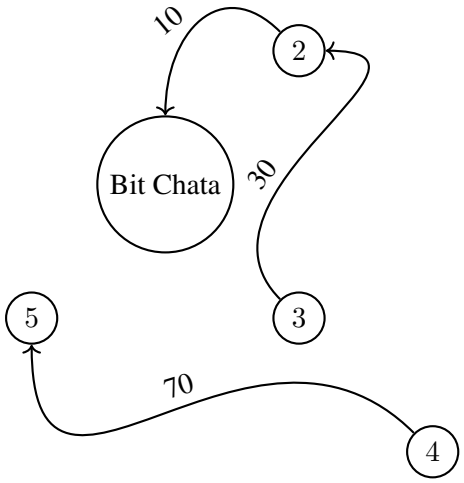
W przykładzie pierwszym, system szlaków w Górach Bajtockich możemy przedstawić jak na rysunku poniżej. Punkty węzłowe oznaczono liczbami w kółkach, zaś czasy przejść pomiędzy nimi podano na strzałkach reprezentujących szlaki. Oczywiście szlaki są jednokierunkowe i turyści nie mogą poruszać się „pod prąd".

Aby system powiadamiania o zagrożeniach testowany przez SRGB działał skutecznie, powinien przekazywać informacje o najkrótszych możliwych czasach przejścia z każdego węzła do schroniska. Wówczas turysta znajdujący się w dowolnym miejscu w górach będzie mógł ocenić czy zdoła dotrzeć do Bit Chaty, czy też musi szukać innego bezpiecznego miejsca zanim pogoda się popsuje.



Jeśli jakiś turysta znajduje się blisko węzła nr 2, to najszybsza ścieżka z tego węzła do Bit Chaty wynosi $10 + 20 = 30$ bajtockich jednostek czasu i prowadzi przez węzeł numer 4. Z węzła nr 3, najszybsza droga do schroniska wiedzie przez węzły nr 2 oraz 4 i wynosi $50 + 10 + 20 = 80$. Natomiast z węzła czwartego możemy dostać się do Bit Chaty bezpośrednio w czasie 20. Wartości te oraz kolejność, w jakiej zostały podane stanowią prawidłową odpowiedź, którą program powinien zwrócić na standardowe wyjście.

Przykład drugi opisuje poniższy rysunek:



W tym przypadku widać, że z węzłów nr 4 i 5 nie można dostać się do Bit Chaty. Z tego względu w linijce 3 i 4 program zwróci wartość -1.

Pozostałe dwa przykłady pozostawiamy czytelnikowi do samodzielnej analizy.

76.2 Rozwiązanie

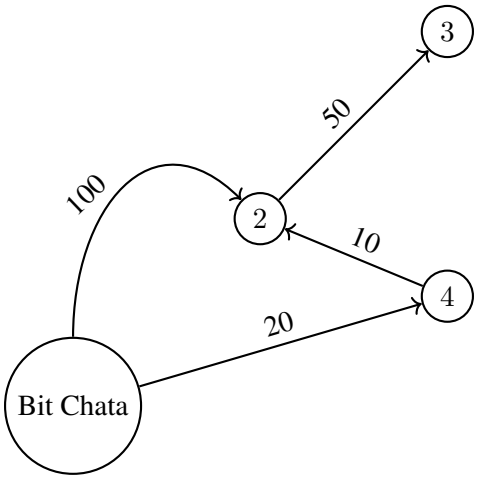
System szlaków w Górach Bajtockich możemy logicznie przedstawić w postaci grafu skierowanego z wagami, którego pierwszym wierzchołkiem zawsze jest Bit Chata. Punkty węzłowe szlaków to pozostałe wierzchołki tego grafu, zaś czasy przejść pomiędzy nimi to wagi jego krawędzi. Graf jest skierowany, ponieważ szlaki są jednokierunkowe (turyści nie mogą poruszać się „pod prąd"). Naszym zadaniem jest podanie informacji o najkrótszych możliwych czasach przejścia z każdego węzła do węzła nr 1.

Aby wyznaczyć prawidłowe wartości należy odwrócić sposób patrzenia na sieć szlaków i zadać sobie pytanie: „Ile wynoszą najkrótsze ścieżki ze schroniska do wszystkich pozostałych punktów węzłowych?"

W teorii grafów odpowiedź na to pytanie w optymalnym czasie otrzymamy stosując algorytm Dijkstry [4] (p. wprowadzenie do tego działu – sekcja 71.13). Należy jedynie „odwrócić" wcześniej graf, czyli zmienić zwrot każdej krawędzi na przeciwny (najlepiej tego dokonać już na etapie wczytywania grafu). Dla przykładu, rozważmy poniższe dane wejściowe:

```
4 4
2 1 100
4 1 20
2 4 10
3 2 50
```

Drugi wiersz oznacza, że z wierzchołka drugiego prowadzi do wierzchołka pierwszego krawędź o wadze 100, więc powinniśmy to zapisać w naszej strukturze danych reprezentującej graf w sposób odwrotny – czyli, że to z 1 prowadzi krawędź do 2. Trzeba tak zrobić dla wszystkich wczytywanych krawędzi, a następnie uruchomić algorytm Dijkstry z wierzchołka nr 1, czyli Bit Chaty.



76.2.1 Python

Źródło: ./zadania/08_Grafy/Bit_Chata/solution.py.

```
import sys
from collections import defaultdict
from queue import PriorityQueue
#algorytm Dijkstry
def Dijkstra(start):
    Q = PriorityQueue()
    visit[start]=1
    Q.put((0,start))

    while not(Q.empty()):
        v=Q.get()
        if not(visit[v[1]]):
            distance[v[1]]=v[0]
            visit[v[1]]=1
        for i in range(len(graph[v[1]])):
            if not(visit[graph[v[1]][i][0]]):
                new=(graph[v[1]][i][1]+v[0],graph[v[1]][i][0])
                Q.put(new)

#wczytanie grafu jako lista sąsiedztwa
n, m = input().split()
n = int(n)
m = int(m)
graph = defaultdict(list)
visit = defaultdict(int)
distance = defaultdict(int)

for i in range(m):
    a, b, c = input().split()
    a = int(a) #koniec
    b = int(b) #początek
    c = int(c) #waga
    edge = (a,c)
    graph[b].append(edge)

#Główna część
for i in range(n+1):
    distance[i]=-1

Dijkstra(1)

for i in range(2,n+1):
    print(distance[i])
```

76.2.2 C++

Źródło: ./zadania/08_Grafy/Bit_Chata/solution.cpp.

```cpp
#include <cstdlib>
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
#define MAXN 100000 //maksymalna liczba wierzcholkow
#define INF 1000000000000001LL //nieskonczonosc
using namespace std;

vector<pair<int,long long> > kraw[MAXN+1];
vector<long long> waga[MAXN+1];
long long dis[MAXN+1];
int vis[MAXN+1];

void Dijkstra(int n) {
  priority_queue<pair<long long,int>,vector<pair<long long,int> >, greater<pair<long long,int> > > Q;
  vis[n]=1;
  Q.push(make_pair(0,n));
  while(!Q.empty()) {
    pair<long long,int> v=Q.top();Q.pop();
    if(!vis[v.second]) {
      dis[v.second]=v.first;
       vis[v.second]=1;
    }
    for(int i=0;i<(int)kraw[v.second].size();i++)
        if(!vis[kraw[v.second][i].first])
           Q.push(make_pair(kraw[v.second][i].second+v.first,kraw[v.second][i].first));
  }
}

int n, m, a, b, c;

int main() {
  scanf("%d%d",&n,&m);
  while(m--) {
    scanf("%d%d%d",&a,&b,&c);
    kraw[b].push_back(make_pair(a,c));
  }
  for(int i=0;i<=n;i++)
    dis[i]=-1;
  Dijkstra(1);
  for(int i=2;i<=n;i++)
      printf("%lld\n",dis[i]);
   return 0;
}
```

77 (VIII) – Archipelag – Gremliny

*Autor: **Jan Filipowicz**, **Politechnika Łódzka***
*Kategoria: **Gremliny (II edycja – zawody algorytmiczne – etap finałowy)***
*Język programowania: **C++/Python***

77.1 Treść zadania

Witryna aukcyjna eBajt planuje w nadchodzącym kwartale poszerzyć strefę objętą swoim serwisem dostawczym o teren Archipelagu Bajtuskiego. Jest to odważne posunięcie, albowiem ów archipelag posiada reputację obszaru notorycznie sprawiającego problemy pod względem logistycznym. Nie tylko składa się on z n wysp połączonych k mostami, ale na dodatek są to mosty bardzo różnej jakości, co wiąże się z różnicami w ich dopuszczalnym udźwigu.

W rezultacie trudno ocenić maksymalną dozwoloną wagę ciężarówki dostawczej do przewozu przedmiotów między dwoma wyspami. Tymczasem serwery eBajtu muszą być gotowe odpowiadać na tego typu pytania nawet tysiące razy na sekundę, bowiem uzależniona jest od tego cena dostawy, a tę trzeba wyświetlić na ekranie użytkownika już kiedy przegląda stronę przedmiotu licytacji.

Zadanie

Znając układ wysp i mostów Archipelagu Bajtuskiego, a także dopuszczalną wagę pojedynczego pojazdu na każdym z mostów, odpowiedz na q zapytań o maksymalną wagę ciężarówki, którą da się przejechać między dwoma zadanymi wyspami (niekoniecznie najkrótszą możliwą trasą).

Opis wejścia

Pierwsza linia standardowego wejścia zawiera 3 liczby całkowite:
n, k, q ($2 \leq n \leq 100\,000$; $1 \leq k \leq 400\,000$; $1 \leq q \leq 100\,000$) ,
oddzielone pojedynczymi spacjami – odpowiednio liczbę wysp, mostów i zapytań.

Kolejne k linii zawiera opisy mostów, przy czym każda i-ta z nich składa się z 3 liczb całkowitych:
a_i, b_i, w_i ($1 \leq a_i, b_i \leq n$; $a_i \neq b_i$; $1 \leq w_i \leq 10^9$) ,
oddzielonych spacjami, gdzie a_i i b_i są numerami wysp połączonych i-tym mostem, zaś w_i oznacza maksymalną wagę pojazdu dozwoloną na tym moście.

Jest zagwarantowane, że z każdej wyspy da się pewną sekwencją mostów dojechać na każdą inną oraz że nie ma pary wysp, między którymi istniałby więcej niż jeden most.

Następne q linii zawiera opisy zapytań, przy czym każda j-ta linia składa się z 2 liczb całkowitych:
u_j, v_j ($1 \leq u_j, v_j \leq n$; $u_j \neq v_j$) .
Są to odpowiednio: numer wyspy, z której miałaby wyruszyć ciężarówka dostawcza i numer wyspy docelowej.

Opis wyjścia

Na standardowe wyjście należy wypisać q linii. Każda z nich powinna zawierać jedną liczbę całkowitą, przy czym j-ta linia powinna zawierać odpowiedź na j-te zapytanie, tj. maksymalną wagę ciężarówki, którą da się przejechać z wyspy u_j na wyspę v_j.

Przykład

Dla przykładowego, podanego poniżej wejścia:

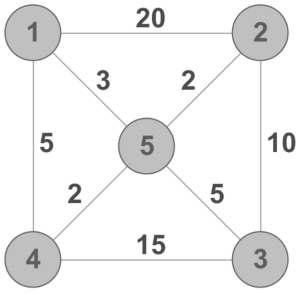
```
5 8 3
1 2 20
2 3 10
3 4 15
4 1 5
5 1 3
5 2 2
5 4 2
5 3 5
2 1
2 4
1 5
```

prawidłową odpowiedzią jest:

```
20
10
5
```

Wyjaśnienie

Podany przykład odpowiada przedstawionemu poniżej układowi wysp i mostów:



Optymalna droga z wyspy 2 na wyspę 1 przebiega bezpośrednim mostem między nimi o maksymalnym udźwigu 20. Najlepsza droga między wyspami 2 i 4 przebiega przez wyspę 3 i wymaga przebycia dwóch mostów o wartościach udźwigu 10 i 15. Waga ciężarówki musi mieścić się w obu tych ograniczeniach, zatem wynik to 10. Z wyspy 1 da się wprawdzie dojechać na wyspę 5 bezpośrednim mostem, ale jego udźwig to jedynie 3. Bardziej opłacalna jest trasa przebiegająca przez sekwencję wysp 1–4–3–5 lub 1–2–3–5. W obu przypadkach maksymalna dopuszczalna waga ciężarówki wynosi 5.

77.2 Rozwiązanie

Archipelag składa się z n wysp i k mostów, każdy o pewnym maksymalnym udźwigu. Dla każdego z q zapytań znajdź maksymalną wagę ciężarówki, którą można przejechać między dwoma zadanymi wyspami.

Łatwo dojść do wniosku, że niektóre mosty w ogóle nie uczestniczą w optymalnych ścieżkach – jeśli między wyspami A i B istnieje most AB o maksymalnym udźwigu w_{AB}, a także pewna ścieżka o maksymalnym udźwigu $w_s > w_{AB}$, to most AB jest nieprzydatny, bo zawsze bardziej będzie się bardziej opłacało skorzystać z owej ścieżki. Takie rozumowanie pozwala zawsze sprowadzić graf archipelagu do postaci drzewa – a konkretnie do maksymalnego drzewa rozpinającego (ang. *maximum spanning tree*) [4][2]. Istnieje kilka klasycznych algorytmów[1] znajdujących owo drzewo w czasie $O(|E| \log |V|)$.

Aby szybko odpowiadać na zapytania, można wykorzystać strukturę wykorzystywaną do odpowiedzi na zapytania o najniższego wspólnego przodka (ang. *lowest common ancestor*, LCA) [5]. Tradycyjnie dla każdego wierzchołka przechowujemy w takiej strukturze jego przodka oddalonego o kolejne potęgi liczby 2. W naszym przypadku, oprócz tego powinniśmy przechowywać także najmniejszą wagę na ścieżce do każdego takiego przodka. Aby odpowiedzieć na zapytanie o najmniejszą wagę między A i B, odnajdujemy $C = \text{LCA}(A, B)$ oraz najmniejszą wagę między A i C oraz między B i C.

Analiza złożoności:

Wydajny algorytm tworzący strukturę LCA działa w czasie $O(|V| \log |V|)$, a struktura pozwala odpowiedzieć na pojedyncze zapytanie w czasie $O(\log |V|)$. Wraz z utworzeniem maksymalnego drzewa rozpinającego, złożoność całego algorytmu to zatem $O(k \log n + n \log n + q \log n)$. Wiemy, że graf jest spójny, zatem $k \geq n-1$, więc możemy uprościć to wyrażenie do postaci $O((k+q) \log n)$.

[1] W praktyce, możemy tu stosować te same algorytmy, których używamy do znalezienia *minimalnego* drzewa rozpinającego (wystarczy odpowiednio zmodyfikować – np. zanegować – wagi krawędzi) – p. sekcja 71.13.

77.2.1 Python

Źródło: ./zadania/08_Grafy/Archipelag/solution.py.

```
class Krawedz():
    def __init__(self, poczatek, koniec, waga):
        self.poczatek = poczatek
        self.koniec = koniec
        self.waga = waga

class KrawedzSkierowana():
    def __init__(self, koniec, waga):
        self.koniec = koniec
        self.waga = waga

class WierzcholekDrzewa():
    def __init__(self, glebokosc, przodkowie):
        self.glebokosc = glebokosc
        self.przodkowie = przodkowie

class StrukturaZbiorowRolacznych():
    def __init__(self, n):
        self._rodzice = list(range(n))
        self._rzedy = n * [0]
    def znajdz(self, indeks):
        if self._rodzice[indeks] != indeks:
            self._rodzice[indeks] = self.znajdz(self._rodzice[indeks])
        return self._rodzice[indeks]
    def polacz(self, lhs, rhs):
        lhs_korzen = self.znajdz(lhs)
        rhs_korzen = self.znajdz(rhs)
        if lhs_korzen == rhs_korzen:
            return False
        if self._rzedy[lhs_korzen] < self._rzedy[rhs_korzen]:
            self._rodzice[lhs_korzen] = rhs_korzen
        else:
            self._rodzice[rhs_korzen] = lhs_korzen
            if self._rzedy[lhs_korzen] == self._rzedy[rhs_korzen]:
                self._rzedy[lhs_korzen] += 1
        return True

n, k, q = [int(x) for x in input().split()]
krawedzie = []
for _ in range(k):
    poczatek, koniec, waga = [int(x) for x in input().split()]
    krawedzie.append(Krawedz(poczatek - 1, koniec - 1, waga))
krawedzie.sort(key = lambda kraw: kraw.waga, reverse = True)
las = StrukturaZbiorowRolacznych(n)
drzewoRozpinajace = [[] for _ in range(n)]
for i in range(k):
    kraw = krawedzie[i]
    if las.polacz(kraw.poczatek, kraw.koniec):
```

```
        drzewoRozpinajace[kraw.poczatek].append(KrawedzSkierowana(kraw.koniec, kraw.waga))
        drzewoRozpinajace[kraw.koniec].append(KrawedzSkierowana(kraw.poczatek, kraw.waga))
drzewoUkorzenione = [WierzcholekDrzewa(0, []) for _ in range(n)]
odwiedzono = n * [False]
stos = [0]
while len(stos) > 0:
    wierzcholek = stos.pop()
    odwiedzono[wierzcholek] = True
    for kraw in drzewoRozpinajace[wierzcholek]:
        if not odwiedzono[kraw.koniec]:
            stos.append(kraw.koniec)
            dziecko = drzewoUkorzenione[kraw.koniec]
            dziecko.glebokosc = drzewoUkorzenione[wierzcholek].glebokosc + 1
            dziecko.przodkowie.append(KrawedzSkierowana(wierzcholek, kraw.waga))
            i = 0
            while i < len(drzewoUkorzenione[dziecko.przodkowie[i].koniec].przodkowie):
                zrodlowaKraw = drzewoUkorzenione[dziecko.przodkowie[i].koniec].przodkowie[i]
                dziecko.przodkowie.append(
                    KrawedzSkierowana(zrodlowaKraw.koniec,
                                      min(zrodlowaKraw.waga, dziecko.przodkowie[i].waga)))
                i += 1
for _ in range(q):
    a, b = [int(x) - 1 for x in input().split()]
    if drzewoUkorzenione[a].glebokosc < drzewoUkorzenione[b].glebokosc:
        a, b = b, a
    roznicaGlebokosci = drzewoUkorzenione[a].glebokosc - drzewoUkorzenione[b].glebokosc
    wynik = 2000000000
    i = 0
    while 1 << i <= roznicaGlebokosci:
        if roznicaGlebokosci & 1 << i:
            wynik = min(wynik, drzewoUkorzenione[a].przodkowie[i].waga)
            a = drzewoUkorzenione[a].przodkowie[i].koniec
        i += 1
    while a != b:
        skok = len(drzewoUkorzenione[a].przodkowie) - 1
        while skok > 0 and drzewoUkorzenione[a].przodkowie[skok].koniec \
                == drzewoUkorzenione[b].przodkowie[skok].koniec:
            skok -= 1
        wynik = min(wynik, drzewoUkorzenione[a].przodkowie[skok].waga)
        a = drzewoUkorzenione[a].przodkowie[skok].koniec
        wynik = min(wynik, drzewoUkorzenione[b].przodkowie[skok].waga)
        b = drzewoUkorzenione[b].przodkowie[skok].koniec
    print(wynik)
```

77.2.2 C++

Źródło: ./zadania/08_Grafy/Archipelag/solution.cpp.

```cpp
#include <algorithm>
#include <climits>
#include <cstdlib>
#include <iostream>
#include <numeric>
#include <utility>
#include <vector>

struct krawedz {
  int poczatek {};
  int koniec {};
  int waga {};
};

struct krawedz_skierowana {
  int koniec {};
  int waga {};
};

struct wierzcholek_drzewa {
  int glebokosc {};
  std::vector<krawedz_skierowana> przodkowie;
};

class struktura_zbiorow_rozlacznych {
public:
  struktura_zbiorow_rozlacznych(int n)
    : rodzice(n), rzedy(n) {
    std::iota(rodzice.begin(), rodzice.end(), 0);
  }
  int znajdz(int indeks) {
    int& p = rodzice[indeks];
    if (p != indeks)
      p = znajdz(p);
    return p;
  }
  bool polacz(int lhs, int rhs) {
    const int lhs_korzen = znajdz(lhs);
    const int rhs_korzen = znajdz(rhs);
    if (lhs_korzen == rhs_korzen)
      return false;
    int& lhs_rzad = rzedy[lhs_korzen];
    const int rhs_rzad = rzedy[rhs_korzen];
    if (lhs_rzad < rhs_rzad) {
      rodzice[lhs_korzen] = rhs_korzen;
    } else {
      rodzice[rhs_korzen] = lhs_korzen;
      if (lhs_rzad == rhs_rzad)
```

```
        lhs_rzad++;
    }
    return true;
  }
private:
  std::vector<int> rodzice;
  std::vector<int> rzedy;
};

bool porownaj_wagi(const krawedz& lhs, const krawedz& rhs) {
  return lhs.waga < rhs.waga;
}

int main() {
  int n, k, q;
  std::cin >> n >> k >> q;
  std::vector<krawedz> krawedzie(k);
  for (int i = 0; i < k; i++) {
    krawedz& kraw = krawedzie[i];
    std::cin >> kraw.poczatek >> kraw.koniec >> kraw.waga;
    kraw.poczatek--;
    kraw.koniec--;
  }
  std::sort(krawedzie.rbegin(), krawedzie.rend(), porownaj_wagi);
  struktura_zbiorow_rozlacznych las(n);
  std::vector<std::vector<krawedz_skierowana>> drzewo_rozpinajace(n);
  for (int i = 0; i < k; i++) {
    const krawedz& kraw = krawedzie[i];
    if (las.polacz(kraw.poczatek, kraw.koniec)) {
      drzewo_rozpinajace[kraw.poczatek].push_back({kraw.koniec, kraw.waga});
      drzewo_rozpinajace[kraw.koniec].push_back({kraw.poczatek, kraw.waga});
    }
  }
  std::vector<wierzcholek_drzewa> drzewo_ukorzenione(n);
  std::vector<bool> odwiedzono(n);
  std::vector<int> stos {0};
  while (!stos.empty()) {
    int wierzcholek = stos.back();
    stos.pop_back();
    odwiedzono[wierzcholek] = true;
    int liczba_krawedzi = drzewo_rozpinajace[wierzcholek].size();
    for (int i = 0; i < liczba_krawedzi; i++) {
      const krawedz_skierowana& kraw = drzewo_rozpinajace[wierzcholek][i];
      if (!odwiedzono[kraw.koniec]) {
        stos.push_back(kraw.koniec);
        wierzcholek_drzewa& dziecko = drzewo_ukorzenione[kraw.koniec];
        dziecko.glebokosc = drzewo_ukorzenione[wierzcholek].glebokosc + 1;
        dziecko.przodkowie.push_back({wierzcholek, kraw.waga});
        for (std::size_t j = 0;
          j < drzewo_ukorzenione[dziecko.przodkowie[j].koniec].przodkowie.size(); j++) {
          krawedz_skierowana nowa_kraw
            = drzewo_ukorzenione[dziecko.przodkowie[j].koniec].przodkowie[j];
```

```
          nowa_kraw.waga = std::min(nowa_kraw.waga, dziecko.przodkowie[j].waga);
          dziecko.przodkowie.push_back(nowa_kraw);
        }
      }
    }
  }
  for (int i = 0; i < q; i++) {
    int a, b;
    std::cin >> a >> b;
    a--;
    b--;
    if (drzewo_ukorzenione[a].glebokosc < drzewo_ukorzenione[b].glebokosc)
      std::swap(a, b);
    int roznica_glebokosci = drzewo_ukorzenione[a].glebokosc - drzewo_ukorzenione[b].glebokosc;
    int wynik = INT_MAX;
    for (int j = 0; 1 << j <= roznica_glebokosci; j++) {
      if (roznica_glebokosci & 1 << j) {
        wynik = std::min(wynik, drzewo_ukorzenione[a].przodkowie[j].waga);
        a = drzewo_ukorzenione[a].przodkowie[j].koniec;
      }
    }
    while (a != b) {
      int skok = drzewo_ukorzenione[a].przodkowie.size() - 1;
      while (skok > 0 && drzewo_ukorzenione[a].przodkowie[skok].koniec
        == drzewo_ukorzenione[b].przodkowie[skok].koniec) {
        skok--;
      }
      wynik = std::min(wynik, drzewo_ukorzenione[a].przodkowie[skok].waga);
      a = drzewo_ukorzenione[a].przodkowie[skok].koniec;
      wynik = std::min(wynik, drzewo_ukorzenione[b].przodkowie[skok].waga);
      b = drzewo_ukorzenione[b].przodkowie[skok].koniec;
    }
    std::cout << wynik << '\n';
  }
  return 0;
}
```

78 (VIII) – Daleka droga do szkoły – Gremliny

*Autor: **Przemysław Gordinowicz**, **Politechnika Łódzka***
*Kategoria: **Gremliny (IV edycja – zawody algorytmiczne – etap finałowy)***
*Język programowania: **C++/Python***

78.1 Treść zadania

Jarek Zadzior, mieszkaniec pewnego (całkiem dużego, jak się zaraz okaże) miasteczka, nieszczególnie lubi chodzić do szkoły. A w zasadzie jeździć, bo gdy tylko pogoda na to pozwala, jeździ do niej rowerem. Ostatnio bardzo zaciekawiły go interesujące problemy algorytmiczne, które poznał na kółku... a tu akurat nieszczęście – poniedziałek. Do tego rano i trzeba jechać do szkoły.

Rodzicom Jarek nie będzie się sprzeciwiał, bo w sumie są OK i komputer mu wymienili ostatnio, ale do szkoły jechać, tak po prostu, gdy tyle rzeczy do przemyślenia, też się Jarkowi nie chce. *A może*, pomyślał Jarek, *pojadę do szkoły **najdłuższą** możliwą drogą? Zawsze mogę powiedzieć, że przecież jadę cały czas, próbuję, a to nie moja wina, że daleko mam. Tylko jak tę najdłuższą drogę wyznaczyć*, pomyślał, *najkrótsze to już znam, ale to chyba nie powinno być trudniejsze...*

Trzeba wiedzieć, że w miasteczku Jarka wszystkie drogi rowerowe są jednokierunkowe, dla bezpieczeństwa. Krzyżują się wybranych punktach miasteczka. Jarek mieszka przy jednym z takich skrzyżowań i tam zaczyna swoją podróż. Szkoła jest przy innym skrzyżowaniu. Ze względów strategicznych, Jarek nie może ponownie przejechać przez skrzyżowanie koło swojego domu (jeśli rodzice go zobaczą to szlaban pewny), a gdy dojedzie do skrzyżowania przy szkole musi zakończyć podróż, zaparkować rower i wejść do szkoły... no bo jakby to wyglądało, gdyby w poniedziałkowy poranek *wyjeżdżał* ze szkoły? Ktoś zauważy, powie komu nie trzeba i szlaban.

To w sumie też ciekawy problem, pomyślał Jarek, *może przeanalizuję go w drodze do szkoły, możliwie **najdłuższej** drodze.*

Zadanie

Napisz program, który rozwiąże problem Jarka, to znaczy wczyta opis sieci dróg rowerowych w jego miasteczku (albo jakimś innym, bo Jarek myśli nad uniwersalnym rozwiązaniem) i wskaże długość najdłuższej drogi łączącej wskazane dwa skrzyżowania. Program musi uwzględnić także sytuację, w której nie istnieje żadna poprawna droga do szkoły, lub w której istnieje nieograniczenie długa droga. W pierwszej sytuacji rodzice mogliby nie pozwolić na jazdę rowerem, a z kolei ta druga – to marzenie Jarka.

Opis wejścia

Pierwszy wiersz danych zawiera dwie liczby naturalne n (liczbę skrzyżowań) i m (liczbę odcinków dróg rowerowych). Liczby spełniają warunki: $2 \leq n \leq 6\,000$, $1 \leq m \leq 20\,000$ (więc w sumie to całkiem wielkie

miasto może być).

Kolejne m wierszy zawiera po trzy dodatnie liczby całkowite, dwie pierwsze z zakresu od 1 do n – numery skrzyżowań i trzecią – długość odcinka drogi (w metrach), z zakresu od 1 do 1 000. Liczby w wierszu oddzielone są pojedynczym odstępem.

Należy przyjąć, że dom Jarka znajduje się przy skrzyżowaniu numer 1, a szkoła – przy skrzyżowaniu numer n. Ponadto między dwoma skrzyżowaniami jest co najwyżej jedna droga w danym kierunku.

Opis wyjścia

Program powinien wypisać:

- Słowo *BRAK* (4 wielkie litery), gdy nie ma drogi łączącej skrzyżowanie numer 1 ze skrzyżowaniem numer n;
- Słowo *WAGARY* (6 wielkich liter), gdy istnieje droga do szkoły oraz istnieje nieograniczenie długa droga wychodząca z domu Jarka (i nie przechodząca już koło niego) – nawet, jeśli nie prowadziłaby do szkoły;
- jedną liczbę całkowitą oznaczającą długość najdłuższej drogi do szkoły (drogi łączącej skrzyżowanie numer 1 ze skrzyżowaniem numer n), gdy nie zachodzi żadna z powyższych sytuacji.

Przykłady

Przykład 1.

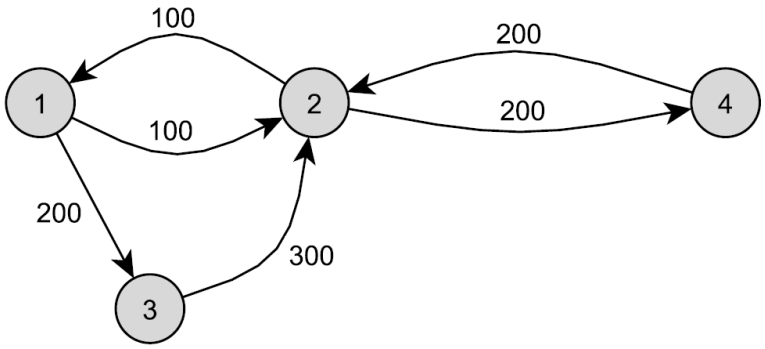
Dla danych wejściowych:

```
4 6
1 2 100
1 3 200
2 1 100
2 4 200
3 2 300
4 2 200
```

program powinien wypisać:

```
700
```

Najdłuższa droga prowadzi przez skrzyżowania 1, 3, 2, 4.



Rysunek 78.1: Graf do przykładu 1

Przykład 2.

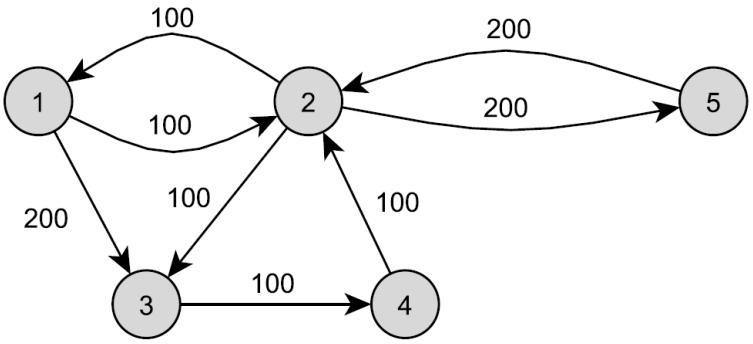
Dla danych wejściowych:

```
5 8
1 2 100
1 3 200
2 1 100
2 3 100
2 5 200
3 4 100
4 2 100
5 2 200
```

program powinien wypisać:

```
WAGARY
```

Jarek może jeździć dowolnie długo nie dojeżdżając ani do domu (skrzyżowanie numer 1) ani do szkoły (numer 5).



Rysunek 78.2: Graf do przykładu 2

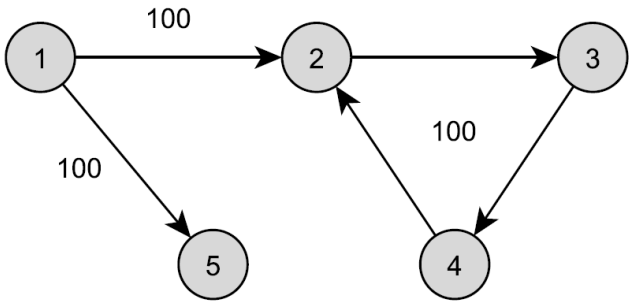
Przykład 3.

Analogiczna sytuacja zachodzi dla danych wejściowych:

```
5 5
1 2 100
2 3 100
3 4 100
4 2 100
1 5 100
```

A zatem również tu program powinien wypisać:

```
WAGARY
```



Rysunek 78.3: Graf do przykładu 3

Przykład 4.

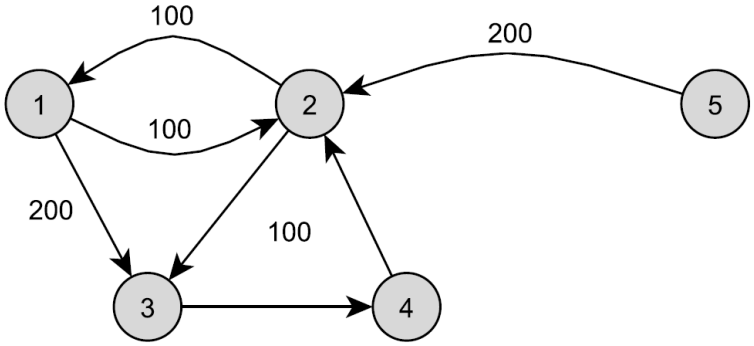
Natomiast dla danych wejściowych:

```
5 7
1 2 100
1 3 200
2 1 100
2 3 100
3 4 100
4 2 100
5 2 200
```

program powinien wypisać:

```
BRAK
```

Nie można dojechać rowerem z domu Jarka do szkoły.



Rysunek 78.4: Graf do przykładu 4

Uwaga

Dla programujących w Pythonie: w wypadku rozwiązania korzystającego z rekurencji warto pamiętać, że w Pythonie głębokość rekursji jest domyślnie ograniczona do 1 000. Aby sobie z tym poradzić można zrezygnować z rekurencji lub zmienić ten limit następującym kodem (w którym n oznacza pożądany limit – warto podać go z pewnym zapasem):

```
import sys
sys.setrecursionlimit(n)
```

78.2 Rozwiązanie

Rozwiązanie sprowadza się do znalezienia *najdłuższej drogi* między wierzchołkami w grafie skierowanym, lub stwierdzenia, że taka droga nie istnieje. Wskazany problem algorytmiczny ma zastosowanie praktyczne w szeregowaniu zadań i planowaniu procesów – wówczas najdłuższa droga wskazuje łączny czas trwania złożonego procesu, a zarazem miejsca ważne dla ewentualnej optymalizacji.

Rozważany graf skierowany, to podana w treści zadania mapa miasteczka. Skrzyżowania są *wierzchołkami* grafu, a odcinki dróg rowerowych (jednokierunkowych) – łukami (skierowanymi krawędziami) grafu. Co istotne, z treści zadania wynika, że w konstrukcji grafu (najlepiej już na etapie wczytywania danych) należy pominąć łuki wchodzące do wierzchołka numer 1 (dom) i łuki wychodzące z wierzchołka numer n (szkoła). Realizujący to kod w Pythonie może wyglądać tak:

```
rd = lambda: map(int, input().strip().split())  # wczytywanie linii danych
n, m = rd()
G = [[] for i in range(n + 1)]
for i in range(m):
    u, v, w = rd()
    if (u != n) and (v != 1):
        G[u].append((v, w))
```

Na kolejnym etapie, przy użyciu algorytmu przechodzenia grafu (np. BFS lub DFS – p. sekcja 71.12) należy ograniczyć badany graf do wierzchołków i łuków *osiągalnych* z wierzchołka 1. Po wykonaniu poniższego, przykładowego kodu (inicjowanego przez `T = {1}`), zbiór `T` zawierać będzie wszystkie węzły osiągalne z 1:

```
def BFS():
    q = [1]
    while len(q) > 0:
        u = q.pop(0)
        for (v, w) in G[u]:
            if v not in T:
                T.add(v)
                q.append(v)
```

Wyznaczony powyżej zbiór `T` powinien zawierać wierzchołek numer n. Jego brak oznacza brak drogi z domu do szkoły. Co więcej, graf ograniczony do zbioru `T` i łuków wychodzących z wierzchołków zawartych w `T` powinien być *acykliczny* – w przeciwnym razie Jarek może dowolnie długo jeździć po istniejącym cyklu. Acykliczność grafu skierowanego najwygodniej sprawdza się wyznaczając *porządek topologiczny* – taki porządek wierzchołków (zrealizowany poniżej przez listę `order`), w którym wszystkie łuki są skierowane zgodnie z tym porządkiem (tzn. łuk $\overrightarrow{uv}$ może istnieć tylko, gdy u jest przed v na liście `order`). Porządek topologiczny można wyznaczyć stosując *algorytm Kahna* [4] – sukcesywnie wstawiając na listę wierzchołki o zerowym *stopniu wejściowym* (o zerowej liczbie nieprzetworzonych łuków wchodzących do wierzchołka) i przetwarzając łuki wychodzące z takich wierzchołków.

```
16 def TopoSort():
17     indegree = [0] * (n + 1)
18     for u in T:
19         for (v, w) in G[u]:
20             indegree[v] += 1
21 
22     S = [1]      # wierzchołki osiągalne z 1 o zerowym indegree
23     while len(S) > 0:
24         u = S.pop()
25         T.discard(u)
26         order.append(u)
27         for (v, w) in G[u]:
28             indegree[v] -= 1    # zmniejszamy stopień wejściowy v
29             if indegree[v] == 0:
30                 order.append(v)  # umieszczamy v w porządku topologicznym
31                 S.append(v)      # i wstawiany do kolejki wierzchołków
32                                  # do przetworzenia
33     return len(T)
```

Zwrócenie przez powyższą funkcję innej wartości niż 0 oznacza istnienie cyklu (w zbiorze T pozostały wierzchołki o niezerowym stopniu wejściowym) i brak możliwości ustalenia porządku topologicznego. Z kolei mając ustalony ten porządek, można – przeglądając wierzchołki zgodnie z nim – wyznaczyć najdłuższą drogę do każdego wierzchołka osiągalnego z 1.

```
34 def CalculateDist():
35     dist = [0] * (n + 1)
36     for u in order:
37         for (v, w) in G[u]:
38             if dist[v] < dist[u] + w:
39                 dist[v] = dist[u] + w  # można wydłużyć drogę
40     return dist[n]
```

Dla porządku, na koniec podano kod łączący całość w gotowe rozwiązanie:

```
41 T = {1}      # zbiór wierzchołków osiągalnych z 1
42 BFS()
43 if n not in T:
44     print("BRAK")
45 else:
46     order = [1]
47     result = TopoSort()
48     if result == 0:
49         print(CalculateDist())
50     else:
51         print("WAGARY")
```

Uwagi

Porządek topologiczny można dość łatwo uzyskać (może nawet łatwiej niż powyżej), przechodząc graf w głąb, czyli za pomocą algorytmu DFS (jest to podstawowy algorytm sortowania topologicznego opisany w [4]). Przykładowy kod, wykorzystujący rekurencyjną implementację algorytmu DFS, może wyglądać jak poniżej:

```
1 def DFS(u):
2     for (v, w) in G[u]:
3         if value[v] == 0:
4             value[0] += 1 # tyle odwiedzonych wierzchołków
5             value[v] = value[0] # v odwiedzony jako kolejny
6             DFS(v)
7     value[u] = n * value[0] - value[u]  #wartość kontrolna porządku
8     order.insert(0, u) # u na początek w porządku
```

W powyższym przykładzie (kod należy uruchomić rekurencyjnie od `DFS(1)`), uzyskana lista `order` wyznacza porządek topologiczny w grafie acyklicznym. Do sprawdzenia acykliczności używana jest dodatkowa tablica wartości kontrolnych `value`. W każdej chwili `value[0]` oznacza liczbę odwiedzonych wierzchołków. Po odwiedzeniu wierzchołka v (linia 5), w `value[v]` tymczasowo zostaje zapamiętane, jako który wierzchołek został on odwiedzony. Z kolei po przetworzeniu wierzchołka u (linia 7) i wstawieniu go do uporządkowanej listy `order`, wartość `value[u]` jest równa liczbie wierzchołków odwiedzonych w tym momencie (pomnożonej przez odpowiednio dużą liczbę) minus wcześniej zapamiętana wartość w momencie odwiedzenia. Graf jest acykliczny (a zatem `order` jest porządkiem topologicznym), gdy wierzchołki będące wcześniej w porządku mają większe wartości kontrolne. Powyższy kod pozwala jednocześnie sprawdzić, czy wierzchołek n jest osiągalny z 1 (warunek `value[n]` $\neq 0$).

W wypadku rozwiązania korzystającego z rekurencji (DFS najłatwiej implementować w sposób rekurencyjny), warto pamiętać, że w Pythonie głębokość rekursji jest domyślnie ograniczona do 1000. Jak wyjaśniono na końcu treści zadania, problem ten można rozwiązać rezygnując z rekurencji lub zmieniając ten limit za pomocą funkcji `sys.setrecursionlimit(n)`.

Prostsze, acz wyraźnie mniej efektywne, jest rozwiązanie zadania z wykorzystaniem algorytmu Bellmana-Forda [4] (p. wprowadzenie do tego działu – sekcja 71.13). Zasadniczo służy on do wyznaczania *najkrótszych dróg* w grafie, ale radzi sobie z ujemnymi wagami łuków (a do tego pozwala wyłapać istnienie cykli o ujemnej wadze). Wystarczy więc pomnożyć wszystkie długości odcinków dróg przez -1 i użyć tego algorytmu, aby znaleźć drogę o najmniejszej łącznej wadze.

78.2.1 Python

Sortowanie topologiczne przez DFS [4]:

Źródło: ./zadania/08_Grafy/Daleka_droga_do_szkoly/solutionDFS.py.

```
import sys
sys.setrecursionlimit(6005) # żeby nie było błędu rekursji

rd = lambda: map(int, input().strip().split())  # wczytywanie danych
# liczba skrzyżowań/dróg
n, m = rd()
G = [[] for i in range(n + 1)]
# wczytywanie grafu/planu miasta - para wierzchołek/waga
for i in range(m):
    u, v, w = rd()
    # ignorujemy drogi wychodzące ze szkoły (n) i wchodzące do domu (1)
    if (u != n) and (v != 1):
        G[u].append((v, w))

def DFS(u):
    # przechodzenie w głąb, ze zliczaniem kolejności (porządek topologiczny)
    for (v, w) in G[u]:
        if value[v] == 0:
            value[0] += 1 # tyle odwiedzonych wierzchołków
            value[v] = value[0] # v odwiedzony jako kolejny
            DFS(v)
    value[u] = n * value[0] - value[u]  #wartość kontrolna porządku
    order.insert(0, u) # u na początek w porządku

def CalculateDist():
    # wyznaczanie dystansów i zwracanie wyniku
    if value[n] == 0:
        return("BRAK")  # nie ma drogi dom-szkoła (z 1 do n)
    dist = [0] * (n + 1)
    # najdłuższe drogi
    for u in order:
        for (v, w) in G[u]:
            if value[v] < value[u]:  # jedziemy zgodnie z porządkiem topologicnym
                if dist[v] < dist[u] + w:
                    dist[v] = dist[u] + w  # można wydłużyć drogę
            else:
                # jest droga niezgodna z porządkiem, czyli cykl - koniec
                return("WAGARY")
    return(dist[n])

value = [0] * (n + 1)  # porządek topologiczny - docelowo - malejąco
value[0] = value[1] = 1
order = []     # skrzyżowania (osiągalne z domu (z 1)) w porządku topologicznym
DFS(1)
print(CalculateDist())
```

Sortowanie topologiczne przez przeliczanie stopni wejściowych (alg. Kahna) [4]:

Źródło: ./zadania/08_Grafy/Daleka_droga_do_szkoly/solutionKahna.py.

```
rd = lambda: map(int, input().strip().split())  # wczytywanie danych
# liczba skrzyżowań/dróg
n, m = rd()
G = [[] for i in range(n + 1)]
# wczytywanie grafu/planu miasta - para wierzchołek/waga
for i in range(m):
    u, v, w = rd()
    # ignorujemy drogi wychodzące ze szkoły (n) i wchodzące do domu (1)
    if (u != n) and (v != 1):
        G[u].append((v, w))

def BFS():
    # przechodzenie grafu wszerz, żeby ustalić wierzchołki/skrzyżowania osiągalne z 1
    q = [1]
    while len(q) > 0:
        u = q.pop(0)
        for (v, w) in G[u]:
            if v not in T:
                T.add(v)
                q.append(v)

def TopoSort():
    # algorytm Kahna z ograniczeniem na wierzchołki osiagalne z 1
    # zwraca 0 - OK (order zawiera porządek topologiczny)
    #        więcej niż 0 - jest cykl
    indegree = [0] * (n + 1)
    for u in T:
        for (v, w) in G[u]:
            indegree[v] += 1
    S = [1]     # wierzchołki osiągalne z 1 o zerowym indegree
    while len(S) > 0:
        u = S.pop()
        T.discard(u)
        order.append(u)
        for (v, w) in G[u]:
            indegree[v] -= 1   # zmniejszamy stopień wejściowy v
            if indegree[v] == 0:
                order.append(v)  # umieszczamy v w porządku topologicznym
                S.append(v)      # i wstawiany do kolejki wierzchołków do przetworzenia
    return len(T)

def CalculateDist():
    # najdłuższe drogi wyznaczanie dystansów i zwracanie wyniku
    dist = [0] * (n + 1)
    for u in order:
        for (v, w) in G[u]:
            if dist[v] < dist[u] + w:
                dist[v] = dist[u] + w  # można wydłużyć drogę
    return dist[n]
```

```
T = {1}     # zbiór wierzchołków osiągalnych z 1
BFS()
if n not in T:
    print("BRAK")
else:
    # porządek topologiczny
    order = [1]
    result = TopoSort()
    if result == 0:
        print(CalculateDist())
    else:
        print("WAGARY")
```

Rozwiązanie przez wyznaczenie najkrótszych ścieżek w grafie z ujemnymi wagami, alg. Bellmana-Forda [4]:

Źródło: ./zadania/08_Grafy/Daleka_droga_do_szkoly/solutionBF.py.

```
rd = lambda: map(int, input().strip().split())  # wczytywanie danych
# liczba skrzyżowań/dróg
n, m = rd()
G = [[] for i in range(n + 1)]
# wczytywanie grafu/planu miasta - para wierzchołek/waga
for i in range(m):
    u, v, w = rd()
    # ignorujemy drogi wychodzące ze szkoły (n) i wchodzące do domu (1)
    if (u != n) and (v != 1):
        G[u].append((v, w))

def CalculateDist():
    dist = [1] * (n + 1)  # nieosiagalne z 1
    dist[1] = 0
    # najdłuższe drogi wg Bellmana-Forda
    for i in range(n):
        corr = False
        for u in range(1, n+1):
            if dist[u] <= 0:  # osiągalne z 1/domu
                for (v, w) in G[u]:
                    if dist[v] > dist[u] - w:
                        dist[v] = dist[u] - w  # można wydłużyć drogę
                        corr = True     # jest poprawa
        if not corr:
            break
    if dist[n] > 0:
        return("BRAK")  # nie ma drogi dom->szkoła (1 do n)
    elif corr:
        return("WAGARY") # skoro ciągle się poprawiało to jest cykl
    else:
        return(-dist[n])

print(CalculateDist())
```

78.2.2 C++

Inny sposób sortowania topologicznego przez DFS (tutaj acykliczność sprawdzana jest w osobnej funkcji): **Źródło:** ./zadania/08_Grafy/Daleka_droga_do_szkoly/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include <set>
using namespace std;

void dfs(vector<vector<pair<int,int>>> &graph,
    set<int> &visited, set<int> &reachable, vector<int> &topological_order, int v) {
    visited.insert(v);
    reachable.insert(v);
    for (pair<int,int> neighbour : graph[v]) {
        if (visited.count(neighbour.first)==0) {
            dfs(graph, visited, reachable, topological_order, neighbour.first);
        }
    }
    topological_order.push_back(v);
}

bool dfs_cycle(vector<vector<pair<int,int>>> &graph, set<int> &visited, set<int> &path, int v) {
    visited.insert(v);
    path.insert(v);
    for (pair<int,int> neighbour : graph[v]) {
        if (visited.count(neighbour.first)==0) {
            if (dfs_cycle(graph, visited, path, neighbour.first)) {
                return true;
            }
        } else if (path.count(neighbour.first)==1) {
            return true;
        }
    }
    path.erase(v);
    return false;
}

int main() {
    ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
    int n, m;
    cin >>n>>m;

    vector<vector<pair<int,int>>> graph(n);

    for (int i=0; i<m; i++) {
        int a,b,c;
        cin >>a >>b >>c;
        if (a != n && b != 1) {
            graph[a-1].push_back(make_pair(b-1, c));
        }
```

```
    }

    set<int> visited;
    set<int> path;
    set<int> reachable;
    vector<int> topological_order;

    dfs(graph, visited, reachable, topological_order, 0);
    if (reachable.count(n-1)==0) {
        //nie ma drogi do szkoły
        cout <<"BRAK"<<endl;
        return 0;
    }

    visited.clear();
    if (dfs_cycle(graph, visited, path,0)) {
        //graf ma cykl, idziemy na wagary
        cout <<"WAGARY"<<endl;
        return 0;
    }

    // graf nie ma cyklu, jest DAGiem, szukam najdłuższych ścieżek programowaniem dynamicznym
    int *LP = new int[n];   //najdłuższe ścieżki od każdego węzła do szkoły
    for (int i=0; i<n; i++) {
        LP[i] = -20000000;  // bugfix - nieosiągające szkoły mają mieć < 0
    }
    LP[n-1] = 0;

    int i=0;
    while (topological_order[i]!=n-1) i++;
    i++;
    for (; i<topological_order.size(); i++) {
        int v = topological_order[i];
        for (pair<int,int> neighbour : graph[v]) {
            LP[v] = max(LP[v], neighbour.second + LP[neighbour.first]);
        }
    }

    cout <<LP[0]<<endl;
}
```

79 (VIII) – Król dzielni© – Gremliny

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gremliny (III edycja – liga algorytmiczna)***
*Język programowania: **C++/Python***

79.1 Treść zadania

Rok 3123 przyniósł wiele zmian w królestwie Bajtocji. Jedną z nich była gruntowna zmiana prawa administracyjnego. Tzw. „*Stary Ład*" został zastąpiony przez „Nowy Porządek". Nowa ustawa wymusiła podział każdego z miast wojewódzkich na dzielnice. Oczywiście pojęcie dzielnicy zostało w „Nowym Porządku" bardzo precyzyjnie zdefiniowane:

§13 ust. 37: *Przez dzielnicę rozumiemy fragment miasta, w którym:*

a) *dla dowolnych dwóch punktów dzielnicy jest możliwy przejazd pomiędzy nimi (w obydwie strony). Przez możliwość przejazdu należy rozumieć istnienie drogi, która w całości zawarta jest w danej dzielnicy.*
b) *dzielnica jest maksymalnym fragmentem miasta posiadającym własność a). Maksymalność rozumiemy w tym sensie, że rozszerzenie dzielnicy o kolejne punkty spowodowałoby naruszenie części a) powyższej definicji.*

Prezydent miasta stołecznego Bajtawy, pan Brajan Bajtkovsky, musi przedstawić na konferencji prasowej nowy podział administracyjny miasta. W tym celu poprosił o wsparcie swoich współpracowników. Pomóż prezydentowi wykonać stojące przed nim zadanie!

Zadanie

Prezydent Bajtkovsky przesłał Ci mailem plany administracyjne. Miasto stołeczne Bajtawa składa się z n punktów (gdzie $3 \leqslant n \leqslant 100\,000$), które połączone są przez $3 \leqslant k \leqslant 1\,000\,000$ jednokierunkowych uliczek. W tym miejscu należy podkreślić, że możliwe jest istnienie kilku równoległych uliczek, tj. biegnących od tego samego punktu a do tego samego punktu b. Twoim celem jest przygotowanie podziału administracyjnego Bajtawy, zgodnie z wymogami najnowszej ustawy.

Opis wejścia

W pierwszej linii wejścia podane są dwie liczby całkowite: n – liczba punktów, z których składa się mapa miasta oraz k – liczba ulic w Bajtawie. Każda z następnych k linii wejścia zawiera dwie liczby: $a, b \leqslant n - 1$. Owa para liczb opisuje jednokierunkową ulicę, wychodzącą z punktu o numerze a i wiodącą bezpośrednio do punktu o numerze b.

Opis wyjścia

W pierwszej linii wyjścia powinna pojawić się pojedyncza liczba całkowita d, opisująca liczbę dzielnic Bajtawy, zgodnie z przepisami „Nowego Porządku”. W następnych d liniach wyjścia powinny pojawić się opisy poszczególnych dzielnic.

Opis pojedynczej dzielnicy składa się z listy należących do niej punktów, ustawionych w kolejności malejącej. Lista dzielnic powinna być również uporządkowana malejąco względem pierwszego elementu ich opisu (tj. punktu dzielnicy o największym numerze).

Przykład 1

Dla przykładowego, podanego poniżej wejścia:

```
3 5
0 1
1 2
2 1
0 2
0 1
```

prawidłową odpowiedzią jest:

```
2
2 1
0
```

Przykład 2

Z kolei dla następującego wejścia:

```
7 8
0 1
1 2
2 0
3 4
4 5
5 6
3 6
3 5
```

prawidłową odpowiedzią jest:

```
5
6
5
4
3
2 1 0
```

Przykład 3

W ostatnim z przykładów, w którym wejście ma postać:

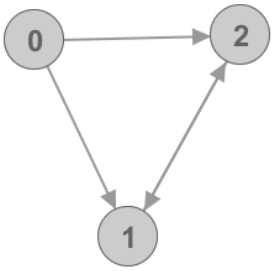
```
8 12
0 1
1 2
1 3
1 4
2 0
3 0
3 5
3 7
4 5
5 6
6 4
7 5
```

prawidłową odpowiedzią jest:

```
3
7
6 5 4
3 2 1 0
```

Wyjaśnienie przykładów

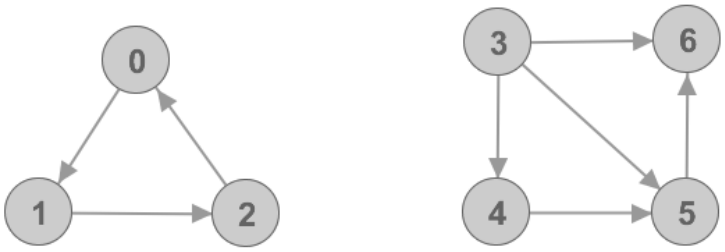
Naszkicujmy sobie rozpatrywane powyżej przypadki.



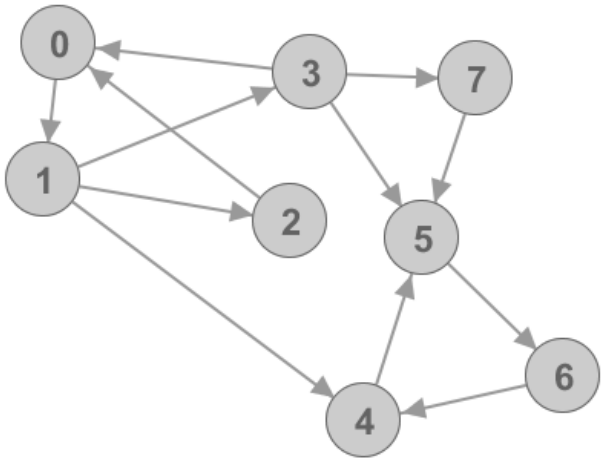
Rysunek 79.1: Schemat miasta, składający się z trzech punktów połączonych strzałkami opisany w pierwszym przykładzie

W pierwszym przykładzie mamy do czynienia z prostym schematem, w którym wierzchołki 1 i 2 są połączone w obydwie strony. Z kolei z wierzchołka 0 można dotrzeć do 1 i 2, ale nie można już w żaden sposób doń wrócić, co daje nam dwie dzielnice.

Schemat występujący w drugim przykładzie jest nieco bardziej złożony. Wyraźnie narzucają się nam dwie struktury. Pierwsza, która składa się z wierzchołków 0, 1 i 2 tworzy dzielnicę. Istotnie, z każdego z tych punktów daje się dotrzeć do dwóch pozostałych (niekoniecznie bezpośrednią trasą). W drugiej strukturze mamy jednak do czynienia z czterema jednopunktowymi dzielnicami. Istotnie, z wierzchołka 3 da się dotrzeć do każdego innego wierzchołka, ale nie da się poprowadzić trasy w drugim kierunku. Pozostałe wierzchołki również są pod tym względem problematyczne.



Rysunek 79.2: Schemat miasta, składający się z siedmiu punktów połączonych strzałkami opisany w drugim przykładzie



Rysunek 79.3: Schemat miasta, składający się z ośmiu punktów połączonych strzałkami opisany w trzecim przykładzie

W trzecim przykładzie mamy do czynienia z jeszcze bardziej złożoną siecią powiązań. Widzimy, że 0, 1, 2 oraz 0, 1, 3 tworzą dwa trójkątne „ronda”. Te cztery punkty tworzą zatem jedną dzielnicę. Kolejny tego typu trójkąt tworzą wierzchołki 4, 5, 6. Wierzchołka 7 nie da się przypisać do żadnej dotychczas utworzonej dzielnicy, więc łącznie mamy trzy dzielnice.

79.2 Rozwiązanie

Problematyka zadania sprowadza się do przetworzenia grafu skierowanego i znalezienia w nim *silnie spójnych składowych* [2][4]. Pojęcie to odnosi się do takich spójnych składowych grafu skierowanego, w których między każdą parą wierzchołków istnieje ścieżka[1] (co bezpośrednio odpowiada definicji „dzielnicy" z treści zadania). W poniższym omówieniu opiszemy krótko metodę znajdowania silnie spójnych składowych zaproponowaną przez S. Rao Kosaraju [15].

Algorytm sprowadza się do niewyrafinowanych operacji grafowych. W największym skrócie, dla danego wierzchołka będziemy używać algorytmu DFS (przeszukiwanie grafu *w głąb* – p. sekcja 71.12), aby znaleźć wszystkie osiągalne z niego wierzchołki, a następnie odwrócimy kierunek krawędzi (nazywamy to *transpozycją* grafu skierowanego) i wykorzystamy ponownie DFS, w celu identyfikacji wierzchołków, z których z kolei on jest osiągalny. Zauważmy tu, że już podczas wczytywania danych wejściowych, możemy dla każdego wierzchołka zapamiętać od razu zarówno te wierzchołki, do których da się z niego bezpośrednio sięgnąć, jak również te, z których krawędzie prowadzą do niego.

Zaczynamy od oznaczenia każdego z wierzchołków naszego grafu jako nieodwiedzonego. Przygotowujemy też pusty stos. Następnie przechodzimy przez wszystkie wierzchołki, dla każdego z nich sprawdzając, czy jest nieodwiedzony i – jeśli tak – oznaczamy go jako odwiedzony, a następnie rozpoczynamy od niego przeszukiwanie grafu w głąb. Po zakończeniu przeszukiwania DFS (podczas którego dla każdego napotkanego nieodwiedzonego wierzchołka robimy rekurencyjnie to samo) odkładamy wierzchołek na stosie i przechodzimy do kolejnego. Zauważmy, że po zakończeniu tej części, stos będzie zawierał wierzchołki w kolejności, w jakiej były odwiedzane (z wierzchołkiem od którego zaczęliśmy na wierzchu).

Po przejściu całego grafu w jedną stronę wykonujemy podobny przebieg *w drugą stronę*, tzn. po odwróceniu kierunku krawędzi (w tym miejscu przydaje nam się fakt, że zapamiętywaliśmy nie tylko krawędzie wychodzące z wierzchołka, ale również te, które do niego wchodzą). Tym razem jednak kolejność odwiedzania wierzchołków determinować będzie nasz – utworzony przy pierwszym przejściu – stos. Przywracamy wierzchołkom status *nieodwiedzonych* i kolejno zdejmujemy wierzchołki ze stosu. Jeżeli wierzchołek nie został jeszcze przez nas odwiedzony – wykonujemy na nim przeszukiwanie w głąb, uzyskując w ten sposób pojedynczą silnie spójną składową. W przeciwnym razie – wierzchołek już przynależy do jednej ze znalezionych składowych i można go pominąć.

Na koniec pozostaje kwestia odpowiedniej reprezentacji uzyskanych składowych. Tutaj można wykorzystać gotowe algorytmy sortujące – najpierw dla pojedynczych składowych, a potem (tworząc własny komparator) dla wszystkich składowych. Jak można zauważyć, złożoność asymptotyczna wynikająca z sortowania ($\mathcal{O}(n \log n)$) przeważa nad złożonością metody wyszukiwania silnie spójnych składowych (którą łatwo możemy oszacować na $\mathcal{O}(n)$).

[1] Zauważmy tu, że dla grafu nieskierowanego każda spójna składowa jest jednocześnie silnie spójną składową.

79.2.1 Python

Źródło: ./zadania/08_Grafy/Krol_dzielni_c/solution.py.

```
import sys
from queue import LifoQueue

def dfs1(v, graph, visited, postorder):
    stack = LifoQueue()
    stack.put([v,False])
    while not stack.empty():
        v,post = stack.get()
        if post:
            postorder.append(v)
        elif v not in visited:
            visited.add(v)
            stack.put([v, True])
            if v in graph:
                for neighbour in graph[v]:
                    if neighbour not in visited:
                        stack.put([neighbour, False])

def dfs2(v, graph, visited, components):
    stack = LifoQueue()
    stack.put(v)
    while not stack.empty():
        v = stack.get()
        visited.add(v)
        components[v] = [current_component]
        if v in graph:
            for neighbour in graph[v]:
                if neighbour not in visited:
                    stack.put(neighbour)

def reversed_graph(graph):
    p = {}
    for v in graph:
        p[v] = []
    for v, neighbours in graph.items():
        for to in neighbours:
            p[to].append(v)
    return p

def main():
    global current_component

    graph = {}

    n,m = [int(x) for x in input().split()]
    for i in range(0,n):
        graph[i] = []
```

```
    for i in range(0,m):
        a,b = [int(x) for x in input().split()]
        graph.setdefault(a, [])
        graph[a].append(b)


    visited = set()
    postorder = []
    r_graph = reversed_graph(graph)
    for v in r_graph:
        if v not in visited:
            dfs1(v, r_graph, visited, postorder)

    visited = set()
    components = {}
    current_component = 0
    for v in reversed(postorder):
        if v not in visited:
            dfs2(v, graph, visited, components)
            current_component += 1

    visited = set()
    print(current_component)
    rc = reversed_graph(components)
    for i in sorted([i for i in components], reverse=True):
        if i not in visited:
            print(*sorted(rc[components[i][0]], reverse=True))
            visited.update(rc[components[i][0]])

if __name__ == '__main__':
    main()
```

79.2.2 C++

Źródło: ./zadania/08_Grafy/Krol_dzielni_c/solution.cpp.

```
#include <bits/stdc++.h>

using namespace std;

struct Node {
  vector<int> outgoing = vector<int>();
  vector<int> incoming = vector<int>();
};

void dfs_forward(int x, vector<Node> &g, vector<bool> &visited, vector<int> &postorder) {
  stack<pair<int,bool>> S = stack<pair<int,bool>>();
  S.push(make_pair(x,false));
  while(not S.empty()){
    auto para = S.top();
```

```
    S.pop();
    x = para.first;
    if(para.second){
      postorder.push_back(x);
    }
        else{
      if(not visited[x]){
        visited[x] = true;
        S.push(make_pair(x, true));
        for(int i = 0; i < g[x].outgoing.size(); i++){
          if(not visited[g[x].outgoing[i]]){
            S.push(make_pair(g[x].outgoing[i], false));
          }
        }
      }
    }
  }
}

void dfs_backward(int x, vector<Node> &g, vector<bool> &visited, vector<vector<int>> &components){
  stack<int> S = stack<int>();
  S.push(x);
  components.push_back(vector<int>());
  while(not S.empty()){
    x = S.top();
    S.pop();
    if(!visited[x]){
      visited[x] = true;
          components[components.size()-1].push_back(x);
    for(int i = 0; i < g[x].incoming.size(); i++){
      if(not visited[g[x].incoming[i]]){
        S.push(g[x].incoming[i]);
        }
      }
    }
  }
}



vector <vector<int>> Kosaraju(vector<Node> &g, int n){
  vector<bool> visited = vector<bool>(n);
  vector<int> postorder = vector<int>();
  vector <vector<int>> components = vector<vector<int>>();
  for (int i = 0; i < n; i++){
    if (!visited[i]){
      dfs_forward(i,g,visited,postorder);
    }
  }
  for (int i = 0; i < n; i++){
    visited[i] = false;
  }
```

```
  int component_id = 0;
  reverse(postorder.begin(), postorder.end());
  for(int i = 0; i < n; i++){
    int j = postorder[i];
    if(!visited[j]){
      dfs_backward(j,g,visited,components);
    }
  }
  return components;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  int n, m;
  cin >> n >> m;
  vector<Node> g(n);


  int a, b;
  while (m--) {
    cin >> a >> b;
    g[a].outgoing.push_back(b);
    g[b].incoming.push_back(a);
  }

  vector <vector<int>> A = Kosaraju(g,n);

  cout<<A.size()<<'\n';
  for(int i = 0; i < A.size(); i++){
    sort(A[i].begin(),A[i].end(),greater<int>());
  }
  sort(A.begin(),A.end(),[](const std::vector<int>& a, const std::vector<int>& b) {
    return a[0] > b[0];
  });

  for(int i = 0; i < A.size(); i++){
    for(int j = 0; j < A[i].size(); j++){
      cout<<A[i][j]<<' ';
    }
    cout<<'\n';
  }
  return 0;
}
```

80 (VIII) – Squbbits – Gremliny

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gremliny (II edycja – zawody algorytmiczne – etap lokalny)***
*Język programowania: **C++/Python***

80.1 Treść zadania

Biten przypatrywał się wielokrotnie zmaganiom swojego młodszego brata w grze o jakże zacnym tytule Scrabbits^{TM} – planszówce polegającej na układaniu wyrazów z oznaczonych literami kwadratowych płytek. Zauważał, że jego brat spędza długie godziny nad obmyślaniem strategii i planowaniem idealnego doboru układanych słów w każdej sytuacji.

Biten nie mógł być gorszy od swojego brata, więc znalazł sobie własną grę planszową o nazwie Squbbits. Oczywiście wybrana przez niego rozrywka musiała być znacznie bardziej skomplikowana – zamiast kwadratowych płytek, Biten dysponował małymi, znaczonymi sześcianami. Na każdej ze ścianek znajdowała się pojedyncza litera z zakresu od A do Z. Zasady gry pozostawały niezmienione – z elementów, którymi operował gracz, trzeba było układać różne słowa.

Problem w tym, że skomplikowanie gry zniechęcało kolegów Bitena. Zgodzili się z nim zagrać tylko w sytuacji, gdy wprowadzi ich w świat rozgrywki, rozwiązując kilka postawionych mu Squbbitowych zagadek. Pomóż Bitenowi zachęcić swoich kolegów do wspólnej gry!

Zadanie

Biten, by zachęcić kolegów do wspólnej gry musi rozwiązać T postawionych mu zagadek. Każda zagadka ma następującą postać:

Podany jest zestaw k sześciennych kostek Squbbits. Następnie należy zweryfikować, czy wykorzystując podane elementy jest możliwe ułożenie z góry zadanego słowa. Jeśli nie jest możliwe ułożenie danego słowa, to pytanie przyjmuje formę „ile liter należy wykreślić, żeby dane słowo było możliwe do ułożenia z takich kostek?”.

Oczywiście udzielanie błyskawicznych odpowiedzi na takie pytania nie jest sprawą łatwą. Dlatego Biten potrzebuje Twojej pomocy!

Opis wejścia

Pierwszy wiersz standardowego wejścia składa się z pojedynczej liczby T ($1 \leq T \leq 10$), oznaczającej liczbę przypadków testowych. Dalej następuje T opisów zagadek. Pierwsza linia opisu zagadki składa się z pojedynczej liczby naturalnej n ($3 \leq n \leq 10\,000$), opisującej liczbę kostek Squbbits, którą Biten ma do dyspozycji. W drugiej linii testu znajduje się pojedyncze słowo, zapisane wielkimi literami alfabetu. Długość tego słowa nie przekracza $10\,000$ znaków. W kolejnych n liniach testu zawarte zostały opisy poszczególnych kostek.

Opis pojedynczej kostki składa się z sześciu oddzielonych spacjami wielkich liter alfabetu łacińskiego. Opisują

one litery zapisane na ściankach danej kostki. Pojedyncza litera może wystąpić na danej kości więcej niż jeden raz.

Opis wyjścia

Na wyjściu standardowym, dla każdej zagadki powinna być wypisana pojedyncza linia tekstu. W przypadku, gdy możliwe jest ułożenie słowa z danych kostek Squbbits, należy wypisać pojedyncze słowo „TAK”.

Gdy nie jest to możliwe, należy wypisać najmniejszą liczbę liter, które należy wykreślić z wyrazu tak, aby możliwym było ułożenie go z podanego zestawu elementów.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
2
5
ALINA
J N I L I S
A E P E K S
B A B I B O
Z K L A S A
N O Z Y C E
5
KARKI
K O N I K I
A R A L I A
A L I N I A
A B C D E F
R O B O T A
```

prawidłową odpowiedzią jest:

```
TAK
1
```

Wyjaśnienie przykładu

W pierwszym przypadku testowym łatwo zauważyć, że wyraz `ALINA` da się ułożyć ustawiając kostki w następującej kolejności:

- Druga kość – litera `A`;
- Pierwsza kość – litera `L`;
- Trzecia kość – litera `I`;
- Piąta kość – litera `N`;
- Czwarta kość – litera `A`.

Oczywiście nie jest to jedyny układ klocków, który wygeneruje słowo `ALINA`. Naszym celem jest jednak jedynie zweryfikowanie, czy słowo to może zostać ułożone.

W drugim przypadku nie jest możliwe ułożenie wyrazu `KARKI` – do tego potrzebne byłyby co najmniej dwie kości, na których znajdowałaby się litera `K`. Niemniej jednak, po wykreśleniu pierwszej litery `K`, możemy ułożyć z dostępnych kostek wyraz `ARKI`, np. wykorzystując do tego kolejno czwarty, piąty, pierwszy i trzeci sześcian.

80.2 Rozwiązanie

W zadaniu mamy podany zestaw sześciennych klocków z literami i słowo określonej długości. Naszym celem jest zweryfikowanie, czy możliwe jest utworzenie zadanego słowa z podanych klocków, przy czym każdy element może być wykorzystany tylko raz.

Optymalne rozwiązanie problemu zakłada skonstruowanie grafu dwudzielnego, którego jedną część stanowią litery budowanego słowa, drugą – kostki z literami, zaś krawędź pomiędzy kostką i literą jest rysowana w sytuacji, w której dana litera budowanego słowa znajduje się na wskazanej kostce.

Następnie na tak utworzonym grafie budowane jest maksymalne skojarzenie – jeśli jego liczność jest równa długości słowa, to można je zbudować z danego zestawu klocków. Wykorzystać można w tym celu podejście oparte na metodzie Forda-Fulkersona[1], np. algorytm Edmondsa-Karpa [4][23] lub zbliżony do niego algorytm Dinitza [17][23] (w naszym rozwiązaniu przykładowym stosujemy jeszcze inny algorytm: Hopcrofta-Karpa [17][23], który można uznać za pewien przypadek szczególny algorytmu Dinitza).

Do rozwiązania tego zadania można również zastosować podejście heurystyczne, zakładające zliczanie wystąpień kostek z poszczególnymi literami i próbujące na podstawie tej informacji odpowiedzieć, czy jest możliwe utworzenie podanego wyrazu. Podejście to nie okaże się jednak zapewne wystarczająco szybkie, aby zagwarantować zdobycie kompletu punktów za to zadanie, podobnie jak próby konstrukcji zadanego słowa metodami siłowymi lub za pomocą programowania dynamicznego.

[1] Metodę tę przedstawiliśmy we wprowadzeniu do tego działu.

80.2.1 Python

Źródło: ./zadania/08_Grafy/Squbbits/solution.py.

```
import collections
import sys

INFINITY= 2000000000
NIL = 0

class BipartiteGraph:
    U,V =0,0
    Adj = [[]]
    Dist,PairU,PairV = [],[],[]
    Q = collections.deque([])

    def __init__(self,u,v):
        self.U=u+1
        self.V=v+1
        self.Adj = [[] for x in range(self.U)]
        self.Dist = [0 for i in range(self.U)]
        self.PairU,self.PairV = [NIL for i in range(self.U)], [NIL for i in range(self.V)]

    def bfs(self):
        for u in range(1,self.U):
            if self.PairU[u] == NIL:
                self.Dist[u] = 0
                self.Q.append(u)
            else:
                self.Dist[u] = INFINITY
        self.Dist[NIL] = INFINITY

        while self.Q:
            u = self.Q.popleft()
            if u > NIL:
                for v in self.Adj[u]:
                    if self.Dist[self.PairV[v]] == INFINITY:
                        self.Dist[self.PairV[v]] = self.Dist[u] + 1
                        self.Q.append(self.PairV[v])
        return self.Dist[NIL] < INFINITY

    def dfs(self,u):
        if u!=NIL:
            for v in self.Adj[u]:
                if self.Dist[self.PairV[v]] == self.Dist[u] + 1:
                    if self.dfs(self.PairV[v]):
                        self.PairV[v] = u
                        self.PairU[u] = v
                        return True
            self.Dist[u] = INFINITY
            return False
        else:
```

```
            return True

    def addEdge(self,u,v):
        self.Adj[u].append(v)

    def HopcroftKarp(self):
        matchingSize = 0
        while self.bfs():
            for u in range(1,self.U):
                if self.PairU[u]==NIL:
                    if self.dfs(u):
                        matchingSize+=1
        return matchingSize

def getDice():
    return set(input().split(' '))

def test():
    u = int(input())
    s = input()
    LetterEdges = [[] for x in range(26)]
    BG = BipartiteGraph(len(s),u)
    for i in range(u):
        A = getDice()
        for a in A:
            LetterEdges[ord(a) - ord('A')].append(i+1)
    for i in range(len(s)):
        chara = int(ord(s[i]) - ord('A'))
        for dicenumber in LetterEdges[chara]:
            BG.addEdge(i+1,dicenumber)
    result = len(s)-BG.HopcroftKarp()
    if result>0:
        print(result)
    else:
        print("TAK")


Tau = int(input())
for testnumber in range(Tau):
    test()
```

80.2.2 C++

Źródło: ./zadania/08_Grafy/Squbbits/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <climits>

#define INF INT_MAX
//null vertex
#define NIL 0

using namespace std;

class BipartiteGraph {
public:
  int U, V; //number of points in each part of the graph
  vector<vector<int> > Adj; //adjacency list
  vector<int> Pair_U;
  vector<int> Pair_V;
  vector<int> Dist;
  queue<int> Q;

public:
  BipartiteGraph(int u, int v) {
    U = u + 1;
    V = v + 1;
    Adj = vector<vector <int> >(U, vector<int>(0));
    Pair_U = vector<int>(U, NIL);
    Pair_V = vector<int>(V, NIL);
    Dist = vector<int>(U);
    Q = queue<int>();
  }

  bool bfs() {
    for (int u = 1; u < U; u++) {
      if (Pair_U[u] == NIL) {
        Dist[u] = 0;
        Q.push(u);
      }
      else {
        Dist[u] = INF;
      }
    }
    Dist[NIL] = INF;
    while (not Q.empty()) {
      int u = Q.front();
      Q.pop();
      if (u!=NIL) {
        for (int i = 0; i < Adj[u].size(); i++) {
```

```
          int v = Adj[u][i];
          if (Dist[Pair_V[v]] == INF) {
            Dist[Pair_V[v]] = Dist[u] + 1;
            Q.push(Pair_V[v]);
          }
        }
      }
    }
    return (Dist[NIL] < INF);
  }

  bool dfs(int u) {
    if (u != NIL) {
      for (int i = 0; i < Adj[u].size(); i++) {
        int v = Adj[u][i];
        if (Dist[Pair_V[v]] == Dist[u] + 1) {
          if (dfs(Pair_V[v])) {
            Pair_V[v] = u;
            Pair_U[u] = v;
            return true;
          }
        }
      }
      Dist[u] = INF;
      return false;
    }
    return true;
  }


  int HopcroftuKarpu() {
    int matchingu = 0;
    while (bfs()) {
      for (int u = 1; u < U; u++) {
        if (Pair_U[u]==NIL && dfs(u)) {
          matchingu = matchingu + 1;
        }
      }
    }
    return matchingu;
  }

  void AddEdge(int u, int v) {
    Adj[u].push_back(v);
  }
};


void test(){
  int u;
  string s;
  cin>>u>>s;
```

```
  vector<vector<int> > LetterEdges = vector<vector<int> >(26,vector<int>());
  BipartiteGraph BG = BipartiteGraph(s.size(),u);

  set<int> dice = set<int>();
  char side;
  for(int i =1; i<=u; i++){
    for(int sideno=0; sideno<6; sideno++){
      cin>>side;
      dice.insert(side-'A');
    }
    for(set<int>::iterator it = dice.begin(); it!=dice.end(); it++){
      LetterEdges[*it].push_back(i);
    }
    dice.clear();
  }
  int chara;
  for(int i = 0; i <s.size(); i++){
    chara = s[i] - 'A';
    for(int j =0; j<LetterEdges[chara].size(); j++){
      BG.AddEdge(i+1,LetterEdges[chara][j]);
    }
  }
  int A = BG.HopcroftuKarpu();
  if(A==s.size()){
    cout<<"TAK\n";
  }
  else{
    cout<<s.size()-A<<'\n';
  }
}

int main(){
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int T;
  cin>>T;
  while(T-->0){
    test();
  }
  return 0;
}
```

81 (VIII) – Jak przechytrzyć smoka? – Gremliny

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gremliny (IV edycja – zawody algorytmiczne – etap lokalny)***
*Język programowania: **C++/Python***

81.1 Treść zadania

Rządy króla Bajtorego były jednym z najspokojniejszych okresów w burzliwej historii królestwa Bajtocji. Były, ponieważ ten szczęśliwy okres beztroski dobiegł końca z momentem, w którym w górach Bajpatach zamieszkał potężny smok o imieniu Bajtryk. Bestia ta nie tylko pożerała stada bajtockich baranów, ale – przede wszystkim – piła niesamowite ilości wody ze stołecznego akweduktu.

Oczywiście nadmiar wody jest, jak wiadomo, zabójczy dla smoków, ale sprytny Bajtryk zniszczył jedną z odnóg konstrukcji, dostosowując w ten sposób ilość wody do swoich potrzeb. Dzięki temu mógł odtąd komfortowo ustawić się tuż przy wylocie sieci akweduktowej i pić całą wodę spływającą do miasta. Nad mieszkańcami stolicy zawisła więc groźba śmierci z pragnienia, a także z głodu, bo działania smoka uniemożliwiły nawadnianie pól uprawnych.

W takich to dramatycznych okolicznościach, do króla Bajtorego zgłosił się śmiałek – górnik-budowlaniec o imieniu Sebajtstian i zobowiązał się rozwiązać smoczy problem.

— *Jak zamierzasz pozbyć się bestii?* — zapytał go król.

— *Bardzo prosto, Wasza Wysokość!* — odparł na to Sebajtstian.
— *Wystarczy, że naprawimy uszkodzony akwedukt, przywracając go do dawnej świetności, a podły smok, gdy podstawi swój paskudny pysk by znowu pić naszą wodę... wypełni się jak balonik i pęknie!*

— *Odbudować akwedukt do dawnej świetności, hmm... A ile to będzie kosztować królewski skarbiec?*— zaniepokoił się władca.

W tym miejscu rozmowa zmieniła się w iście rynkową awanturę, albowiem król Bajtory bynajmniej nie słynął z hojności. Po długich negocjacjach zgodził się zapłacić za największy ***sensowny*** rurociąg, który zastąpi zniszczony przez Bajtryka odcinek.

Rozmiar (a więc i przepustowość) rurociągu król uznaje za sensowną, jeżeli może być w pełni wykorzystana. Innymi słowy, jeżeli dalsze powiększanie rurociągu nie zwiększa ilości wody dostarczanej do miasta, to przestaje ono mieć sens. Aby pokonać smoka trzeba więc oszacować maksymalny rozmiar odbudowywanego odcinka akweduktu, za który Bajtory jest skłonny zapłacić.

Zadanie

Bajtory udostępnił dla Ciebie i Sebajtstiana plany sieci akweduktów, na których zaznaczone są przepustowości każdego odcinka sieci rurociągów naziemnych. Zniszczony przez Bajtryka kawałek konstrukcji, który ma zostać odbudowany, został na nich zakreślony kółkiem. Twoim zadaniem jest podanie największej możliwej

przepustowości rekonstruowanego odcinka, spełniającej warunki narzucone przez króla. Ponadto, należy obliczyć ile wody przepływa przez sieć akweduktową obecnie, przed dokonaniem naprawy. Te informacje pozwolą Sebajtstianowi oszacować, ile czasu trzeba będzie czekać aż smok zdechnie.

Opis wejścia

W pierwszej linii wejścia otrzymasz trzy liczby naturalne:

- $5 \leqslant n \leqslant 1\,000$, opisującą liczbę punktów węzłowych akweduktu (poszczególne odcinki akweduktu mogą się łączyć ze sobą w tych punktach – i tylko tam);
- $5 \leqslant m \leqslant 10\,000$, która odpowiada liczbie odcinków akweduktu, z których składa się sieć (łącznie z odcinkiem zniszczonym przez Bajtryka);
- $1 \leqslant s \leqslant n - 1$, opisującą liczbę źródeł wody (na potrzeby zadania można przyjąć, że źródło wody jest w stanie wygenerować dowolną jej ilość). Dla ułatwienia przyjmować będziemy, że źródła wody zlokalizowane są w punktach węzłowych o numerach od 1 do s.

W kolejnych $m - 1$ wierszach znajdziesz po trzy liczby naturalne a, b, c, opisujące kolejne $m - 1$ odcinków istniejącej sieci. Liczby a i b przyjmują wartości z zakresu od 1 do n i oznaczają numery punktów węzłowych połączonych danym odcinkiem (przy czym woda płynie w nim zawsze tylko w jednym kierunku, tj. od punktu a do punktu b). Liczba c nie przekracza 10^6 i oznacza przepustowość danego odcinka (wyrażoną w m^3/min – metrach sześciennych na minutę).

Ponadto sieć posiada tylko jedno ujście, przy którym czatuje smok – jest to punkt węzłowy o największym numerze, tj. n.

Ostatni wiersz zawiera tylko dwie liczby a, b, opisujące odbudowywany odcinek sieci wodnej.

Opis wyjścia

Na wyjściu powinny pojawić się dwie liczby całkowite: $\overline{A}$ oraz $\overline{S}$. Pierwsza z tych liczb oznacza przepustowość sieci wodnej przed odbudową brakującego odcinka, a druga – przepustowość odbudowywanego odcinka. Jeżeli z jakiegoś powodu nie jest możliwe, żeby królewski skarbiec sfinansował odbudowę akweduktu (np. w sytuacji, gdy można bez końca powiększać przepływ sieci wodnej), należy wypisać na wyjście wielkimi literami smutny komunikat (bez polskich znaków diakrytycznych) o treści:

```
BAJTRYK ZWYCIEZYL HAHAHAHA
```

Przykład

Dla przykładowego wejścia

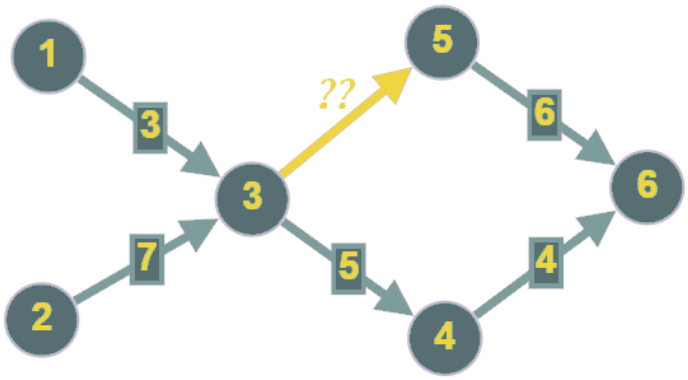
```
6 6 2
1 3 3
2 3 7
3 4 5
4 6 4
5 6 6
3 5
```

poprawnym wyjściem jest

```
4 6
```

Wyjaśnienie przykładu

Przyjrzyjmy się jak wygląda nasza sieć akweduktów:



Rysunek 81.1: Schemat sieci akweduktów, składający się z sześciu punktów i sześciu krawędzi opisanych w przykładzie

Zauważmy, że ze źródeł (węzły 1, 2) może łącznie wypłynąć maksymalnie $10\,m^3/min$.

Jednak dalej woda może płynąć tylko trasą $3 \to 4 \to 6$, która jest w stanie transportować do węzła końcowego (6) maksymalnie $4m^3/min$ (i to jest pierwsza wartość, która powinna pojawić się na wyjściu). Możemy jednak poprawić sytuację, odbudowując zniszczony przez Bajtryka odcinek między węzłem 3 i 5, co zapewni możliwość transportu wody również trasą $3 \to 5 \to 6$. Optymalną przepustowością odbudowanego odcinka akweduktu jest 6 (i to jest druga wartość, która powinna pojawić się na wyjściu). Zauważmy, że w takim przypadku do węzła końcowego będzie wpływało sumarycznie $10\,m^3/min$.

81.2 Rozwiązanie

Problematyka zadania sprowadza się do znalezienia przepustowości *sieci przepływowej* zdefiniowanej początkowo, a następnie zweryfikowanie jak zmieni się ona po dołożeniu dodatkowej krawędzi (o dowolnie dużej przepustowości, ale określonym początku i końcu).

Wprowadźmy oznaczenia pomocnicze – niech G oznacza początkową sieć przepływową opisaną w zadaniu, zaś u, v niech będą odpowiednio początkiem i końcem odbudowywanej krawędzi. Pomysł na rozwiązanie przypadku bazowego jest bardzo prosty i sprowadza się do trzech punktów:

1. Znajdujemy maksymalny przepływ w grafie G bez krawędzi (u, v). Niech F oznacza graf wykorzystanych przepływów (tj. krawędź (a, b) w grafie F odpowiada przepływowi z punktu a do b w grafie G przy realizacji znalezionego maksymalnego przepływu).
2. Modyfikujemy graf G, dodając do niego krawędź (u, v) o nieskończonej przepustowości.
3. Na podstawie zmodyfikowanego grafu G konstruujemy graf G', w którym przepustowości poszczególnych krawędzi są różnicą przepustowości początkowych i przepływów z F, a następnie liczymy maksymalny przepływ w G'. Otrzymany wynik jest największą sensowną przepustowością krawędzi (u, v).

Samo znajdowanie maksymalnego przepływu w grafie można zrealizować za pomocą jednego z kilku znanych algorytmów, takich jak algorytm Dinitza [17][23] lub algorytm Edmondsa-Karpa [4][23] (stanowiący implementację metody Forda-Fulkersona – p. wprowadzenie do tego działu). Podane poniżej rozwiązanie przykładowe jest oparte na macierzowej reprezentacji grafu i wykorzystuje algorytm Edmondsa-Karpa.

Zastanówmy się jeszcze, kiedy może dojść do sytuacji, w której trzeba będzie wypisać na standardowe wyjście komunikat o zwycięstwie smoka? Do takiej sytuacji może dojść wtedy, gdy powiększanie krawędzi (u, v) zwiększa maksymalny przepływ w nieograniczony sposób. Oznacza to, że $v = n$ (w przeciwnym wypadku ścieżki wychodzące z v musiałyby łącznie mieć nieskończoną przepustowość, a to jest wykluczone w treści zadania). Skupmy się teraz na wierzchołku u. Łatwo zauważyć, że zwiększanie przepływu w nieskończoność nastąpić może tylko wtedy, gdy u jest wierzchołkiem źródłowym. Do zwykłego wierzchołka wchodzi bowiem skończona liczba krawędzi o skończonej przepustowości – w pewnym momencie wyczerpalibyśmy cały przepływ, który mógł wyjść z wierzchołka u. Tak więc sytuacją, którą należy wychwycić jest $u \leq s$ oraz $t = n$.

81.2.1 Python

Źródło: ./zadania/08_Grafy/Jak_przechytrzyc_smoka/solution.py.

```
INFTY = 100000000
#Znalezienie ścieżek z użyciem algorytmu BFS
def bfs(C, F, s, t):
        kolejka = [s]
        sciezki = {s:[]}
        if s == t:
            return sciezki[s]
        while kolejka:
            u = kolejka.pop(0)
            for v in range(len(C)):
                    if(C[u][v]-F[u][v]>0) and v not in sciezki:
                        sciezki[v] = sciezki[u] + [(u,v)]
                        if v == t:
                            return sciezki[v]
                        kolejka.append(v)
        return None

#Algorytm Forda-Fulkersona w wersji Edmondsa-Karpa
def Edmonds_Karp(C, s, F):
        n = len(C)
        t = n-1
        sciezka = bfs(C, F, s, t)
        while sciezka != None:
            przeplyw = min(C[u][v] - F[u][v] for u,v in sciezka)
            for u,v in sciezka:
                F[u][v] += przeplyw
                F[v][u] -= przeplyw
            sciezka = bfs(C, F, s, t)
        return sum(F[s][i] for i in range(n))

def read_graph(n,m,s):
    C = [[0 for qqq in range(n+1)] for _ in range(n+1)]
    for i in range(m-1):
        edge = list(map(int, input().split()))
#        print(edge)
        C[edge[0]][edge[1]] = edge[2]
    for i in range(1,s+1):
        C[0][i] = INFTY
    return C

DEBUG = False
DEBUG2 = False

#Wczytujemy parametry zadania
config = list(map(int, input().split()))
#Wczytujemy graf do postaci macierzowej
C = read_graph(config[0],config[1],config[2])
```

```
#Początkowe wartości przepływów między wierzchołkami ustawiamy na zero
F = [[0] * (config[0]+1) for i in range(config[0]+1)]
#Wczytujemy interesującą nas krawędź:
missing_edge = list(map(int, input().split()))
#Jeżeli możemy zwiększać w nieskończoność krawędź łączącą któreś ze źródeł z wyjściem
# to mamy problem...
if(missing_edge[0]<=config[2] and missing_edge[1] == config[0]):
    print("BAJTRYK ZWYCIEZYL HAHAHAHA")
else:
    #obliczamy max_flow bez brakującej krawędzi
    initial_flow = Edmonds_Karp(C,0,F)
    #obliczamy max_flow z brakującą krawędzią (pozwalamy jej na dowolnie duży przepływ)
    C[missing_edge[0]][missing_edge[1]] = INFTY
    after_flow = Edmonds_Karp(C, 0, F)
    if(DEBUG):
        for row in C:
            print(row)
    if(F[missing_edge[0]][missing_edge[1]] != after_flow-initial_flow):
        print("MAMY PROBLEM")
    print(initial_flow, after_flow-initial_flow, sep=' ')
```

81.2.2 C++

Źródło: ./zadania/08_Grafy/Jak_przechytrzyc_smoka/solution.cpp.

```cpp
#include <bits/stdc++.h>

using namespace std;

#define INFTY 100000000

struct Graph{
  vector<vector<int>> graph;

  int size(){
    return graph.size();
  }

  void print(){
    for(const auto& value: graph){
      for(const auto& i: value){
        cout<<i<<" ";
      }
    cout<<endl;
    }
  return;
  }

};

void print(vector<pair<int,int>> D){
  for(const auto& value: D){
    cout<<"["<<value.first<<", "<<value.second<<"], ";
  }
  cout<<endl;
}

//Znalezienie ścieżek z użyciem algorytmu BFS
vector<pair<int,int>> bfs(Graph &C, Graph &F, int s,int t){
  int n = C.size();
  queue<int> kolejka = queue<int>();
  kolejka.push(s);
  vector<vector<pair<int,int>>> sciezki = vector<vector<pair<int,int>>>(n,vector<pair<int,int>>());
  if(s == t){
    return sciezki[s];
  }
  else{
    while(not kolejka.empty()){
      int u = kolejka.front();
      kolejka.pop();
      for(int v=1; v<n;v++){
          if(C.graph[u][v]-F.graph[u][v]>0 and sciezki[v].size()==0){
            sciezki[v] = sciezki[u];
```

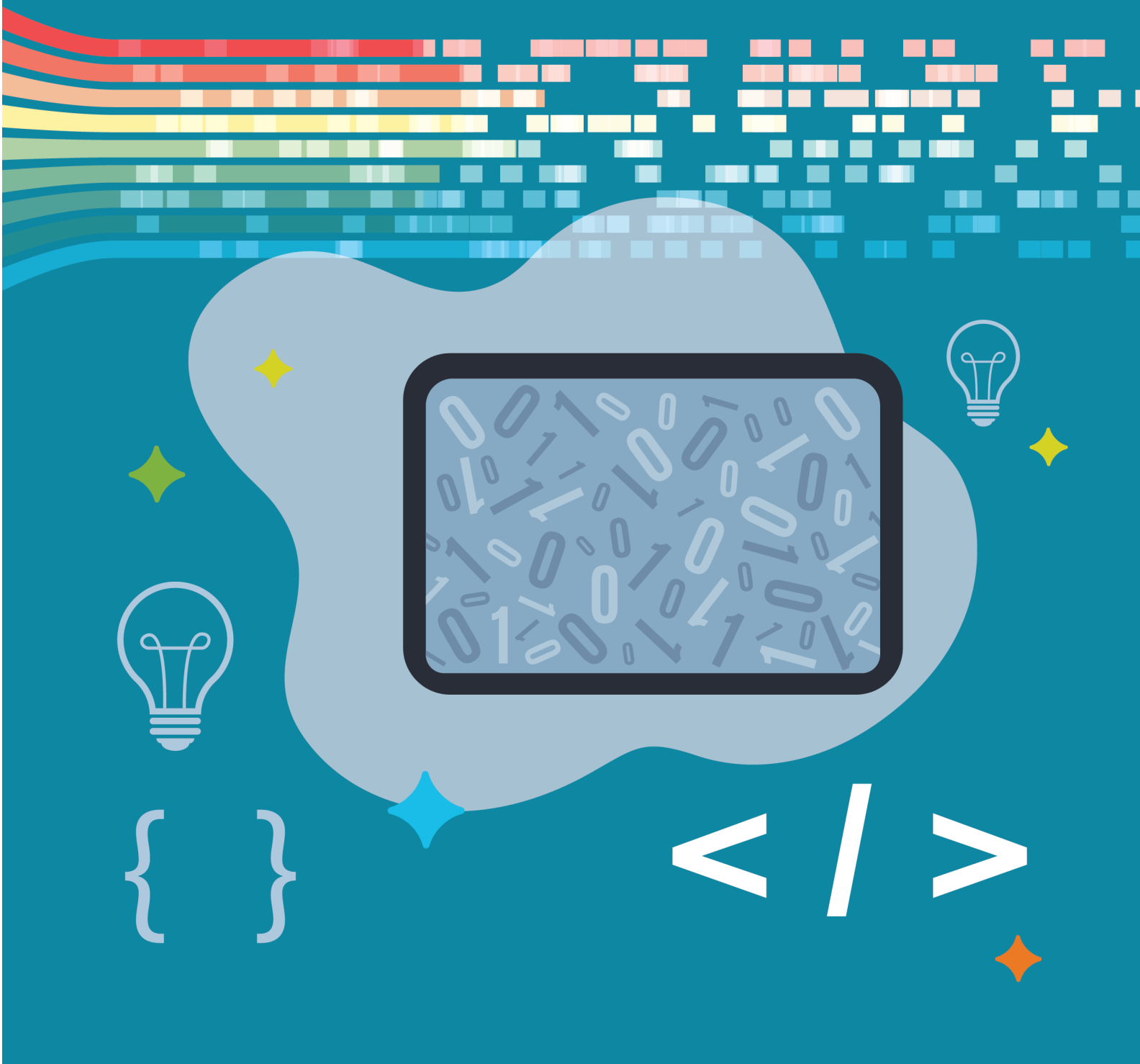
```
            sciezki[v].push_back(make_pair(u,v));
            /*
            for(const auto& value: sciezki){
              print(value);
            }
            */
            if(v == t){
              return sciezki[v];
            }
            kolejka.push(v);
          }
        }
      }
  }
  return vector<pair<int,int>>();
}

int minuv(Graph &C, Graph& F, vector<pair<int,int>> P){
  int X = INFTY;
  for(const auto& value: P){
    X = min(X, C.graph[value.first][value.second] - F.graph[value.first][value.second]);
  }
  return X;
}

//Algorytm Forda-Fulkersona w wersji Edmondsa-Karpa
int Edmonds_Karp(Graph &C, int s, Graph &F){
        int n = C.size();
        int t = n-1;
        vector<pair<int,int>> sciezka = bfs(C, F, s, t);
        while(sciezka.size()>0){
//      cout<<"Found\n";
            int przeplyw = minuv(C,F,sciezka);
      for(const auto& value: sciezka){
        F.graph[value.first][value.second] += przeplyw;
        F.graph[value.second][value.first] -= przeplyw;
      }
            sciezka = bfs(C, F, s, t);
    }
    int X = 0;
    for(int i = 0; i <n; i++){
      X+=F.graph[0][i];
    }

  return X;
}

Graph read_graph(int n,int m,int s){
    Graph C = Graph();
  C.graph = vector<vector<int>>(n+1,vector<int>(n+1));
  int x,y,z;
    for(int i = 1; i<m; i++){
```

```
    cin>>x>>y>>z;
        C.graph[x][y] = z;
  }
    for(int i = 1; i<s+1; i++){
        C.graph[0][i] = INFTY;
  }
    return C;
  }

int main(){
    int n,m,s;
    cin >> n >> m >> s;
  //Wczytujemy graf do postaci macierzowej

    Graph C = read_graph(n,m,s);

  Graph F = Graph();
  F.graph = vector<vector<int>>(n+1,vector<int>(n+1));
  //Wczytujemy interesującą nas krawędź:
  int x,y;
  cin>>x>>y;

  //Jeżeli możemy zwiększać w nieskończoność krawędź łączącą któreś ze źródeł z wyjściem
  //to mamy problem...
  if(x<=s and y == n){
    cout<<"BAJTRYK ZWYCIEZYL HAHAHAHA\n";
    return 0;
  }
  //obliczamy max_flow bez brakującej krawędzi
  int initial_flow = Edmonds_Karp(C,0,F);
  //obliczamy max_flow z brakującą krawędzią (pozwalamy jej na dowolnie duży przepływ)
  C.graph[x][y] = INFTY;
  int after_flow = Edmonds_Karp(C, 0, F);

  if(F.graph[x][y] != after_flow-initial_flow){
    cout<<"MAMY PROBLEM\n";
  }
  cout<<initial_flow<<' '<<after_flow-initial_flow<<'\n';
    return 0;
}
```

Centrum Mistrzostwa Informatycznego

Dział IX

Programowanie dynamiczne



82 (IX) – Wprowadzenie

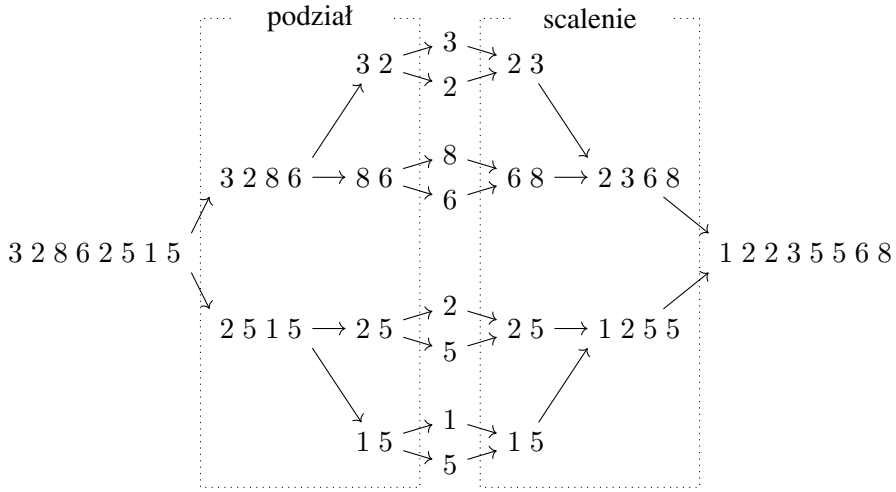
Autor: Andrzej Jastrzębski, Politechnika Gdańska

Programowanie dynamiczne jest bardzo efektywną techniką rozwiązywania niektórych problemów algorytmicznych [2][4]. Zanim jednak o niej powiemy, przypomnijmy na czym polega inna, również bardzo skuteczna i chętnie stosowana metoda konstrukcji algorytmów, bazująca na:

- podziale danych wejściowych na kilka mniejszych zestawów danych;
- rozwiązaniu podproblemów związanych z tymi mniejszymi zestawami;
- scaleniu wyników[1].

Mowa oczywiście o klasycznej technice „dziel i zwyciężaj” (ang. *divide and conquer*) [2][4][7].

Jako przykład użycia tej techniki możemy rozważyć jeden ze znanych algorytmów sortowania danych (p. dział V) – *sortowanie przez scalanie*. W tym algorytmie faza podziału danych wejściowych jest trywialna, tj. dzielimy tablicę w połowie na dwie podtablice o podobnych rozmiarach. Następnie te podtablice niezależnie sortujemy. Ostatnia faza – scalanie – jest realizowana poprzez odpowiednie złączenie dwóch posortowanych tablic w jedną. Na rysunku 82.1 jest naszkicowany przykład sortowania przez scalanie ciągu ośmiu liczb, z wyróżnionymi fazami podziału i scalania.



Rysunek 82.1: Przykład sortowania przez scalanie

Technika „dziel i zwyciężaj” charakteryzuje się tym, że podproblemy są niezależne od siebie, tj. do rozwiązania jednego podproblemu niepotrzebne są wyniki innych podproblemów. *Programowanie dynamiczne* jest podobne, z tym jednym wyjątkiem, że rozwiązania podproblemów mogą być zależne od siebie.

Ta zależność narzuca zwykle pewną określoną kolejność rozwiązywania podproblemów. Typowe dla programowania dynamicznego jest zapisywanie kolejnych wyników w tablicy[2], do której sięgamy, aby wykorzystać

[1] Podproblemy powinny być podobne do bazowego problemu – w idealnym przypadku jest to ten sam problem, tylko „mniejszy” (dla mniejszego zestawu danych).

[2] Można też korzystać z innych kontenerów takich jak mapa czy słownik.

rozwiązania podproblemów otrzymane wcześniej (w celu rozwiązania bieżącego). Zauważmy, że jest to możliwe tylko w przypadku takich problemów, których rozwiązanie optymalne (najlepsze, najszybsze,...) da się „złożyć" z optymalnych (najlepszych, najszybszych, ...) rozwiązań podproblemów. Znalezienie tej metody „składania" – zwykle w postaci pewnej funkcji lub reguły – stanowi tu główne wyzwanie.

Jednym z najprostszych problemów, które można rozwiązać za pomocą programowania dynamicznego jest wyznaczanie wartości ciągu Fibonacciego.

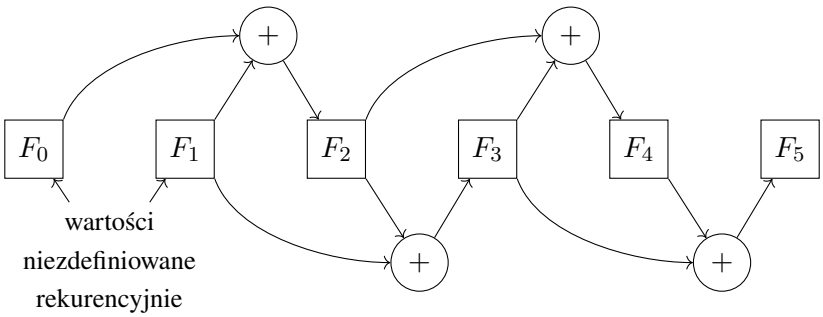
Ciąg Fibonacciego [7] jest zdefiniowany rekurencyjnie jako:

$$\begin{aligned} F_0 &= 0\,; \\ F_1 &= 1\,; \\ F_{n+2} &= F_{n+1} + F_n, \text{dla } n \geq 0\,. \end{aligned}$$

Zgodnie z definicją, kolejnymi wartościami ciągu Fibonacciego są liczby:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, ...$$

Aby obliczyć wartość F_k dla pewnego k, musimy obliczyć dwie wcześniejsze wartości ciągu: F_{k-1} oraz F_{k-2}. Można zatem powiedzieć, że w tym miejscu dzielimy nasz problem na dwa podproblemy, a funkcją „składającą" bieżące rozwiązanie z poprzednich jest zwykła suma: F_{k-1} + F_{k-2}.



Zauważmy jednak, że aby wyznaczyć z kolei F_{k-1}, potrzebujemy między innymi wartości F_{k-2}, która była także wymagana do obliczenia F_k. Musimy zatem tak skonstruować nasz algorytm, aby nie powielać niepotrzebnie obliczeń. Zobaczmy, jak można poradzić sobie z tym problemem na przykładzie trzech poniższych implementacji.

Wersja pierwsza – wyznaczenie wszystkich wartości i zapisanie ich do tablicy:

```
k = int(input())
f = [0,1]
for _ in range(k-1):
    f.append(f[-1]+f[-2]) # suma dwóch ostatnich wartości z tablicy
print(f[k])
```

Druga implementacja jest podobna, ale zapamiętujemy tylko najważniejsze dane tj. dwie ostatnie wartości (zauważmy, że nie potrzebujemy tu w ogóle tablicy):

```
k = int(input())
if k==0: print(0)
elif k==1: print(1) # bez tej linii też zadziała
```

```
else:
    a,b = 1,0
    for _ in range(k-1):
        a,b = a+b,a
    print(a)
```

W trzecim rozwiązaniu wykorzystamy funkcję rekurencyjną wraz z zapamiętywaniem wartości pośrednich:

```
1 rozw = {}
2 def fibo(n):
3     if n<=1: return n
4     if n in rozw: return rozw[n]
5     rozw[n] = odp = fibo(n-1)+fibo(n-2)
6     return odp
7 k = int(input())
8 print(fibo(k))
```

W tym miejscu należy zauważyć, że czas działania powyższej implementacji byłby oczywiście dużo dłuższy, gdybyśmy nie korzystali z obliczonych wcześniej wartości (wiersz 4 kodu). W istocie złożoność obliczeniowa tego algorytmu bez zapamiętywania wartości wynosi około $O(1,618^n)$, a czas obliczenia F_{40} na komputerze autora przekracza 1 minutę.

Zaprezentowane algorytmy pokazują – na bardzo prostym przykładzie – istotę programowania dynamicznego. Mówiąc o liczbach Fibonacciego, warto również pamiętać, że istnieje alternatywny, szybszy sposób ich generowania zgodny z wzorem rekurencyjnym:

$$\begin{aligned} &F_0 = 0, \qquad F_1 = 1, \qquad F_2 = 1\,; \\ &F_{2n} = F_{n+1}^2 - F_{n-1}^2, \text{ dla } n > 1\,; \\ &F_{2n+1} = F_{n+1}^2 + F_n^2, \text{ dla } n \geq 1\,. \end{aligned}$$

Na wzmiankę zasługuje także metoda macierzowa wyznaczania liczb Fibonacciego, do której można zastosować algorytm szybkiego potęgowania:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Innym klasycznym zadaniem, które rozwiązujemy poprzez programowanie dynamiczne, jest *problem plecakowy* [7]. W tym problemie mamy do dyspozycji plecak o nośności W (maksymalna waga ładunku) oraz n przedmiotów, z których każdy ma swoją wagę w_i oraz wartość c_i dla $1 \leq i \leq n$. Należy wybrać takie przedmioty, aby ich sumaryczna waga nie przekraczała nośności plecaka, a sumaryczna wartość była maksymalna. Na przykład, dla poniższych przedmiotów:

i	w_i	c_i
1	2	6
2	4	11
3	2	9
4	1	2
5	1	1
6	3	8

oraz plecaka o nośności $W = 5$, otrzymamy maksymalną sumaryczną wartość przedmiotów równą 17 (dla przedmiotów $1, 3, 4$ lub dla przedmiotów $3, 6$).

Zauważmy tu od razu dwie rzeczy. Po pierwsze, optymalne rozwiązanie można uzyskać na więcej niż jeden sposób, co jest dość typowe w kontekście zadań algorytmicznych wymagających rozwiązania opartego na programowaniu dynamicznym[3]. Po drugie, problem plecakowy jest znacząco trudniejszy niż wyznaczanie liczb Fibonacciego. W szczególności, nie zadziałają tu proste metody oparte np. na *strategii zachłannej* (ang. *greedy algorithm*), typu: „wkładaj w każdym kroku przedmiot o największej wartości". Istotnie, jeśli zaczniemy pakowanie od „najcenniejszego" przedmiotu nr 2 (o wartości 11), to potem będziemy mogli już włożyć tylko przedmiot 4 lub 5 (oba mało wartościowe), bo pozostałe są za ciężkie. Wybierając przedmiot 4, uzyskamy wtedy maksymalną sumaryczną wartość zaledwie 13. Podobnie bezowocne okaże się przyjęcie strategii „zacznij od przedmiotów najlżejszych". W tym konkretnym przypadku zadziałałaby strategia „zacznij od przedmiotów o największej jednostkowej wartości (c_i/w_i)". Uzyskana kolejność rozpatrywania przedmiotów byłaby wówczas następująca: $3, 1, 2, 6, 4, 5$. Jednak w ogólności ta strategia także może nie dać wyniku optymalnego.

Aby skutecznie rozwiązać nasz problem za pomocą programowania dynamicznego, spróbujmy wyznaczyć regułę w postaci rekurencyjnego równania, określającego maksymalną sumę wartości przedmiotów dla plecaka o danej nośności. Niech $p(k, o)$ będzie rozwiązaniem tego zadania dla plecaka o nośności o (możliwym obciążeniu) i dla pierwszych k przedmiotów (zakładamy, że przedmioty są ponumerowane od 1 do n). Będziemy chcieli obliczyć maksymalną sumaryczną wartość plecaka, czyli $p(k, o)$, dla każdego k i o. W praktyce, musimy wypełnić prostokątną tablicę, wpisując w pole na przecięciu każdego k-tego wiersza i każdej o-tej kolumny naszą wartość $p(k, o)$ obliczoną na podstawie wcześniej otrzymanych wartości innych pól. Rozpatrzmy tu następujące przypadki:

- jeśli $k = 0$, wtedy dla każdego o wartość $p(0, o) = 0$, bo maksymalna wartość zera przedmiotów jest równa zero niezależnie od nośności plecaka;
- jeśli $w_k > o$, czyli k-ty przedmiot sam jeden waży więcej niż nośność plecaka, to oczywiście nie możemy go wziąć i przyjmujemy: $p(k, o) = p(k - 1, o)$, czyli maksymalna wartość k pierwszych przedmiotów mieszczących się do plecaka o nośności o jest taka sama, jak maksymalna wartość $k - 1$ pierwszych przedmiotów mieszczących się do tego samego plecaka;
- w przeciwnym przypadku mamy dwie możliwości:
  - k-tego przedmiotu nie bierzemy do plecaka i wtedy maksymalna sumaryczna wartość przedmiotów w plecaku jest nadal równa $p(k - 1, o)$, jak w poprzednim przypadku;
  - k-ty przedmiot bierzemy do plecaka i wtedy maksymalna wartość jest równa $c_k + p(k - 1, o - w_k)$.

To, którą z tych dwóch ostatnich możliwości wybierzemy (biorąc k-ty przedmiot lub nie biorąc) zależy oczywiście od tego, która da nam większą sumaryczną wartość plecaka.

Skąd wzięło się wyrażenie $c_k + p(k - 1, o - w_k)$, wyznaczające maksymalną wartość plecaka (o nośności o, którą aktualnie przyjmujemy), po dołożeniu do niego k-tego przedmiotu? Otóż przedmiot ten waży w_k, a więc – aby nie przekroczyć nośności o – dokładamy go do optymalnie zapakowanego plecaka o nośności $o - w_k$. Maksymalna sumaryczna wartość przedmiotów w tym plecaku wynosi oczywiście $p(k - 1, o - w_k)$, co policzyliśmy wcześniej, a po dołożeniu naszego k-tego przedmiotu wzrośnie o jego wartość, czyli o c_k.

[3] W zadaniach tych wymagana jest zwykle odpowiedź w postaci jednej liczby, np. maksymalnej lub minimalnej wartości czegoś, a nie sposobu jak ją konkretnie uzyskać (np. ważne jest ile wynosi największa możliwa wartość plecaka, a nie – co do niego włożyć).

Powyższe obserwacje możemy podsumować w postaci następującego równania rekurencyjnego:

$$p(k,o) = \begin{cases} 0, & \text{jeśli } k = 0\,; \\ p(k-1,o), & \text{jeśli } w_k > o\,; \\ \max(p(k-1,o), c_k + p(k-1,o-w_k)) & \text{w przeciwnym przypadku}\,. \end{cases}$$

W oparciu o to równanie możemy łatwo skonstruować algorytm obliczający wartości $p(k,o)$. Zakładając, że `W` to nośność plecaka, zaś `n` oznacza liczbę przedmiotów o wagach i wartościach podanych w listach `w` oraz `c`, odpowiednio, możemy przedstawić następującą implementację w języku Python:

```
p = [[0]*(W+1) for _ in range(n+1)] #macierz rozmiaru (W+1) na (n+1)
for k in range(n):
    for o in range(W+1):
        if w[k]>o: p[k+1][o] = p[k][o]
        else: p[k+1][o] = max(p[k][o], c[k] + p[k][o-w[k]])
print(p[-1][-1])
```

Wartości $p(k,o)$ są tu przechowywane w tablicy `p`, która – dla przykładowego zbioru przedmiotów rozważanego wyżej – wyglądać będzie następująco:

	$o=0$	$o=1$	$o=2$	$o=3$	$o=4$	$o=5$
$k=0$	0	0	0	0	0	0
$k=1$	0	0	6	6	6	6
$k=2$	0	0	6	6	11	11
$k=3$	0	0	9	9	15	15
$k=4$	0	2	9	11	15	17
$k=5$	0	2	**9**	11	15	**17**
$k=6$	0	2	9	11	15	17

Jak widać, mamy tu policzoną największą sumaryczną wartość plecaka o każdej możliwej pojemności $o \leq 5$ i dla każdego zestawu pierwszych k przedmiotów. Przykładowo, wartość 6 w trzecim wierszu (dla $k = 2$) wynika stąd, że mając do dyspozycji tylko $k = 2$ pierwsze przedmioty i plecak o nośności $o = 2$ lub $o = 3$, możemy wykorzystać tylko przedmiot nr 1 (o wadze 2 i wartości 6). Zwiększenie nośności plecaka do $o = 4$ lub $o = 5$ pozwoli na włożenie drugiego przedmiotu (zamiast pierwszego) o wadze 4 i wartości 11.

Ostatecznym wynikiem jest oczywiście wartość 17 z prawego dolnego narożnika (uwzględniająca wszystkie 6 przedmiotów i plecak o nośności $o = W = 5$). Pogrubione wartości w przedostatnim wierszu pokazują oba sposoby, na jakie to rozwiązanie można uzyskać w ostatnim kroku algorytmu. I tak, pogrubiona wartość 9, reprezentuje optymalny plecak o nośności $o = 2$ (tak naprawdę uzyskany wcześniej, tj. już po uwzględnieniu przedmiotu nr 3), do którego możemy teraz dołożyć przedmiot nr 6, zwiększając wagę z 2 do 5 i wartość z 9 do 17. Pogrubiona wartość 17 reprezentuje natomiast optymalny plecak o nośności $o = 5$ (też otrzymany wcześniej – po uwzględnieniu przedmiotu nr 4), do którego już nic nie dokładamy.

Dysponując kompletną tablicą `p`, możemy ją przejść „wstecz”, rozpoczynając od prawego dolnego narożnika, aby sprawdzić, które przedmioty mogą zostać wybrane do plecaka. Poniższa implementacja wypisuje jedno z możliwych poprawnych rozwiązań (dla naszych przykładowych danych uzyskamy odpowiedź: $6, 3$):

```
#n,c,w,W,p - z poprzedniej implementacji
odp = []
o = W
for k in range(n,0,-1):
    if o>=w[k-1] and p[k][o] == p[k-1][o-w[k-1]]+c[k-1]:
        odp.append(k)
        o -= w[k-1]
print(odp)
```

Podobnie, jak w przykładzie z ciągiem Fibonacciego, możemy zredukować wymagania pamięciowe naszego algorytmu, zapamiętując tylko najważniejsze dane. Zauważmy jednak, że podczas gdy tam cały problem „mieścił się" w jednowymiarowej liście, z której niezbędne były w gruncie rzeczy tylko *dwie ostatnie liczby Fibonacciego*, tak tutaj – zamiast dwuwymiarowej tablicy `p` – wystarczy nam pamiętać tylko jej *ostatnio obliczony, pojedynczy wiersz*:

```
#n,c,w,W - wcześniej wczytane dane
p = [0]*(W+1)
for k in range(n):
    for o in range(W, w[k]-1, -1):
        p[o] = max(p[o], c[k] + p[o-w[k]])
print(p[-1])
```

Jak łatwo zauważyć, używamy tu znacznie mniej pamięci, ale przez to nie możemy uzyskać informacji, jakie przedmioty trzeba wybrać do plecaka (znamy jedynie maksymalną sumaryczną wartość tych przedmiotów).

Jak pamiętamy, trzecia implementacja rozwiązania problemu wyznaczania k-tego elementu ciągu Fibonacciego oparta była na podejściu rekurencyjnym, które jest oczywiście możliwe także w przypadku problemu plecakowego. Poniżej prezentujemy rekurencyjną wersję rozwiązania, uwzględniającą również zapamiętywanie wartości pośrednich:

```
1  #n,c,w,W - z poprzedniej implementacji
2  rozw = {}
3  def p(k, o):
4      if k==0 or o==0: return 0
5      if (k,o) in rozw: return rozw[(k,o)]
6      if w[k-1]>o: odp = p(k-1,o)
7      else: odp = max(p(k-1,o), c[k-1]+p(k-1,o-w[k-1]))
8      rozw[(k,o)] = odp
9      return odp
10 print(p(n,W))
```

W tej implementacji, tak jak w przykładzie z ciągiem Fibonacciego, zapamiętujemy rozwiązania (p. wiersz 5 i 8 powyższego kodu). W istocie, mapa `rozw` zawiera najważniejsze informacje tablicy z pierwszej implementacji.

	$o = 1$	$o = 2$	$o = 3$	$o = 4$	$o = 5$
$k = 1$	0	6	6	6	6
$k = 2$	0	6	6	11	11
$k = 3$	0	9	9	15	15
$k = 4$	2	9	—	15	17
$k = 5$	—	9	—	—	17
$k = 6$	—	—	—	—	17

Warto zauważyć, że implementacja ta pozwala jednocześnie na uzyskanie informacji, które przedmioty włożyć do plecaka.

83 (IX) – Bilabirynt Bitezeusza – Gnomy

*Autor: **Filip Turoboś**, **Politechnika Łódzka***
*Kategoria: **Gnomy (II edycja – liga algorytmiczna)***
*Język programowania: **C++/Python***

83.1 Treść zadania

W stolicy zinformatyzowanego królestwa Bajtocji znajduje się słynny Bilabirynt. Był on miejscem, w którym doświadczenie zdobywały dziesiątki słynnych bajtockich bohaterów – Bitkules, Bajterman, Bit-woman i wielu innych.

Aspirujący do zostania kolejną bajtocką legendą Bitezeusz stoi u wrót Bilabiryntu. Nie jest zapewne tak silny jak Bitkules ani tak zręczny jak Bit-woman, ale posiada zupełnie inne rodzaje supermocy – niebywale inteligentnych przyjaciół (w tym Ciebie!). Bitezeusz zamierza opracować strategię, która pozwoli mu przejść labirynt zwycięsko – czyli wyjść z drugiego końca żywym, wynosząc tyle doświadczenia, ile to tylko możliwe. Ponieważ jesteś jego przyjacielem, prosi Cię o pomoc w tym przedsięwzięciu.

Zadanie

Wkraczając do labiryntu, Bitezeusz ma świadomość, że napotka pewną liczbę n potworów ($0 < n \leqslant 1\,000\,000$).

Każdy z potworów po pokonaniu nagradza Bitezeusza pewną całkowitą liczbą punktów doświadczenia, należącą do przedziału od 0 do $5\,000\,000$. Walki z potworami nie są obowiązkowe – można ich uniknąć wybierając odpowiednią drogę w labiryncie (obchodząc danego potwora „bokiem"). Są jednak z całą pewnością bardzo wyczerpujące i po każdej z nich Bitezeusz musi odpocząć, pomijając starcie z następną kreaturą w kolejce.

Twoim celem jest znalezienie takiej trasy przez labirynt, w której zmaksymalizujesz ilość zdobytego przez Bitezeusza doświadczenia, jednocześnie nie narażając go na uszczerbek na zdrowiu (który mógłby wyniknąć z walki z dwoma potworami pod rząd).

Opis wejścia

Pierwszy wiersz standardowego wejścia zawiera pojedynczą liczbę n ($0 < n \leqslant 1\,000\,000$) oznaczającą liczbę potworów, które można napotkać w labiryncie (potwory są podane w takiej kolejności, w jakiej może napotkać je Bitezeusz). W kolejnej linii znajdziesz n liczb całkowitych exp_i ($0 \leqslant exp_i \leqslant 5\,000\,000$), opisujących liczby punktów doświadczenia, które można uzyskać pokonując i-tego w kolejności potwora.

Opis wyjścia

Na wyjściu standardowym powinna się znaleźć pojedyncza liczba całkowita, opisująca maksymalną liczbę punktów doświadczenia, jakie może zdobyć Bitezeusz przechodząc przez labirynt.

Przykład

Dla przykładowego, podanego poniżej wejścia:

```
5
1 2 3 4 5
```

prawidłową odpowiedzią jest:

```
9
```

Z kolei dla przykładowego wejścia:

```
6
0 2 1 1 5 0
```

prawidłową odpowiedzią jest:

```
7
```

Wyjaśnienie przykładu

W pierwszym przykładzie Bitezeusz może pokonać pierwszego, trzeciego i piątego potwora (odpoczywając między nimi), by zdobyć maksymalną możliwą liczbę punktów doświadczenia.

Z kolei w przykładzie drugim, najefektywniejszą strategią jest zgładzenie tylko drugiej i przedostatniej kreatury[1].

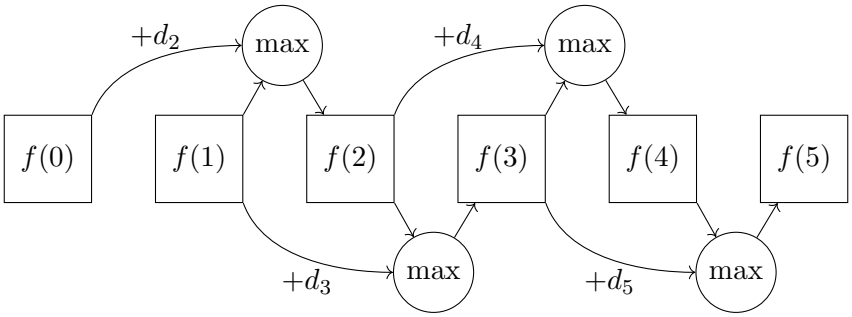
[1] Jeśli uspokoi to Twoje sumienie, to możemy Cię zapewnić, że w trakcie opracowywania przez Ciebie optymalnej strategii przechodzenia przez labirynt nie ucierpi żaden potwór.

83.2 Rozwiązanie

Niech d_i oznacza liczbę punktów doświadczenia, które można uzyskać za walkę z i-tym potworem, gdzie $1 \leq i \leq n$. Niech $f(i)$ oznacza maksymalną sumę punktów doświadczenia, które można uzyskać po i-tym potworze labiryntu. Oczywiście $f(0) = 0$ (nie było jeszcze potworów) oraz $f(1) = d_1$ (z jednym potworem na pewno będziemy walczyli).

Bitezeusz przy każdym spotkaniu z i-tym przeciwnikiem ($i > 1$) ma dwie możliwości: albo będzie walczył, albo nie będzie walczył. Walka z przeciwnikiem oznacza, że z poprzednim przeciwnikiem Bitezeusz na pewno nie walczył, co przekłada się na $d_i + f(i-2)$ punktów doświadczenia. Jeśli Bitezeusz nie wybierze walki z i-tym potworem, to uzyska $0 + f(i-1)$ punktów doświadczenia.

Równaniem rekurencyjnym, które opisuje f jest $f(i) = \max(d_i + f(i-2), f(i-1))$. To równanie rekurencyjne jest podobne do równania Fibonacciego w tym sensie, że wartość $f(i)$ zależy od $f(i-1)$ i $f(i-2)$. Oznacza to, że do implementacji tego równania można wykorzystać modyfikację dowolnej implementacji ciągu Fibonacciego przedstawionej we wprowadzeniu do tego działu.



83.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Bilabirynt_Bitezeusza_Gnomy/solution.py.

```python
n = int(input().strip())
potwor = list(map(int, input().split()))
punkty = [0, potwor[0]]
for i in range(n-1):
    punkty.append(max(punkty[i+1], punkty[i]+potwor[i+1]))
print(punkty[n])
```

83.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Bilabirynt_Bitezeusza_Gnomy/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<int64_t> potwor(n);
    for(int j = 0; j < n; j++){
        cin>>potwor[j];
    }

    vector<int64_t> punkty(n+1);
    punkty[1]=potwor[0];
    for(int i=1; i<n; i++) {
        punkty[i+1] = max(punkty[i], punkty[i-1]+potwor[i]);
    }
    cout<<punkty[n];
}
```

84 (IX) – Bilabirynt Bitezeusza – Skrzaty

Autor: Filip Turoboś, Politechnika Łódzka
Kategoria: Skrzaty (II edycja – liga algorytmiczna)
Język programowania: Scratch

84.1 Treść zadania

W stolicy zinformatyzowanego królestwa Bajtocji znajduje się słynny Bilabirynt. Był on miejscem, w którym doświadczenie zdobywały dziesiątki słynnych bajtockich bohaterów – Bitkules, Bajterman, Bit-woman i wielu innych.

Aspirujący do zostania kolejną bajtocką legendą Bitezeusz stoi u wrót Bilabiryntu. Nie jest zapewne tak silny jak Bitkules ani tak zręczny jak Bit-woman, ale posiada zupełnie inne rodzaje supermocy – niebywale inteligentnych przyjaciół (w tym Ciebie!). Bitezeusz zamierza opracować strategię, która pozwoli mu przejść labirynt zwycięsko – czyli wyjść z drugiego końca żywym, wynosząc tyle doświadczenia, ile to tylko możliwe. Ponieważ jesteś jego przyjacielem, prosi Cię o pomoc w tym przedsięwzięciu.

Wkraczając do labiryntu, Bitezeusz ma świadomość, że napotka pewną liczbę n potworów ($0 < n \leqslant 40\,000$). Każdy z potworów po pokonaniu nagradza Bitezeusza pewną całkowitą liczbą punktów doświadczenia, należącą do przedziału od 0 do $5\,000\,000$. Walki z potworami nie są obowiązkowe – można ich uniknąć wybierając odpowiednią drogę w labiryncie (obchodząc danego potwora "bokiem"). Są jednak z całą pewnością bardzo wyczerpujące i po każdej z nich Bitezeusz musi odpocząć, pomijając starcie z następną kreaturą w kolejce.

Twoim celem jest znalezienie takiej trasy przez labirynt, w której zmaksymalizujesz ilość zdobytego przez Bitezeusza doświadczenia, jednocześnie nie narażając go na uszczerbek na zdrowiu (który mógłby wyniknąć z walki z dwoma potworami pod rząd).

Zadanie

W nowym, pustym projekcie Scratch należy stworzyć dwie listy i nadać im nazwy:

- `DANE`
- `WYNIKI`

Listę `DANE` należy wypełniać ręcznie, zaś do listy `WYNIKI` Twój program powinien wpisać rezultat swojego działania.

Zadanie polega na napisaniu programu, który odczyta z listy `DANE` opis kolejnych potworów w labiryncie (potwory są podane w takiej kolejności, w jakiej może napotkać je Bitezeusz) i obliczy największą możliwą do zdobycia liczbę punktów doświadczenia.

Opis wejścia

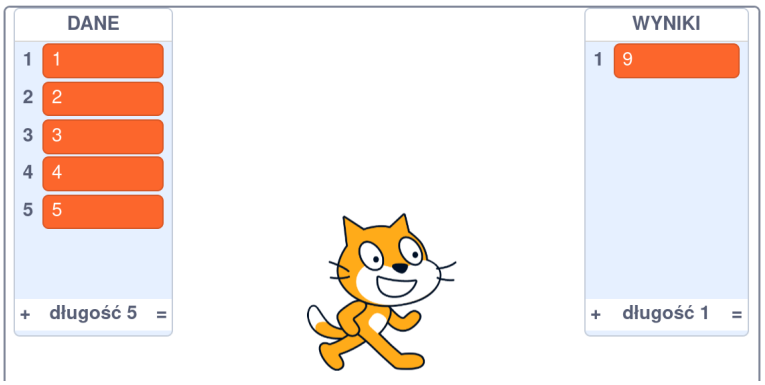
Lista wejściowa `DANE` zawiera n liczb całkowitych exp_i (gdzie: $0 < n \leqslant 40\,000$ oraz $0 \leqslant exp_i \leqslant 5\,000\,000$). Każda z nich definiuje liczbę punktów doświadczenia, które można uzyskać pokonując i-tego w kolejności potwora.

Opis wyjścia

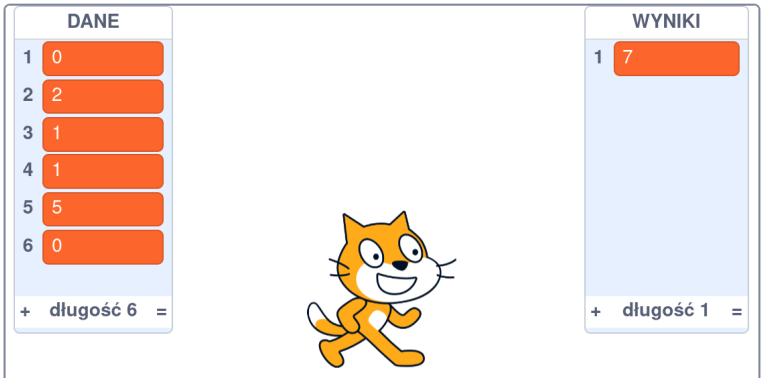
Po zakończeniu Twojego programu lista `WYNIKI` powinna zawierać dokładnie jeden element: liczbę całkowitą równą maksymalnej liczbie punktów doświadczenia, jakie może zdobyć Bitezeusz przechodząc przez labirynt.

Przykład

Na poniższych rysunkach pokazano przykładowe dane wejściowe i wyniki działania programu.



Rysunek 84.1: Wynik działania programu (przykład 1)



Rysunek 84.2: Wynik działania programu (przykład 2)

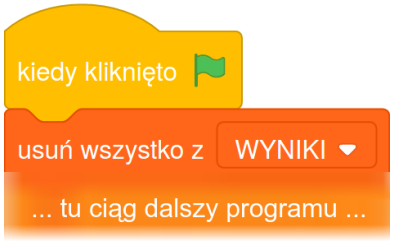
Wyjaśnienie przykładu

W pierwszym przykładzie Bitezeusz może pokonać pierwszego, trzeciego i piątego potwora (odpoczywając między nimi), by zdobyć maksymalną możliwą liczbę punktów doświadczenia. Z kolei w przykładzie drugim, najefektywniejszą strategią jest zgładzenie tylko drugiej i przedostatniej kreatury[1].

[1] Jeśli uspokoi to Twoje sumienie, to możemy Cię zapewnić, że w trakcie opracowywania przez Ciebie optymalnej strategii przechodzenia przez labirynt nie ucierpi żaden potwór.

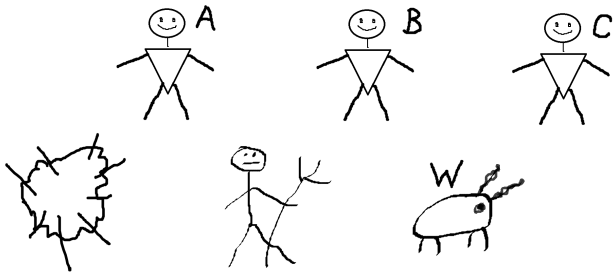
Uwagi

Pamiętaj, że listę `DANE` należy wypełniać ręcznie, wpisując dane z klawiatury. Warto przetestować działanie programu również dla innych wartości niż w przykładzie. Zawartość listy `WYNIKI` powinna być za każdym razem na początku usuwana. Twój program musi więc zaczynać się tak, jak pokazano poniżej:



84.2 Rozwiązanie

Na rysunku 84.3 jest naszkicowany Bitezeusz, który najpierw ma A punktów doświadczenia, a następnie B i C punktów (oczywiście niektóre z tych wartości mogą być takie same). Każdy potwór w dolnym rzędzie może dodać Bitezeuszowi punkty doświadczenia – przykładowo, walcząc z ostatnim wrogiem, Bitezeusz zyska W punktów doświadczenia.



Rysunek 84.3: Zbieranie doświadczenia przez Bitezeusza

Dla każdego potwora Bitezeusz musi się zastanowić, czy walczyć z nim, czy zrezygnować z walki. Jeśli z potworem walczy, to na pewno nie walczył z poprzednim potworem. Liczba punktów doświadczenia, którą uzyskuje w takiej sytuacji jest równa sumie punktów doświadczenia aktualnego potwora i punktów doświadczenia Bitezeusza po przejściu – z walką bądź bez walki – potwora, który był *przed poprzednim* potworem. W takim przypadku nasz „obrazkowy Bitezeusz" ma $C = A + W$ punktów doświadczenia.

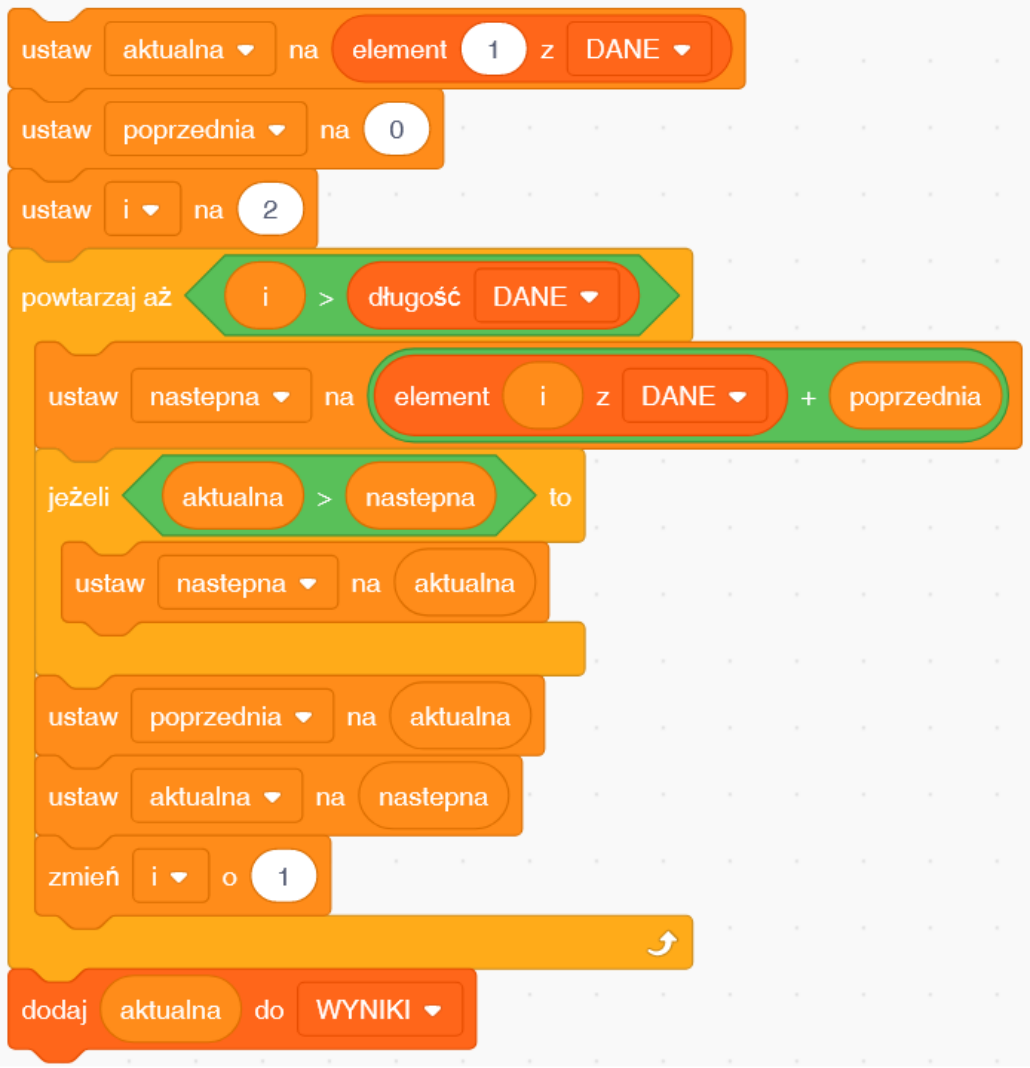
Z drugiej strony, jeśli z aktualnym potworem nie będzie walczył, to jego liczba punktów doświadczenia będzie taka sama jaką uzyskał po przejściu poprzedniego potwora. Wtedy „obrazkowy Bitezeusz" ma $C = B$ punktów doświadczenia.

Naturalnie, aby zmaksymalizować wartość punktów doświadczenia wystarczy wybrać większą z tych dwóch liczb. W każdym kroku naszego algorytmu musimy zatem znać dwie wartości: liczbę punktów doświadczenia Bitezeusza po poprzednim potworze oraz po potworze poprzedzającym poprzedniego.

Zauważmy, że są to te same dane, których potrzebowaliśmy, aby wyznaczyć k-ty wyraz ciągu Fibonacciego we wprowadzeniu do tego działu. Sam algorytm jest też analogiczny. W przykładowej implementacji, podanej na następnej stronie, do przechowywania tych danych posługujemy się dwiema zmiennymi: `poprzednia` i `aktualna`, które początkowo inicjalizowane są – odpowiednio – wartością 0 oraz wartością punktów doświadczenia pierwszego potwora. Wczytując punktację kolejnych potworów, aktualizujemy te zmienne, posiłkując się pomocniczą zmienną `nastepna`, która chwilowo przechowuje nową, wyliczoną wartość doświadczenia. Oczywiście to wyliczenie związane jest ze stwierdzeniem, czy nasza sumaryczna punktacja po poprzednim potworze i po walce z następnym jest większa, czy mniejsza niż po walce z bieżącym.

Zauważmy na koniec, że przedstawione tu rozwiązanie jest analogiczne do drugiej implementacji algorytmu wyznaczania k-tego wyrazu ciągu Fibonacciego, przedstawionej we wprowadzeniu. Moglibyśmy również zastosować podejście analogiczne do pierwszej implementacji, tzn. zapamiętać *wszystkie* kolejno wyznaczane liczby punktów doświadczenia, zapisując je w tablicy (chociaż wciąż potrzebujemy oczywiście tylko dwóch ostatnich). Takie rozwiązanie znajdziesz w pliku:

./zadania/09_Programowanie_dynamiczne/Bilabirynt_Bitezeusza_Skrzaty/solution.sb3.



Rysunek 84.4: Bilabirynt Bitezeusza – przykładowa implementacja (bez zapamiętywania całej tablicy)

85 (IX) – Rajd przez płotki – Gremliny

Autor: ***Łukasz Skonieczny***, ***Politechnika Warszawska***
Kategoria: ***Gremliny (IV edycja – liga algorytmiczna)***
Język programowania: ***C++/Python***

85.1 Treść zadania

– *„Biegnij najszybciej jak się da! Tu liczy się tylko czas!”* – zmierzał już na start corocznego Rajdu Przez Płotki, gdy mimowolnie przypomniały mu się porady kolegów.

– *„Ale przecież trasa składa się z położonych obok siebie torów, oddzielonych małym progiem. Mógłbym czasem przeskakiwać na drugi tor, żeby ominąć dłuższe serie płotków znajdujących się na moim torze...”* – w ten sposób próbował zwykle z nimi polemizować.

– *„Nie kombinuj! Przeskok na drugi tor spowoduje, że tylko stracisz czas!”* – szybko ucinali jego argumentację.

– *„Ale przecież przeskok przez płotek na torze też powoduje, że tracę czas, więc gdybym rozsądnie wybie...”* – próbował wyjaśniać, ale nigdy nie pozwalali mu dokończyć.

–*„To nic nie da! Po prostu biegnij! Nie myśl! Biegnij!”*

Stał już samotnie na linii startu. Miał przed sobą dwa biegnące obok siebie tory tej samej długości. Mógł wybrać, z którego startuje. 10, 9... *„Nie myśl! Biegnij!”* – akurat! – to przecież nie przynosiło mu nigdy dobrych efektów. Nie! Tym razem nie będzie po prostu biegł bezmyślnie! 8, 7... Zna liczbę i układ płotków na każdym torze. Przyjmie, że przeskok przez płotek wiąże się z taką samą stratą czasu, co przeskok na sąsiedni tor. 6, 5... Ułoży plan biegu, który zminimalizuje łączną liczbę skoków i przeskoków. Pokaże wszystkim, że czasem warto myśleć! 4, 3...

Opis wejścia

W pierwszym wierszu standardowego wejścia znajduje się liczba całkowita $1 \leq n \leq 1\,000\,000$ oznaczająca długość trasy podaną jako liczbę dziesięciometrowych segmentów. W kolejnych dwóch liniach znajduje się po n znaków, które opisują wygląd dwóch torów. Każdy znak oznacza jeden kolejny dziesięciometrowy segment toru. Znak „.” oznacza segment pusty, a znak „x” oznacza segment, na końcu którego znajduje się płotek.

Opis wyjścia

Na standardowe wyjście należy wypisać najmniejszą liczbę manewrów (skoków przez płotki oraz przeskoków na sąsiedni tor), które trzeba wykonać by przebyć całą trasę.

Przykłady

Dla przykładowego, podanego poniżej wejścia:

```
20
xxxxxx......xxxxxxxx
.......xxx..........
```

prawidłową odpowiedzią jest:

```
2
```

Z kolei dla innego wejścia:

```
13
..xx...xx...x
....x..xxx..x
```

prawidłową odpowiedzią jest:

```
4
```

Wyjaśnienie

W pierwszym przypadku trasa składa się z 20 segmentów. Na pierwszym torze znajduje się łącznie 14 płotków, a na drugim – 3 płotki. Gdyby biec cały czas pierwszym torem, trzeba byłoby wykonać 14 skoków. Gdyby biec drugim torem – 3 skoki. Optymalnie będzie jednak zacząć rajd od drugiego toru, na siódmym lub ósmym segmencie przeskoczyć na pierwszy tor, a potem wrócić na drugi tor na segmencie 11, 12 lub 13. W ten sposób trasa zostanie pokonana bez skakania przez płotki, a jedynie za pomocą dwóch przeskoków na sąsiednie tory. Stąd 2 na wyjściu.

W drugim przypadku trasa składa się z 13 segmentów. Optymalnie będzie zacząć rajd na drugim torze, przeskoczyć na pierwszy tor na piątym segmencie (pamiętajmy, że płotek znajduje się dopiero na końcu segmentu) i biec po tym torze aż do mety, skacząc po drodze przez 3 płotki.

...2, 1, 0, START!

85.2 Rozwiązanie

Trasa rajdu składa się dwóch torów, nazwijmy je `A` i `B`. Każdy tor jest zbudowany z n segmentów. Ponumerujmy te segmenty od 1 do n. Niech A_i oznacza najmniejszą liczbę manewrów, jaką trzeba wykonać, by znaleźć się na końcu segmentu i na torze `A`, już po przeskoczeniu płotka na segmencie i toru, jeżeli tam się znajduje. B_i będzie zdefiniowane analogicznie. W zadaniu musimy zatem znaleźć $\min(A_n, B_n)$.

Są dwa sposoby, żeby znaleźć się na końcu i tego segmentu toru `A`:

1. wbiegamy na niego z $i-1$ segmentu toru `A`;
2. wbiegamy na niego z $i-1$ segmentu toru `B`.

W obu przypadkach należy przeskoczyć przez płotek na i-tym segmencie toru `A` – jeśli się tam znajduje. W drugim przypadku należy dodatkowo przeskoczyć przez próg oddzielający oba tory. Zatem:

$$A_i = \min(A_{i-1}, B_{i-1}+1) + (1 \text{ jeśli na } i\text{-tym segmencie toru } \texttt{A} \text{ jest płotek}).$$

Analogicznie możemy obliczyć wartość B_i:

$$B_i = \min(B_{i-1}, A_{i-1}+1) + (1 \text{ jeśli na } i\text{-tym segmencie toru } \texttt{B} \text{ jest płotek}).$$

W warunkach początkowych mamy $A_0 = 0, B_0 = 0$. Nie musimy wykonywać żadnych manewrów, by znaleźć się na końcu zerowego segmentu, czyli *de facto* – na starcie toru `A` lub `B` (regulamin rajdu mówi, że możemy zacząć od dowolnie wybranego toru).

Powyższe zależności prowadzą do prostego rozwiązania metodą programowania dynamicznego. W rozwiązaniu tym wyliczamy kolejne wartości A_i, B_i na podstawie ich wartości poprzednich.

85.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Rajd_przez_plotki/solution.py.

```
def main():
    #wczytywanie danych
    n = int(input())
    t1 = input().strip()
    t2 = input().strip()
    #wyznaczanie wartości
    counter1, counter2 = 0, 0
    for i in range(n):
        c1 = min(counter1, 1 + counter2)
        c2 = min(counter2, 1 + counter1)
        if t1[i]=='x':
            c1 += 1
        if t2[i]=='x':
            c2 += 1
        counter1, counter2 = c1, c2
    #wypisanie odpowiedzi
    print(min(counter1, counter2))
if __name__ == '__main__':
    main()
```

85.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Rajd_przez_plotki/solution.cpp.

```cpp
#include <iostream>
#include <vector>
#include<algorithm>
using namespace std;
int main(){
    //wczytanie danych
    int n;
    cin >> n;
    vector<bool> T1(n), T2(n);
    for(int i=0; i<n; i++) {
        char c;
        cin >> c;
        T1[i] = (c=='x');
    }
    for(int i=0; i<n; i++) {
        char c;
        cin >> c;
        T2[i] = (c=='x');
    }
    //wyznaczanie wartości
    int counter1 = 0, counter2 = 0;
    for(int i=0; i<n; i++) {
        int c1 = T1[i] + min(counter1, 1 + counter2);
        int c2 = T2[i] + min(counter2, 1 + counter1);
        counter1 = c1;
        counter2 = c2;
    }
    //wypisanie odpowiedzi
    cout << min(counter1, counter2);
}
```

86 (IX) – Huśtawka – Gremliny

*Autor: **Bartłomiej Stasiak**, **Politechnika Łódzka***
*Kategoria: **Gremliny (IV edycja – zawody algorytmiczne – etap lokalny)***
*Język programowania: **C++/Python***

86.1 Treść zadania

Procesłowacja jest państwem na wskroś nowoczesnym i wysoce zinformatyzowanym. Programowania i algorytmiki uczą się tu wszyscy niemalże od kołyski, co jednakowoż – w opinii niektórych osób o staroświeckich poglądach – odbija się niekorzystnie na pewnych aspektach rozwoju dzieci i młodzieży. Mowa tu zwłaszcza o ruchu fizycznym, a raczej o jego braku...

Miasto stołeczne, Procław, zaproponowało swoim mieszkańcom pewien środek zaradczy: innowacyjny projekt placu zabaw, mającego w założeniu „wyciągnąć" choć część dzieci ze świata wirtualnego.

Do realizacji tego niełatwego zadania wykorzystano najnowocześniejsze środki techniki i wynalazki przemysłu rozrywkowego. Jednakże spośród wszystkich atrakcji, łącznie z antygrawitacyjnymi zjeżdżalniami dwukierunkowymi, kolejkami wysokogórskimi i szybkoobrotowymi karuzelami hipersonicznymi, największym hitem okazała się... zwykła huśtawka.

Swoją popularność ta niepozorna atrakcja zawdzięczała skutecznemu rozwiązaniu odwiecznego problemu niedopasowania wagi obu huśtających się osób. Sprytny system siłowników ukryty w korpusie huśtawki pozwolił wreszcie huśtać się swobodnie nawet przedszkolakowi z bratem-licealistą i... trudno się dziwić, że kolejka do huśtawki codziennie biła nowe rekordy długości!

Ściślej mówiąc, były to dwie kolejki: w jednej stały dzieci podchodzące do jednego, a w drugiej – do drugiego siedziska huśtawki. Czas huśtania został ograniczony do minuty, ale i tak urządzenie pracowało non-stop przez cały dzień, zużywając przy okazji pokaźne ilości energii elektrycznej. Dyrekcja placu zabaw, po pierwszym rachunku za prąd, postanowiła wdrożyć program oszczędnościowy, oparty na pewnym sprytnym pomyśle.

Otóż, ponieważ zużycie prądu jest oczywiście proporcjonalne do różnicy wagi obu użytkowników huśtawki, zatem opłaca się tak dobierać dzieci, aby ta różnica była możliwie jak najmniejsza. Dyskretny pomiar wagi dzieci w obu kolejkach nie stanowił problemu, aby jednak uniknąć zamieszania i nie zmieniać ich kolejności, przyjęto proste rozwiązanie, polegające na możliwości pozostawiania jednego z dzieci na huśtawce na następną minutę (lub dwie, trzy, itd.), tak aby łączna różnica wagi dla wszystkich par użytkowników była jak najmniejsza.

Zadanie

Znając wagę dzieci w obu kolejkach (w takim porządku, w jakim w nich stoją), należy ustalić, które dzieci powinno się pozostawić na huśtawce (i na jaką liczbę minut), tak aby zminimalizować łączne zużycie prądu, zgodnie z powyższym opisem. Jako odpowiedź, wystarczy podać sumaryczną różnicę wagi dla wszystkich par dzieci huśtających się danego dnia.

Jako pierwsza para huśta się pierwsze dziecko z jednej kolejki z pierwszym dzieckiem z drugiej. Analogicznie, ostatnia para to ostatnie dziecko z jednej kolejki z ostatnim dzieckiem z drugiej.

Opis wejścia

W pierwszym wierszu wejścia znajdują się dwie liczby całkowite n, m, oddzielone pojedynczą spacją (przy czym $1 \leq n \leq 3\,000$; $1 \leq m \leq 3\,000$), oznaczające liczbę dzieci w pierwszej i drugiej kolejce, odpowiednio. Drugi wiersz zawiera n liczb całkowitych, oddzielanych pojedynczą spacją, reprezentujących wagi dzieci stojących w pierwszej kolejce. Analogicznie, trzeci wiersz wejścia zawiera m liczb całkowitych i reprezentuje drugą kolejkę. Wagi dzieci są z zakresu od 1 do 1000 (wspomniany wcześniej brak aktywności fizycznej ma – jak się łatwo domyślić – ujemny, albo może raczej **mocno dodatni** wpływ na masę ciała Procesłowackiej młodzieży).

Opis wyjścia

Na wyjściu należy podać jedną liczbę całkowitą, oznaczającą sumę różnic wagi dzieci huśtających się ze sobą w kolejnych minutach.

Przykłady

Przykład 1:

Dla poniższego wejścia:

```
5 5
40 30 50 20 20
40 50 30 50 20
```

poprawnym wyjściem jest:

```
10
```

Przykład 2:

Dla wejścia:

```
7 2
35 21 33 41 32 20 22
40 21
```

poprawnym wyjściem jest:

```
42
```

Przykład 3:

Dla danych wejściowych:

```
4 3
31 23 24 17
31 15 11
```

poprawnym wyjściem jest:

```
23
```

Wyjaśnienie

W pierwszym przykładzie w obu kolejkach stoi tyle samo dzieci. Gdyby jednak wpuszczać co minutę po jednym dziecku z obu kolejek, różnice wag w kolejnych parach wyniosłyby:

```
0  (= |40 - 40|)
20 (= |30 - 50|)
20 (= |50 - 30|)
30 (= |20 - 50|)
0  (= |20 - 20|)
```

co sumarycznie dałoby wynik $20 + 20 + 30 = 70$. Jak łatwo zauważyć, zatrzymanie pierwszego dziecka z pierwszej kolejki na drugą minutę (tak, aby huśtało się jeszcze z dzieckiem o wadze 50 z drugiej kolejki) jest korzystniejsze, ponieważ różnice wag wyniosą wtedy:

```
0  (= |40 - 40|)
10 (= |40 - 50|)
0  (= |30 - 30|)
0  (= |50 - 50|)
0  (= |20 - 20|)
0  (= |20 - 20|)
```

a zatem sumaryczna różnica wag będzie równa zaledwie 10. Zauważmy też, że ostatnie dziecko z drugiej kolejki musiało w tym przypadku również pozostać na drugą minutę, aby ostatnie dziecko z pierwszej kolejki miało się z kim huśtać.

W drugim przykładzie, kolejki są różnej długości. Jest więc jasne, że przynajmniej jedno dziecko z drugiej kolejki będzie huśtać się z wieloma dziećmi z pierwszej. Problem sprowadza się do decyzji, w którym momencie powinna nastąpić zmiana dziecka w drugiej kolejce. Jeśli nastąpi zaraz po pierwszej parze (a taką decyzję mogłaby sugerować identyczna waga drugich dzieci w obu kolejkach), to otrzymamy następujące różnice wag:

```
5  (= |35 - 40|)
0  (= |21 - 21|)
12 (= |33 - 21|)
20 (= |41 - 21|)
11 (= |32 - 21|)
1  (= |20 - 21|)
0  (= |22 - 21|)
```

Suma różnic wyniesie wówczas $5 + 12 + 20 + 11 + 1 = 47$. Nie jest to jednak wartość optymalna. Zauważmy mianowicie, że w drugiej kolejce drugie dziecko jest dużo lżejsze niż pierwsze, natomiast trzecie, czwarte i piąte dziecko w pierwszej kolejce jest stosunkowo ciężkie, co generuje spore różnice wag ($12, 20, 11$, odpowiednio). Sensowniej byłoby więc pozostawić pierwsze dziecko z drugiej kolejki na huśtawce aż do piątego dziecka z pierwszej kolejki włącznie. Dopiero dwie ostatnie osoby z pierwszej kolejki będą stanowić dobrą parę dla drugiego dziecka z drugiej. Różnice wag wyniosą wówczas:

```
5  (= |35 - 40|)
19 (= |21 - 40|)
7  (= |33 - 40|)
1  (= |41 - 40|)
8  (= |32 - 40|)
1  (= |20 - 21|)
1  (= |22 - 21|)
```

co da w rezultacie sumaryczną wartość 42.

W trzecim przykładzie istnieje kilka optymalnych rozwiązań, np.:

```
0  (= |31 - 31|)
8  (= |23 - 15|)
9  (= |24 - 15|)
6  (= |17 - 11|)
```

albo:

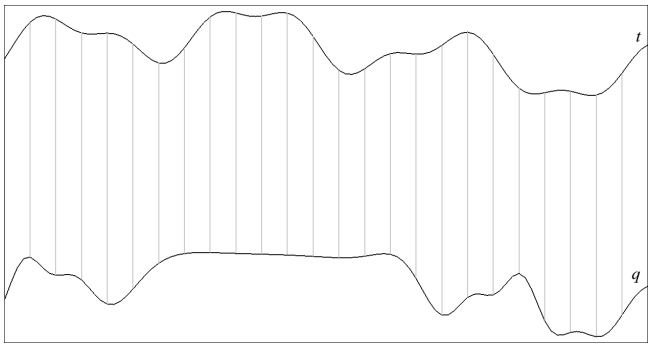
```
0  (= |31 - 31|)
8  (= |23 - 31|)
7  (= |24 - 31|)
2  (= |17 - 15|)
6  (= |17 - 11|)
```

jednak w każdym z nich otrzymamy tę samą sumę różnic wag, równą 23 i to jest poprawna odpowiedź.

86.2 Rozwiązanie

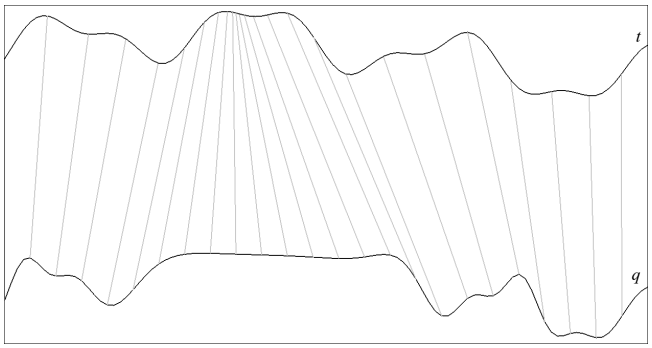
Zadanie polega na optymalnym dopasowaniu do siebie dwóch sekwencji liczbowych, przy założeniu, że mogą one być w dowolny sposób „rozciągane" w czasie względem siebie. To „rozciąganie" wyraża się oczywiście zatrzymywaniem danego dziecka na huśtawce przez większą liczbę minut. Przykładowo, gdyby każde dziecko z danej kolejki spędzało na huśtawce dwie minuty, oznaczałoby to, że czas w tej kolejce płynie w pewnym sensie dwa razy wolniej.

Jest to analogia do problemu porównywania ze sobą sygnałów takich jak fragmenty nagrań dźwiękowych – na przykład w zadaniu rozpoznawania słów w sygnale mowy. Każdy mówca może wszak mówić w innym tempie, jak również rozciągać, czy skracać w czasie poszczególne głoski. Dwa nagrania porównywane bezpośrednio „jeden-do-jednego", mogą więc wydawać się pozornie bardzo różne od siebie – nawet jeśli na obu wypowiedziano to samo słowo, czy zdanie (Rys. 86.1).



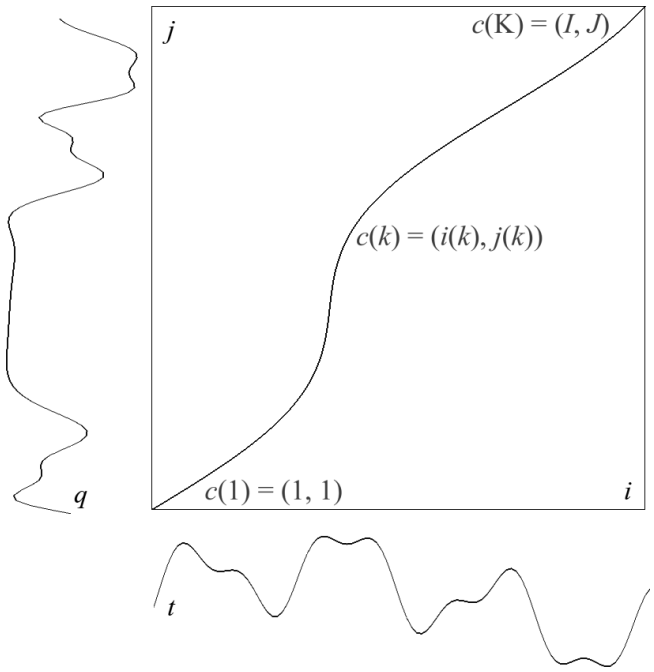
Rysunek 86.1: Przykład bezpośredniego porównania dwóch sekwencji, t i q – jak łatwo zauważyć, sekwencja q jest w środkowej części rozciągnięta w czasie względem t [18]

Jeżeli jednak zmienimy odpowiednio „upływ czasu", to może okazać się, że obie sekwencje pasują do siebie dużo bardziej (Rys. 86.2).



Rysunek 86.2: Przykład optymalnego dopasowania sekwencji t i q [18]

Do znalezienia takiego optymalnego dopasowania obu sekwencji w czasie, stosujemy algorytm DTW (ang. *Dynamic Time Warping*), który dostarcza nam informacji o tym, który element pierwszej sekwencji powinien być porównany z którym elementem drugiej, tak aby sumaryczna różnica wartości porównywanych elementów była jak najmniejsza. Odkładając kolejne numery elementów sekwencji $t(i)$ na osi poziomej (od lewej do prawej), a sekwencji $q(j)$ na osi pionowej (od dołu do góry), poszukujemy optymalnej ścieżki dopasowania c na płaszczyźnie $i - j$ (Rys. 86.3).



Rysunek 86.3: Optymalna ścieżka dopasowania sekwencji t i q [18]

Wartości I i J oznaczają długość, czyli liczbę elementów sekwencji t i q, odpowiednio, zaś k jest numerem pary optymalnie dopasowanych do siebie elementów sekwencji. Zauważmy przy tym, że całkowita liczba par składających się na optymalną ścieżkę (oznaczona tu przez K) jest zwykle większa zarówno od I, jak i od J – zależy to oczywiście od konkretnego kształtu ścieżki.

Algorytm DTW oparty jest na programowaniu dynamicznym i polega zasadniczo na wypełnieniu dwuwymiarowej tablicy g (reprezentującej płaszczyznę $i - j$) odpowiednimi wartościami. Oblicza się je tak, aby wartość w tablicy g w punkcie o współrzędnych (i, j) reprezentowała minimalny koszt dopasowania początkowych odcinków obu sekwencji (ściślej rzecz biorąc: odcinka złożonego z pierwszych i elementów sekwencji t oraz odcinka złożonego z pierwszych j elementów sekwencji q).

Poszczególne wartości tablicy g oblicza się wiersz po wierszu (od lewej do prawej, od dołu do góry – na Rys. 86.3), zgodnie z poniższym wyrażeniem (*równaniem DTW*):

$$\underset{\substack{i=1,2,\ldots,I \\ j=1,2,\ldots,J}}{\forall} \quad g[i,j] = d(i,j) + \min \begin{cases} g[i,j-1] \\ g[i-1,j-1] \\ g[i-1,j] \end{cases}$$

Funkcja $d(i, j)$ oznacza tu wynik porównania i-tego elementu sekwencji t z j-tym elementem sekwencji q (w naszym przypadku będzie to wartość bezwzględna różnicy wag i-tego dziecka z pierwszej kolejki i j-tego dziecka z drugiej kolejki).

Jak łatwo zauważyć, wartość tablicy g w bieżącym punkcie o współrzędnych (i, j) obliczamy jako wynik porównania odpowiednich elementów obu sekwencji **zsumowany** z minimalnym kosztem „dotarcia” do jednego z trzech wcześniejszych punktów sąsiadujących z bieżącym. Mówiąc ściślej, rozważamy tu punkt położony poniżej bieżącego, punkt położony na lewo od bieżącego i punkt położony jednocześnie na lewo i poniżej bieżącego. Jak łatwo zauważyć, prawidłowe działanie algorytmu DTW wymaga wstępnej inicjalizacji „zerowego” (najniższego) wiersza i zerowej (skrajnie lewej) kolumny tablicy g. Wpisujemy tam jakąś bardzo dużą wartość,

tak aby nigdy nie mogła zostać wybrana przez funkcję $minimum$ w powyższym wzorze. Jedynym wyjątkiem jest punkt przecięcia zerowego wiersza i zerowej kolumny, czyli $g[0,0]$. Inicjalizujemy go wartością 0, co zagwarantuje nam, że uzyskana optymalna ścieżka dopasowania będzie zaczynać się w lewym dolnym narożniku tablicy. Oczywiście wynikiem algorytmu DTW, czyli minimalnym kosztem dopasowania obu porównywanych sekwencji, jest wartość, którą odczytamy z prawego górnego narożnika tablicy, czyli $g[I,J]$.

86.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Hustawka/solution.py.

```
def main():
    lengths = list(map(int, input().split()))
    s = list(map(int, input().split()))
    t = list(map(int, input().split()))
    n, m = len(s), len(t)
    prevRow, nextRow = [99999999]*(m + 1), [99999999]*(m + 1)
    prevRow[0] = 0
    for i in range(n):
        for j in range(m):
            cost = abs(s[i] - t[j])
            nextRow[j+1] = cost + min(prevRow[j+1], nextRow[j], prevRow[j])
        prevRow[0] = 99999999
        nextRow,prevRow = prevRow,nextRow
    print(prevRow[m])

if __name__ == "__main__":
    main()
```

86.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Hustawka/solution.cpp.

```cpp
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

int dynamic(const vector<int> &s, const vector<int> &t) {
    const int infty = 1000000000;
    vector<int> prevRow(tLen + 1, infty);
    vector<int> nextRow(tLen + 1, infty);
    prevRow[0] = 0;
    for (int i = 0; i < sLen; i++) {
        for (int j = 0; j < tLen; j++) {
            int cost = abs(s[i], t[j]);
            nextRow[j+1] = cost + min(min(prevRow[j+1], prevRow[j]), nextRow[j]);
            prev[0] = infty;
        }
        swap(prevRow, nextRow);
    }
    return prev[tLen];
}

int main() {
    int sLen, tLen;
    cin >> sLen >> tLen;
    vector<int> s(sLen);
    vector<int> t(tLen);
    for(int i = 0; i < sLen; i++) cin >> s[i];
    for(int i = 0; i < tLen; i++) cin >> t[i];
    cout << dynamic(s, t) << endl;
}
```

87 (IX) – Precz z palindromami! – Gremliny

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gremliny (II edycja – zawody algorytmiczne – etap lokalny)
Język programowania: C++/Python

87.1 Treść zadania

Wyobraź sobie dłuuugą taśmę, na której wydrukowano ciąg liter. Twoim zadaniem jest pozbycie się całej taśmy (skrócenie jej do zera) poprzez wykonanie pewnej liczby ruchów polegających na wycinaniu z niej *palindromów*, czyli wyrazów, które czytane wspak wyglądają tak samo. Na przykład palindromem jest ciąg znaków `BACDCAB` lub `ABBA`. Palindromem jest również pojedyncza litera.

W jednym ruchu można wyciąć tylko jeden wyraz/palindrom złożony z liter bezpośrednio ze sobą sąsiadujących. Na przykład z ciągu `XYZYWVVY` można wyciąć wyraz `YZY` lub `VV`, względnie dowolnie wybraną pojedynczą literę.

Jeśli usunięcie (wycięcie) palindromu spowodowało rozdzielenie taśmy na dwa rozłączne kawałki, przed następnym ruchem ulegają one złączeniu. Na przykład po wycięciu wyrazu `YZY` z ciągu `XYZYWVVY` otrzymamy ciąg `XWVVY`.

Jeśli cały ciąg liter na taśmie jest palindromem, wtedy można go usunąć w jednym ruchu.

Zadanie

Twoim zadaniem jest znalezienie najmniejszej liczby ruchów potrzebnych do usunięcia całego ciągu. Twój program powinien czytać dane ze standardowego wejścia i wypisywać swój wynik na standardowe wyjście.

Opis wejścia

Pierwszy wiersz danych wejściowych zawiera liczbę naturalną n ($1 \leq n \leq 200$), oznaczającą długość ciągu liter na taśmie. W drugim wierszu zapisane jest n wielkich liter alfabetu łacińskiego bez odstępów.

Opis wyjścia

Program powinien wypisać wiersz tekstu zawierający minimalną liczbę ruchów potrzebnych do usunięcia całego ciągu.

Przykład

Dla danych wejściowych:

```
7
CEBADDB
```

prawidłowym wynikiem jest:

```
4
```

Istotnie, należy najpierw usunąć wyraz `DD` (1. ruch).

Otrzymujemy ciąg znaków:

```
CEBAB
```

Teraz usuwamy wyraz `BAB` (2. ruch) i otrzymujemy:

```
CE
```

Następnie usuwamy na przykład literę `C` (3. ruch) i dostajemy:

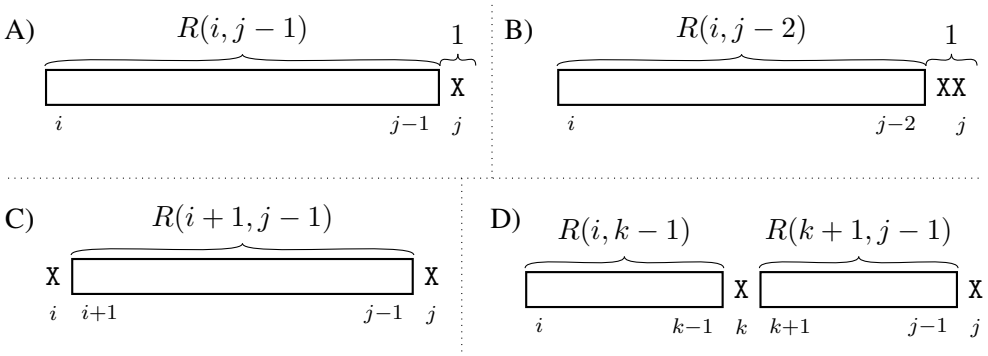
```
E
```

Ostatnią literę `E` usuwamy w 4. ruchu.

87.2 Rozwiązanie

Przez l_i oznaczmy i-tą literę na taśmie, gdzie $1 \leq i \leq n$. Minimalną liczbę ruchów potrzebnych do usunięcia ciągu od i-tej do j-tej litery oznaczymy przez $R(i,j)$. Zgodnie z warunkami zadania $R(i,i) = 1$. Oczywiście, rozwiązaniem całego zadania jest $R(1,n)$.

Zastanówmy się, kiedy zostanie usunięta ostatnia litera w ciągu $i-j$. Wszystkie możliwe przypadki rozrysowane są poniżej:

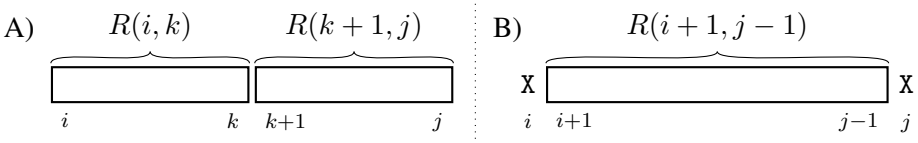


Ostatnia litera może zostać usunięta jako pojedyncza litera (przypadek A) – wtedy wartość $R(i,j)$ jest równa $1 + R(i,j-1)$. Ostatnia litera może być częścią większego palindromu – wtedy istnieje k takie że $i \leq k < j$, dla którego $l_k = l_j$. Dla tego przypadku (przypadek D) $R(i,j)$ jest równe $R(i,k-1) + R(k+1,j-1)$, przy czym dla $k = i$ (przypadek C) mamy $R(i,j) = R(i+1,j-1)$, a dla $k = j-1$ (przypadek B) mamy $R(i,j) = R(i,j-2)+1$. Żeby zredukować szczególne przypadki (B i C) załóżmy, że jeśli $s > r$, to $R(s,r) = 0$. Wtedy równanie rekurencyjne sprowadza się do postaci $R(i,j) = R(i,k-1) + \max(1, R(k+1,j-1))$. Uwzględniając wszystkie przypadki, pełne równanie rekurencyjne jest następujące:

$$R(i,j) = \begin{cases} 0 & \text{gdy } i > j \\ 1 & \text{gdy } i = j \\ \min\Big(1 + R(i,j-1), & \\ \quad \min\big\{R(i,k-1) + \max(1, R(i+1,j-1)) & \\ \quad\quad : i \leq k < j \wedge l_j = l_k\big\}\Big) & \end{cases}$$

Przy takim równaniu rekurencyjnym, najprostszą i najszybszą do napisania implementacją jest funkcja rekurencyjna z zapamiętywaniem wartości pośrednich. Ta wersja rozwiązania jest zaimplementowana w plikach `solutionreku.py` i `solutionreku.cpp` (podanych dalej).

Alternatywnie, dla tej samej definicji funkcji R (oznaczającej minimalną liczbę ruchów), możemy ułożyć inne równanie rekurencyjne, które nie będzie uwzględniało tylu przypadków. Przy tym podejściu mogą wystąpić tylko dwa przypadki – pokazane poniżej:



Przedział $i-j$ dzielimy na dwa przedziały i wtedy $R(i,j) = \min\{R(i,k) + R(k+1,j) : i \leq k < j\}$ (przypadek A). Przy takim podejściu do problemu jest jeden szczególny przypadek (B), gdy $l_i = l_j$. Wówczas

wartość $R(i,j)$ może być równa $\max(R(i+1,j-1),1)$. Funkcja $\max$ jest istotna dla $i=j$ i $i+1=j$; wtedy $R(i,j)=1$, a $R(i-1,j+1)=0$.

Reasumując:

$$R(i,j)=\begin{cases}0 & \text{gdy } i>j\\ 1 & \text{gdy } i=j\\ \min\{R(i,k)+R(k+1,j)\colon i\le k<j\} & \text{gdy } l_i\neq l_j\\ \min\Big(\max(R(i+1,j-1),1),\\ \quad \min\{R(i,k)+R(k+1,j)\colon i\le k<j\}\Big) & \text{w przeciwnym przypadku}\end{cases}$$

W plikach `solution.py` i `solution.cpp` (podanych dalej) jest zaimplementowany pewien wariant ostatniego równania rekurencyjnego. W wariancie tym definiujemy funkcję $D(i,d)=R(i,i+d-1)$, oznaczającą minimalną liczbę ruchów potrzebnych do usunięcia części ciągu zaczynającego się na pozycji i-tej i mającego d liter. Równanie rekurencyjne związane z funkcją D ma postać:

$$D(i,d)=\begin{cases}0 & \text{gdy } d\le 0\\ 1 & \text{gdy } d=1\\ \min\{D(i,k)+D(i+k,d-k)\colon 1\le k<d\} & \text{gdy } l_i\neq l_{i+d-1}\\ \min\Big(\max(D(i+1,d-2),1),\\ \quad \min\{D(i,k)+D(i+k,d-k)\colon 1\le k<d\}\Big) & \text{w przeciwnym przypadku}\end{cases}$$

Pozostaje jeszcze wrócić do pierwszego rozwiązania i zastanowić się, czy rozpatrywanie ostatniej litery ciągu $i-j$ jest istotne w rozwiązaniu tego zadania. Okazuje się tutaj, że analogiczne równanie można ułożyć rozpatrując pierwszą literę ciągu.

87.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Precz_z_palindromami/solutionreku.py.

```
n = int(input())
ll = input()
rozw = {}
def R(pocz,kon):
    if pocz>kon: return 0
    if pocz==kon: return 1
    if (pocz, kon) in rozw: return rozw[(pocz,kon)]
    odp = R(pocz, kon-1)+1
    for k in range(pocz,kon):
        if ll[k]==ll[kon]:
            odp = min(odp, R(pocz, k-1)+max(R(k+1, kon-1),1))
    mm[(pocz,kon)] = odp
    return odp

print( R(0, n-1) )
```

Źródło: ./zadania/09_Programowanie_dynamiczne/Precz_z_palindromami/solution.py.

```
n = int(input())
ll = input()
D = [[0]*n, [1] * n]
for d in range(2, n + 1):
    D.append([
        min(D[k][i] + D[d - k][i + k] for k in range(1, d))
        for i in range(n-d+1)
    ])
    for i in range(n-d+1):
        if ll[i] == ll[i + d - 1]:
            D[d][i] = min(D[d][i], D[d - 2][i + 1])
print(D[n][0])
```

87.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Precz_z_palindromami/solutionreku.cpp.

```cpp
#include <iostream>
#include <map>
using namespace std;

using Para = pair<int,int>;
map<Para, int> rozw;
string ll;

int R(int i, int j) {
    if(i>j) return 0;
    if(i==j) return 1;
    Para ij = {i,j};
    auto iter = rozw.find(ij);
    if(iter!=rozw.end()) return *iter;
    int odp = R(i, j-1)+1;
    for(int k=i; k<j; k++) {
        if(ll[k]==ll[j]) {
            int val = R(i,k-1) + max(R(k+1,j-1),1);
            if(val<odp) {
                odp = val;
            }
        }
    }
    rozw[ij] = odp;
    return odp;
}

int main() {
    int n;
    cin >> n >> ll;
    cout << R(0,n-1);
}
```

Źródło: ./zadania/09_Programowanie_dynamiczne/Precz_z_palindromami/solution.cpp.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    string ll;
    cin >> n >> ll;
    vector<vector<int>> D(n+1);
    D[0] = vector<int>(n,0);
    D[1] = vector<int>(n,1);

    for(int d=2; d<=n; d++) {
        vector<int> &curD = D[d];
        for(int i=0; i<n-d; i++) {
            int minval = 1+D[d-1][i+1];
            for(int k=2; k<=i; k++) {
                int curval = D[k][i] + D[d-k][i+k];
                if(curval<minval) {
                    minval = curval;
                }
            }
            if(ll[i]==ll[i+d-1] && minval>D[d-2][i+1]) {
                minval = D[d-2][i+1]
            }
            curD.push_back(minval);
        }
    }
    cout << D[n][0];
}
```

88 (IX) – Troskliwi ogrodnicy – Gremliny

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gremliny (IV edycja – liga algorytmiczna)
Język programowania: C++/Python

88.1 Treść zadania

W ogrodzie warzywnym przy królewskim pałacu w Bitawie założono dużą plantację *wciulistków*. Wciulistki (*Multum folia herbum*) to niewysokie zioła z wielką ilością smacznych i zdrowych listków – można z nich robić sałatki i aromatyczne napoje.

Niestety uprawa wciulistków wymaga od ekipy ogrodników wielu starań i zabiegów. Grządki trzeba podlewać i nawozić, a prócz tego należy tępić wszelkie szkodniki, zwłaszcza wciuliszki i wciulotki, którym bardzo smakują aromatyczne listki uprawianych roślin. Największym wyzwaniem jest jednak konieczność okrywania upraw po zachodzie słońca – dla ochrony przed nocnymi żarłokami: wciutoperzami i wcinockami. Gdyby nie to okrywanie, bezlitośni łupieżcy wyżarliby cenne roślinki do gołej ziemi.

Wciulistki sadzone są na grządkach w pojedynczych rzędach. Wszystkie rzędy mają tę samą długość, a krzaczki sadzone są w odległościach jednej stopy od siebie. Zdarza się niestety, że niektórych roślinek brakuje, bo ogrodnicy nie upilnowali wszystkich krzaczków.

Materiał okrywający uprawy dostępny jest w belach o rozmiarze odpowiadającym szerokości rzędu na grządce, natomiast jego długość jest nieograniczona. Okrycie roślin w jednym rzędzie musi być wykonane w formie jednego kawałka materiału, nawet jeśli pewne miejsca pod nim nie będą zajęte. Na jedną roślinkę (lub jedno puste miejsce) przypada jedna stopa długości materiału. Na przykład jeśli rząd na grządce wygląda następująco (znak **x** oznacza krzaczek, a kropka **.** to wolne miejsce):

```
..xx.xxx...
```

wtedy do jego okrycia będzie potrzebny kawałek materiału o długości 6 stóp (zaznaczony podkreśleniem): `..xx.xxx...`

Materiał służący do okrywania jest bardzo drogi i należy go używać niezwykle oszczędnie – ogrodnicy muszą zatem obliczyć, ile materiału potrzebują na okrycie wszystkich rzędów. Dobra wiadomość (również dla

wciutoperzy i wcinocków) jest taka, że królewski zarządca ogrodu dopuszcza poświęcenie (czyli nieokrywanie) pewnej liczby krzaczków w jednym lub więcej rzędów. Można pozostawić mniej nieokrytych krzaczków, niż wyznaczył to zarządca. Nadal obowiązuje jednak zasada: jeśli okrywamy krzaczki rosnące w jednym rzędzie, wtedy musi to być jeden spójny kawałek materiału.

Zadanie

Napisz program, który wczyta opis rzędów z posadzonymi krzaczkami wciulistków oraz maksymalną liczbę nieokrywanych krzaczków i obliczy niezbędną ilość materiału do zabezpieczenia ogrodu.

Opis wejścia

Pierwszy wiersz danych zawiera trzy liczby naturalne n, d oraz k oznaczające odpowiednio liczbę rzędów, ilość roślin w rzędzie (łącznie z pustymi miejscami) oraz ilość krzaczków, które można pominąć ($1 \leq n \leq 400$, $1 \leq d \leq 500$, $0 \leq k \leq 200$).

Każdy z kolejnych n wierszy zawiera ciąg d znaków bez odstępów: **x** oznacza krzaczek, zaś **.** to wolne miejsce.

Opis wyjścia

Program powinien wypisać jedną liczbę naturalną: minimalną łączną długość materiału potrzebnego na okrycie upraw.

Przykład

Dla danych wejściowych:

```
2 7 1
.x....x
xx..xx.
```

program powinien wypisać:

```
7
```

Można bowiem pominąć jeden krzaczek z pierwszego rzędu (na przykład ten na końcu) i przykryć wciulistki w następujący sposób:

```
.xx....x
xxxx..xx.
```

Jeśli natomiast zarządca okaże się rygorystyczny i nie pozwoli pominąć żadnego krzaczka (co oznacza $k = 0$):

```
2 7 0
.x....x
xx..xx.
```

wówczas nasz program powinien wypisać:

```
12
```

Okrycie w tym przypadku musi obejmować wszystkie roślinki:

```
.x....x
xx..xx.
```

Dla innych danych wejściowych:

```
3 6 3
.x.x.x
x.x.x.
.x.xxx
```

program powinien natomiast wypisać:

```
9
```

ponieważ można np. zakryć każdy rząd wciulistków taśmą długości 3, pomijając w każdym rzędzie pierwszy krzaczek:

```
.x.x.x
x.x.x.
.x.xxx
```

88.2 Rozwiązanie

Zacznijmy rozwiązanie od pojedynczej grządki. Najpierw wyznaczymy pozycję wszystkich posadzonych ziół w grządce. Przez l oznaczamy liczbę ziół w grządce, gdzie $0 \leq l \leq d$, przez $z(i)$ oznaczamy pozycję i-tego krzaczka w grządce, gdzie $0 \leq i < l$.

```
#grzadka - wczytane wcześniej
z = [i for i, c in enumerate(grzadka) if c=='x']
```

W zadaniu możemy nie przykryć k krzaczków. Nieprzykryte krzaczki będą jedynie na początku lub na końcu grządki. Minimalna długość materiału wymaganego do przykrycia wszystkich oprócz k krzaczków oznaczmy przez $p(k)$. Oczywiście $p(0) = z(l-1) - z(0) + 1$. Natomiast ogólnym wzorem jest $p(k) = \min\{z(l-1-k+i) - z(i) + 1 \colon 0 \leq i \leq k\}$, dla $k < l$ oraz $p(k) = 0$ dla $k \geq l$. Wyznaczenie $p(k)$ dla $k < l$ można zaimplementować klasycznie:

```
#z, k - wczytane wcześniej
p_k = 1+min(z[len(z)-1-k+i] - z[i] for i in range(k+1))
```

W zadaniu będziemy jednak chcieli wyznaczyć wartości $p(0), \ldots, p(k)$ i wtedy można napisać:

```
#z, k - wczytane wcześniej
p,l = [],len(z)
for j in range( min(l, k+1) ):
    p.append( 1+min(z[l-1-j+i] - z[i] for i in range(j+1)) )
```

lub zgrabniej:

```
#z - wczytane wcześniej
l = len(z)
p = [(  1+min(z[i+j] - z[i] for i in range(l-j))  )
    for j in range(l)]
p.reverse()
p.append(0)
```

Natomiast, gdy uwzględnimy ograniczenie k, czyli liczby nieprzykrytych krzaczków ziół w grządce, to otrzymujemy kod:

```
#z,k - wczytane wcześniej
l = len(z)
p = [(  1+min(z[i+j] - z[i] for i in range(l-j))  )
    for j in range(max(0, l-k), l)]
p.reverse()
p.append(0) #musimy dodawać, gdy l<=k
```

Przy powyższych założeniach, zawartość listy pozycji z oraz listy minimalnych długości materiału p, dla $k < l$, przedstawia się – dla pewnej przykładowej grządki – następująco:

$$\texttt{.xx...xxx.x...xx.xxx...}$$
$$\downarrow$$
$$z\colon 1\ 2\ 6\ 7\ 8\ 10\ 14\ 15\ 17\ 18\ 19$$
$$\downarrow$$
$$p\colon 19\ 18\ 14\ 13\ 12\ 10\ 6\ 5\ 3\ 2\ 1\ 0$$

Skoro potrafimy wyznaczyć $p(\cdot)$ dla pojedynczej grządki, to możemy wyznaczyć takie wartości dla wszystkich grządek. Przez $p_i(j)$ oznaczmy minimalną długość materiału wymaganego do przykrycia wszystkich oprócz j krzaczków w i-tej grządce, gdzie $1 \leq i \leq n$.

Wystarczy, że będziemy uwzględniać po kolei rzędy grządek i liczby nieprzykrytych krzaczków. Przez $D(i, j)$ oznaczmy długość materiału potrzebnego do zakrycia pierwszych i grządek, przy czym j krzaczków nie jest przykrytych. Dodatkowo, jeśli w i-tej grządce jest l_i krzaczków, to zakładamy, że $p_i(j) = 0$ dla $j > l_i$. Naturalnie $D(0, j) = 0$ (nie ma grządek, nie ma materiału), natomiast dla dowolnego $i > 0$ mamy $D(i, j) = \min\{D(i-1, j-s) + p_i(s)\colon 0 \leq s \leq j\}$. Gdybyśmy uwzględnili liczbę krzaczków w grządce, to równanie wyglądałoby następująco $D(i, j) = \min\{D(i-1, j-s)+p_i(s)\colon 0 \leq s \leq \min(j, l_i)\}$. Oczywiście ostatecznym rozwiązaniem całego zadania jest wartość $D(n, k)$.

Zauważmy na koniec, że $D(i, \cdot)$ jest zależne tylko od $D(i-1, \cdot)$. Z tego powodu, do implementacji tego zadania możemy skorzystać z implementacji problemu plecakowego (p. wprowadzenie do tego działu), bo rozwiązania tych problemów opierają się na podobnych równaniach rekurencyjnych.

88.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Troskliwi_ogrodnicy/solution.py.

```
n, m, k = map(int, input().split())
D = [0] * (k+1)
for _ in range(n):
    grzadka = input()
    z = [i for i, c in enumerate(grzadka) if c=='x']
    l = len(z)
    if l==0: continue #nie obsługujemy pustych grządek
    p = [1+min(z[i+j]-z[i] for i in range(l-j))
        for j in range(min(0, l-k), l)]
    p.reverse()
    p.append(0)
    D = [min(D[i-s]+p[s] for s in range(min(i,l)+1))
        for i in range(k+1)]
    #alternatywnie zamiast powyższej linii można wykonać:
    #for i in range(k, -1, -1):
    #    D[i] = min(D[i-s]+p[s] for s in range(min(i,l)+1))
print(D[-1])
```

88.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Troskliwi_ogrodnicy/solution.cpp.

```cpp
#include <iostream>
using namespace std;

int main() {
  int n, m, k;
  string grzadka;
  vector<int> z, p, D;
  cin >> n >> m >> k;
  p.resize(k+1);
  D.resize(k+1);
  for(int i=1; i<=n; i++) {
    cin >> grzadka;
    z.clear();
    for(int j=0;j<m;j++) {
      if(grzadka[j]=='x') {
        z.push_back(j);
      }
    }
    int l = z.size();
    if(l==0) continue;
    for(int j=0; j<min(l, k+1); j++) {
      p[j] = MAX_INT;
      for(int s=0; s<=j; s++) {
        p[j] = min(p[j], z[l-1-j+s]-z[s]+1)
      }
    }
    for(int j=k; j>=0; j--) {
      D[j] += p[0];
      for(int s=1; s<=min(j, l); s++) {
        D[j] = min(D[j], D[j-s]+p[s]);
      }
    }
  }
  cout << D[k];
}
```

89 (IX) – Dwa zaklęcia – Gremliny

Autor: Andrzej Dyrek, Akademia Górniczo-Hutnicza w Krakowie
Kategoria: Gremliny (II edycja – zawody algorytmiczne – etap finałowy)
Język programowania: C++/Python

89.1 Treść zadania

Dżesika uczy się w szkole dla młodych czarownic. Jak wiadomo, ważnym działem wiedzy magicznej jest numerologia, czyli nauka o nadnaturalnych własnościach liczb i ich sekwencji. Na zajęciach z tego przedmiotu jest właśnie przerabiane pewne ćwiczenie polegające na utworzeniu dwóch zaklęć numerycznych w oparciu o daną sekwencję liczb.

Nauczyciel numerologii, profesor Infinitus, pisze na tablicy ciąg dodatnich liczb naturalnych, a zadaniem każdego akolity jest wybranie dwóch podciągów przedstawionej sekwencji. Każdy podciąg musi reprezentować zaklęcie numeryczne zbudowane według ścisłych reguł: każde dwa sąsiadujące elementy muszą różnić się dokładnie o 1 lub też ich różnica musi być podzielna przez 11. Podciąg musi zawierać przynajmniej jedną liczbę.

Przy wyborze podciągu musi być zachowana oryginalna kolejność elementów z sekwencji profesora. Poza tym jeśli jakiś element sekwencji zostanie wybrany do jednego podciągu, wtedy nie może on być wykorzystany w drugim podciągu. Oto przykład sekwencji Infinitusa:

25 12 15 23 16 20 22

i przykładowe dwa podciągi-zaklęcia (czerwone, niebieskie):

25 12 15 23 16 20 22

W zaklęciu czerwonym różnica liczb 12 oraz 23 wynosi 11 (więc jest podzielna przez 11), zaś liczb 23 oraz 22 – wynosi 1. W zaklęciu niebieskim różnica $|15 - 16|$ wynosi 1.

W sekwencji profesora liczby mogą się powtarzać – na przykład w poniższej sekwencji liczba 15 występuje trzykrotnie:

10 15 13 15 14 16 15

Można tutaj wyodrębnić dwa przykładowe podciągi (fioletowy, pomarańczowy):

10 15 13 15 14 16 15

Łączna długość otrzymanych zaklęć powinna być jak największa. Dżesice bardzo pomogłoby, gdyby potrafiła wyznaczyć taką maksymalną długość dla danej sekwencji profesora. Tylko czy poradzi sobie z tym problemem?

Zadanie

Pomóż Dżesice i napisz program, który obliczy największą wartość sumy długości obydwu utworzonych zaklęć.

Twój program powinien czytać dane ze standardowego wejścia i wypisywać swój wynik na standardowe wyjście.

Opis wejścia

Pierwszy wiersz danych wejściowych zawiera liczbę naturalną n ($2 \leq n \leq 1000$) oznaczającą długość sekwencji profesora Infinitusa.

W drugim wierszu zapisane jest n liczb całkowitych z zakresu od 1 do 10^5 – są to elementy sekwencji.

Liczby w wierszu oddzielone są pojedynczymi odstępami.

Opis wyjścia

Program powinien wypisać wiersz tekstu zawierający największą możliwą łączną długość obydwu zaklęć-podciągów.

Przykład

Dla danych wejściowych:

```
7
25 12 15 23 16 20 22
```

prawidłowym wynikiem jest:

```
5
```

Zaklęcia o maksymalnej łącznej długości to $(12, 23, 22)$ oraz $(15, 16)$.

Dla danych wejściowych:

```
2
10 20
```

prawidłowym wynikiem jest:

```
2
```

W tym przykładzie można zbudować dwa jednoelementowe zaklęcia: (10) oraz (20).

Dla danych wejściowych:

```
4
1 2 13 14
```

prawidłowym wynikiem jest:

```
4
```

„Maksymalne” zaklęcia w tym przykładzie to (1) oraz $(2, 13, 14)$, $(1, 2)$ oraz $(13, 14)$ lub $(1, 2, 13)$ oraz (14).

89.2 Rozwiązanie

Niech $c_1, \ldots, c_n$ będzie ciągiem dodatnich liczb naturalnych, które profesor Infinitus zapisał na tablicy. Rozpocznijmy od pojedynczego zaklęcia. Wprowadźmy dla uproszczenia opisu predykat $d(x, y)$, który jest prawdziwy jeśli liczby x i y są kolejnymi liczbami naturalnymi (w dowolnej kolejności) lub różnica liczb x i y jest podzielna przez 11. Ten ostatni warunek jest równoważny temu, że x i y w dzieleniu przez 11 dają tę samą resztę. Matematycznie predykat można zapisać jako $d(x, y) \iff (11|x - y \vee 1 = |x - y|)$. Natomiast w Pythonie ten predykat zapiszemy jako funkcję:

```
def d(x,y):
    return x%11==y%11 or abs(x-y)==1
```

Skoro długość pojedynczego zaklęcia zależy od kolejnych liczb, to możemy wprowadzić $A(i)$ oznaczającą długość najdłuższego zaklęcia, które kończy się na i-tej liczbie. Widać, że równaniem rekurencyjnym, które opisuje wartości tej funkcji jest $A(i) = 1 + \max\{A(j) : 1 \leq j < i \wedge d(c_i, c_j)\}$, przy czym zakładamy, że $\max\{\} = 0$. Rozwiązaniem tego zadania jest $\max\{A(i) : 1 \leq i \leq n\}$. Jedna z poprawnych implementacji to:

```
#c - wczytany wcześniej
wynik, A = 0, []
for i in range(len(c)):
    w = 1 + max(
        (A[j] for j in range(i) if d(c[i],c[j])),
        default = 0
    )
    wynik = max(wynik, w)
    A.append(w)
```

Można także uwzględnić $A[0] = 0$ i wtedy:

```
#c - wczytany wcześniej
wynik, A = 0, [0]
for i in range(len(c)):
    w = 1 + max( A[j] for j in range(i+1)
        if j==0 or d(c[i],c[j-1]) )
    wynik = max(wynik, w)
    A.append(w)
```

Trzeba jednak zauważyć, że jest to algorytm o złożoności kwadratowej. Lepszym algorytmem będzie taki, który będzie zapamiętywał najlepsze wartości i tylko do nich będzie się odwoływał. Patrząc na predykat d widzimy, że będziemy musieli zapamiętywać dwa typy wyników. Pierwszym typem są wyniki zależne od reszt z dzielenia c_i przez 11. Drugim typem są wyniki zależne od wartości c_i. Potrzebujemy zatem dwóch dodatkowych kontenerów:

```
#c - wczytany wcześniej
maxMod = [0]*11 # wyniki związane z resztami z dzielenia
maxNum = {} # wyniki związane z wartościami
wynik = 0
for v in c:
    w = 1+max(maxMod[v%11], maxNum.get(v-1,0), maxNum.get(v+1,0))
    wynik = max(wynik, w)
    # zapamiętanie wyników pośrednich
    maxMod[v%11] = w
    maxNum[v] = w
```

Ponieważ znamy ograniczenia na wartości liczb, które profesor pisze na tablicy ($1 \leq c_i \leq 100\,000$), więc możemy zmienić nieznacznie implementację:

```
#c - wczytany wcześniej
maxMod = [0]*11
maxNum = [0]*100002 # potrzebujemy pozycji 0 i 100001 !
wynik = 0
for v in c:
    w = 1+max(maxMod[v%11], maxNum[v-1], maxNum[v+1])
    wynik = max(wynik, w)
    maxMod[v%11] = w
    maxNum[v] = w
```

Ta zmiana powoduje, że złożoność obliczeniowa zmodyfikowanego algorytmu jest liniowa.

Skoro potrafimy szybko stworzyć pojedyncze zaklęcie, to teraz spróbujemy stworzyć dwa zaklęcia. Musimy uwzględnić, że liczby związane z oboma zaklęciami nie mogą być wspólne. Wprowadźmy oznaczenie $\langle i, j\rangle$ na najdłuższe podwójne zaklęcie, dla którego pierwsze zaklęcie kończy się na i-tej liczbie, a drugie zaklęcie kończy się na j-tej liczbie. Przez $\langle i, 0\rangle$ lub $\langle 0, j\rangle$ rozumiemy pojedyncze zaklęcie. Wprowadźmy $B(i, j)$ oznaczającą długość $\langle i, j\rangle$. Widać, że $B(0, i) = B(i, 0) = A(i)$. Ponieważ kolejność zaklęć w podwójnych zaklęciach nie ma znaczenia, więc $B(i, j) = B(j, i)$. Zaklęcia nie mogą mieć wspólnych liczb, w szczególności ostatnie liczby nie mogą być wspólne, z tego powodu nie istnieje $\langle i, i\rangle$, a wartość $B(i, i)$ nie jest zdefiniowana.

Równanie rekurencyjne będzie związane z przedłużaniem jednego z podwójnych zaklęć o jedną liczbę. Jeśli $i < j$, to zaklęcie $\langle i, j\rangle$ powstało przez dodanie c_j do drugiej części jednego z zaklęć $\langle i, k\rangle$, gdzie $1 \leq k < j$ lub przez dodanie c_j jako drugiej części zaklęcia do zaklęcia $\langle i, 0\rangle$. Z tych przemyśleń, dla $i < j$, równanie rekurencyjne ma postać:

$$B(i, j) = 1 + \max\Big(B(i, 0), \max\big\{B(i, k) \colon 1 \leq k < j \wedge k \neq i \wedge d(c_j, c_k)\big\}\Big)$$

lub

$$B(i, j) = 1 + \max\Big\{B(i, k) \colon 0 \leq k < j \wedge k \neq i \wedge (k = 0 \vee d(c_j, c_k))\Big\}$$

Rozwiązaniem zadania jest $\max\{B(i, j) \colon 1 \leq i \leq n \wedge 1 \leq j \leq n\}$.

W rozwiązaniu rozszerzaliśmy tylko drugą część zaklęcia. Gdybyśmy jednak rozważali rozszerzanie pierwszej części zaklęcia, przy założeniu $i < j$, to zaklęcie $\langle i, j\rangle$ jest przedłużeniem przynajmniej jednego z zaklęć $\langle k, j\rangle$,

dla $1 \leq k < i$, bo definicja $\langle k, j \rangle$ gwarantuje nam, że drugie zaklęcie nie zawiera c_k. Problemem jest jednak to, że nie możemy być pewni, czy zaklęcie $\langle 0, j \rangle$ używa c_i, więc nie możemy go w łatwy sposób rozszerzyć.

Prostą implementacją tego równania rekurencyjnego jest:

```
#c - wczytany wcześniej
n,wynik = len(c),0
B = [[0]*(n+1) for _ in range(n+1)]
for j in range(1,n+1):
    for i in range(j):
        w = 1+max(B[i][k] for k in range(j+1)
            if i!=k and (k==0 or d(c[j], c[k-1])))
        B[i][j] = B[j][i] = w
        wynik = max(wynik, w)
```

Jako dodatkowe ćwiczenie, czytelnik może rozwiązać rozszerzenie problemu zaklęć, na przypadek gdy konstruujemy potrójne zaklęcia lub gdy tworzymy podwójne zaklęcia, ale pierwsza część zaklęcia jest ważniejsza (np. waga 3) od drugiej części zaklęcia (np. waga 2).

89.2.1 Python

Źródło: ./zadania/09_Programowanie_dynamiczne/Dwa_zaklecia/solution.py.

```
n, M, ans = int(input()), 11, 0
c = [0] + [int(i) for i in input().split()]
B = [[0]*(n + 1) for i in range(n + 1)]
maxNum = [0] * (10**5 + 2)
maxMod = [0] * M
for y in range(n + 1):
    maxMod, By = [0] * M, B[y]
    for i in c:
        maxNum[i] = 0
    for i in range(y):
        w, wm = c[i], c[i]%M
        maxMod[wm] = max(maxMod[wm], By[i])
        maxNum[w] = max(maxNum[w], By[i])
    for i in range(y + 1, n + 1):
        w, wm = c[i], c[i]%M
        tmp = 1 + max(maxMod[wm], maxNum[w+1],
            maxNum[w-1], By[0])
        B[y][x] = B[x][y] = tmp
        maxMod[wm] = max(maxMod[wm], tmp)
        maxNum[w] = max(maxNum[w], tmp)
        ans = max(ans, w)
print(ans)
```

89.2.2 C++

Źródło: ./zadania/09_Programowanie_dynamiczne/Dwa_zaklecia/solution.cpp.

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int M = 11;
int B[5010][5010];
int c[5010],maxNum[100010],maxMod[M];

int main() {
    int n, ans{ };
    cin >> n;
    for (int i=1; i<=n; i++) {
        cin >> c[i];
    }
    for (int i=0; i<=n; i++) {
        memset(maxNum,0,sizeof(maxNum));
        memset(maxMod,0,sizeof(maxMod));
        for (int j=1; j<i; j++) {
            pre[c[j]] = max(pre[c[j]], B[i][j]);
            mod[c[j]%M] = max(mod[c[j]%M], B[i][j]);
        }
        for (int j=i+1; j<=n; j++) {
            int tmp = 1+max({ pre[a[j]+1], pre[a[j]-1],
                mod[a[j]%M], B[i][0] });
            B[j][i] = B[i][j] = tmp;
            pre[a[j]] = max(pre[a[j]], tmp);
            mod[a[j]%M] = max(mod[a[j]%M], tmp);
            ans = max(ans, tmp);
        }
    }
    cout << ans;
}
```

Bibliografia

[1] Claudi Alsina. *Plany metra i sieci neuronowe. Teoria grafów*. RBA Coleccionables, 2022.

[2] Lech Banachowski, Krzysztof M. Diks i Wojciech Rytter. *Algorytmy i struktury danych*. Wydawnictwo Naukowe PWN, 2012.

[3] Jason R. Briggs. *Python dla dzieci. Programowanie na wesoło*. Wydawnictwo Naukowe PWN, 2023.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest i Clifford Stein. *Wprowadzenie do algorytmów*. Wydawnictwo Naukowe PWN, 2012.

[5] Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki i Jakub Radoszewski. *Przygody Bajtazara: 25 lat Olimpiady Informatycznej : wybór zadań*. Wydawnictwo Naukowe PWN, 2018.

[6] Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki i Jakub Radoszewski. *W poszukiwaniu wyzwań: wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego*. Warszawa, 2012.

[7] Adam Drozdek i Donald L. Simon. *Struktury danych w języku C*. Wydawnictwa Naukowo-Techniczne, 1996.

[8] Knuth Donald E. *Sztuka programowania*. Wydawnictwa Naukowo-Techniczne, 2002.

[9] Katarzyna Gocławska. *Elementarz młodego programisty. Język C++*. Katarzyna Gocławska, 2021.

[10] Kędzia Grażyna i Wachulec Magdalena. *Keep scratching. Scratchowe wyzwania dla zaawansowanych*. Centrum Mistrzostwa Informatycznego, 2023. URL: https://expansja.pl/wp-content/uploads/2023/11/PublikacjaCMI.pdf.

[11] Jerzy Grębosz. *Opus magnum C++11. Programowanie w języku C++*. Helion, 2020.

[12] Jerzy Grębosz. *Symfonia C++ Standard*. Edition 2000, 2008.

[13] Mark Lutz. *Python. Wprowadzenie*. Helion, 2020.

[14] Glenn Manacher. „A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String". W: *Journal of the ACM* 22.3 (1975), s. 346–351.

[15] Micha Sharir. „A strong-connectivity algorithm and its applications to data flow analysis". W: *Computers and Mathematics with Applications* 7(1) (1981), s. 67–72.

[16] O. M. Sharkovsky. „Coexistence of cycles of a continuous mapping of the line into itself". W: *Ukrainian Mathematical Journal* 16 (1964), s. 61–71.

[17] Piotr Stańczyk. *Algorytmika praktyczna*. Wydawnictwo Naukowe PWN, 2009.

[18] Bartłomiej Stasiak. „Follow That Tune: Adaptive Approach to DTW-based Query-by-Humming System". W: *Archives of Acoustics* 39 (2014), s. 467–476.

[19] Bjarne Stroustrup. *Język C++*. Wydawnictwa Naukowo-Techniczne, 2002.

[20] Al Sweigart. *Bawimy się, programując w Scratchu 3 (ebook)*. Wydawnictwo Naukowe PWN, 2021.

[21] Robert J. Wilson. *Wprowadzenie do teorii grafów*. Wydawnictwo Naukowe PWN, 1998.

[22] Niklaus Wirth. *Algorytmy + struktury danych = programy*. Wydawnictwa Naukowo-Techniczne, 2001.

[23] Lipski Witold. *Kombinatoryka dla programistów*. Wydawnictwa Naukowo-Techniczne, 2004.

ISBN 978-83-927875-9-4