Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Piotr Stańczyk
Nr albumu: 209291

Algorytmika praktyczna w konkursach informatycznych

Praca magisterska
na kierunku INFORMATYKA
w zakresie ALGORYTMIKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Diksa
Instytut Informatyki

Styczeń 2007

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Praca ta stanowi przegląd algorytmów oraz technik programowania najczęściej stosowanych podczas konkursów programistycznych, takich jak ACM ICPC czy TopCoder. Została ona stworzona na bazie „biblioteczki" algorytmicznej wykorzystywanej przez zespoły Uniwersytetu Warszawskiego podczas konkursów programistycznych. Prezentowana praca może stanowić cenny zasób wiedzy nie tylko dla uczestników konkursów informatycznych, ale również dla uczniów i studentów poszerzających swoją wiedzę algorytmiczną.

Słowa kluczowe

algebra liniowa, algorytm, geometria obliczeniowa, graf, kombinatoryka, struktury danych, teoria liczb

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

F. Theory of Computation
F.2 Analysis of algorithms and problem complexity
F.2.0 General

Tytuł pracy w języku angielskim

Algorithmic practice in computer science competitions

Spis treści

Rozdział 1

Przedmowa

Najstarszym akademickim konkursem programistycznym jest *International Collegiate Programming Contest* organizowany przez *Association for Computing Machinery* (w skrócie *ACM*, oficjalna strona konkursu — *http://icpc.baylor.edu/icpc/*). Pierwsze zawody finałowe odbyły się 2 lutego 1977 roku w Atlancie, w Stanach Zjednoczonych (stan Georgia). Na początku lat dziewięćdziesiątych w eliminacjach do tego konkursu brało udział około 400 drużyn, z których 25 kwalifikowało się co roku do finału. Dziś o 70 - 80 miejsc w finałach walczy ponad 4000 drużyn z 1600 uczelni całego świata. Wzrost liczby uczestników świadczy o ogromnej popularności konkursu, której towarzyszy zwiększający się stopień przygotowania drużyn oraz trudność zadań.

Światowej rangi konkursem, organizowanym dla uczniów szkół średnich, jest *Międzynarodowa Olimpiada Informatyczna* (strona konkursu — *http://www.ioinformatics.org/*), która została po raz pierwszy zorganizowana w 1989 roku. W konkursie tym bierze udział po czterech reprezentantów z każdego kraju, wybieranych w ramach olimpiad narodowych. Oprócz tych, cieszących się wieloletnią tradycją konkursów, powstaje wiele nowych, takich jak *TopCoder*, *Google Code Jam* czy *Imagine Cup*.

Udział w konkursach dla uczniów oraz studentów staje się nieodzownym elementem nauki algorytmiki. Nieustannie podwyższający się poziom konkursów przynosi ze sobą różne metodologie przygotowywania się do zawodów oraz specjalne techniki pisania programów. Ich celem jest minimalizacja ilości czasu potrzebnego na rozwiązanie zadania, przy jednoczesnym unikaniu jak największej liczby ewentualnych błędów. Ze względu na ograniczony czas trwania konkursów, ich uczestnicy nie mogą sobie pozwolić na tworzenie rozwiązań wszystkich zadań od zera. W wielu przypadkach muszą wykorzystywać pomysły zastosowane w podobnych zadaniach, rozwiązywanych już wcześniej, co pozwala im zaoszczędzić trochę czasu.

Książka ta jest swego rodzaju podręcznikiem do algorytmiki. Nie zawiera ona jednak wnikliwego opisu algorytmów wraz z ich analizą złożoności oraz dowodem poprawności. Jest to kolekcja implementacji różnych algorytmów, bardzo przydatnych podczas zawodów oraz zbiór zadań pozwalający na przećwiczenie ich wykorzystania w praktyce.

Znaczącą część grona czytelników stanowić będą zapewne osoby zainteresowane zwiększeniem swoich umiejętności w zakresie szybkiego implementowania rozwiązań zadań algorytmicznych w języku C++. Lektura niniejszej książki na pewno wpłynie pozytywnie na szybkość rozwiązywania zadań podczas takich konkursów jak *Olimpiada Informatyczna*. Jej użyteczność jest tym większa na konkursach typu *ACM ICPC*, podczas których dozwolone jest korzystanie z literatury. Wiele algorytmów z tej książki może zostać po prostu przepisanych do implementowanych podczas zawodów programów. Prezentowane tutaj algorytmy zostały tak napisane, aby ich adaptacja — w celu wykorzystania w różnych zadaniach — była jak

najprostsza, a jednocześnie ich kod źródłowy jak najkrótszy.

Pozycja ta może służyć nie tylko zawodnikom — w połączeniu z książką dotyczącą podstaw algorytmiki, taką jak *Wprowadzenie do algorytmów [WDA]*, może także stanowić doskonały sposób nauki „od podstaw", łączący w sobie teoretyczną analizę algorytmów z podejściem bardziej praktycznym, opisanym w tej książce. Naukę taką w istotny sposób ułatwią liczne odwołania do literatury.

Pomysł na napisanie książki narodził się podczas treningów mojego zespołu — *Warsaw Predators*, do zawodów *ACM ICPC*. W czasie wielu sesji treningowych tworzyliśmy i rozbudowywaliśmy naszą biblioteczkę algorytmiczną, w której umieszczaliśmy implementacje najczęściej wykorzystywanych w trakcie zawodów algorytmów i struktur danych. Książkę tę można określić mianem szczegółowej dokumentacji do biblioteczki algorytmicznej drużyny *Warsaw Predators*. Mam nadzieję, że materiały w niej zawarte pomogą wielu osobom w przygotowaniach do zawodów.

1.1. Struktura książki

Książka została podzielona na siedem rozdziałów, w których omówione są algorytmy z różnych dziedzin algorytmiki: teoria grafów, geometria obliczeniowa, kombinatoryka, teoria liczb, struktury danych, algorytmy tekstowe oraz algebra liniowa. W każdym z tych rozdziałów, przedstawione są różne algorytmy wraz z ich implementacjami, pochodzącymi z biblioteczki algorytmicznej. Ich omówienie ma na celu przybliżenie problematyki poruszanego zagadnienia oraz zaprezentowanie sposobu wykorzystania algorytmów w zadaniach. W celu zapoznania się z dowodami poprawności, czy dokładną analizą złożoności, należy sięgać do literatury, do której odwołania umieszczone są w różnych miejscach książki.

W dziale dodatków znajduje się rozdział dotyczący programowania dynamicznego oraz zachłannego. Z uwagi na fakt, iż z tymi dwoma technikami nie wiążą się specyficzne struktury danych ani algorytmy (rozwiązania zadań bazujących na programowaniu dynamicznym, czy zachłannym wykorzystują algorytmy z różnych dziedzin), zatem rozdział ten nie zawiera żadnych przydatnych implementacji, lecz odwołania do wielu zadań, na których można ćwiczyć umiejętność stosowania tych metod w praktyce.

W dodatkach umieszczone są również obserwacje oraz rady pochodzące od wielu światowej klasy zawodników. Lektura tego rozdziału może okazać się niezwykle cenna, gdyż zawarte w nim informacje pomagają uniknąć wielu problemów podczas przygotowań do zawodów.

W poszczególnych rozdziałach książki umieszczone są zadania związane z omawianą tematyką. Algorytmy potrzebne do ich rozwiązania są opisane, w miarę możliwości, w rozdziałach poprzedzających wystąpienie kolejnych zadań. W ten sposób, śledząc w kolejności poszczególne rozdziały książki, czytelnik będzie posiadał wiedzę potrzebną do rozwiązania wszystkich zadań. Większość z nich pochodzi z polskich konkursów informatycznych, takich jak *Olimpiada Informatyczna*, *Potyczki Algorytmiczne* czy *Pogromcy Algorytmów*. Programy stanowiące ich rozwiązania znajdują się na dołączonej płycie. Dodatkowo, na końcu książki umieszczony jest rozdział zawierający wskazówki do wszystkich tych zadań. Do każdego z nich znajduje się po kilka podpowiedzi, które mogą być pomocne podczas rozwiązywania — każda kolejna wskazówka uściśla pomysł naszkicowany w poprzedniej. Dzięki takiemu podejściu, jeśli początkowe wskazówki okażą się niewystarczające do rozwiązania zadania, można pokusić się o przeczytanie kolejnych.

Oprócz pełnych zadań, w książce znajdują się również odwołania do innych, pochodzących z internetowych serwisów, umożliwiających automatyczną weryfikację poprawności nadsyłanych rozwiązań. Czytelnik może najpierw rozwiązać samodzielnie zadanie, a następnie wysłać

je do systemu sprawdzającego, w celu zweryfikowania jego poprawności. Poniżej znajduje się lista serwisów internetowych, z których wybrane zostały zadania do tej książki:

- http://acm.sgu.ru/ — Saratov State University :: Online Contester
- http://acm.uva.es/ — Valladolid Programming Contest Site
- http://spoj.sphere.pl/ — Sphere Online Judge

Wszystkie zadania zostały podzielone na trzy kategorie: proste, średniej trudności oraz trudne. Poziom trudności jest rzeczą subiektywną. Podział użyty w książce bazuje na statystykach rozwiązań, dostępnych w wyżej wymienionych serwisach. Zadania trudne charakteryzują się stosunkowo małą liczbą prób ich rozwiązania, a wśród nich niewielka część rozwiązań okazuje się być poprawna. Proste problemy z kolei cechują się wysokim odsetkiem zaakceptowanych rozwiązań w stosunku do wszystkich nadesłanych programów.

1.2. Wymagania wstępne

Wszystkie przedstawione w tej książce algorytmy są zaimplementowane w języku C++ — jego znajomość jest nieodzowna do zrozumiałej analizy przedstawianych tu algorytmów. Konieczna jest również znajomość biblioteki Standard Template Library (w skrócie STL), której dokumentację można znaleźć na stronie *http://www.sgi.com/tech/stl/*. Implementacja algorytmów wykorzystuje różne struktury danych oraz funkcje z tej biblioteki — najważniejsze z nich to funkcje: `sort`, `swap` i `binary_search` oraz struktury danych: `vector`, `map` i `priority_queue`. Oprócz powyższych elementów, biblioteka STL zawiera wiele bardzo użytecznych narzędzi. Polecam zapoznanie się z nią, gdyż może ona w istotny sposób wpłynąć na szybkość rozwiązywania zadań algorytmicznych.

1.3. Podziękowania

Opisywana w tej książce biblioteczka algorytmiczna została stworzona wspólnie z moimi kolegami z zespołu *Warsaw Predators* — Markiem Cyganem i Marcinem Pilipczukiem. Chciałbym również podziękować Tomaszowi Idziaszkowi, którego kilka „chwytów programistycznych" znalazło się w naszej biblioteczce oraz Erykowi Kopczyńskiemu, który zgodził się na opublikowanie zbioru swoich makr używanych w konkursie *TopCoder*.

Chciałbym podziękować Tomaszowi Czajce, Andrzejowi Gąsienicy-Samkowi, Tomaszowi Malesińskiemu, Krzysztofowi Onakowi oraz Marcinowi Stefaniakowi za wyrażenie zgody na publikację ich „dobrych rad", których udzielili swoim młodszym kolegom z Uniwersytetu Warszawskiego, biorącym udział w konkursie ACM ICPC. Wskazówki te zostały umieszczone w dziale *Dodatki*.

Jednym z wyzwań, podczas gromadzenia materiałów do książki, był proces wybierania odpowiednich zadań z różnych serwisów internetowych. Liczba dostępnych zadań jest ogromna (w momencie pisania tego tekstu, serwis *Valladolid Programming Contest Site* zawierał ponad dwa tysiące zadań). Chciałbym podziękować Jakubowi Radoszewskiemu za udzielenie cennych wskazówek dotyczących wyboru zadań.

Wielkie podziękowania należą się prof. Krzysztofowi Diksowi — naszemu wykładowcy i wielkiemu przyjacielowi, który wytrwale opiekował się nami i stwarzał warunki pozwalające na efektywne przygotowania naszych drużyn do zawodów. Niniejsza książka powstała pod jego opieką i nadzorem.

1.4. Nagłówki

Wszystkie programy w tej książce są zaimplementowane w języku C++, z wykorzystaniem biblioteki STL. Ponieważ podczas zawodów bardzo istotne jest, aby kody źródłowe implementowanych programów były jak najkrótsze, dlatego wielu zawodników stosuje pewne skróty dla najczęściej występujących w programach konstrukcji językowych. Podczas zawodów ACM ICPC, w ciągu pięciogodzinnych sesji, do rozwiązania jest 7 do 10 zadań, zatem opłaca się na początku konkursu przepisać najważniejsze instrukcje, a następnie wykorzystywać je we wszystkich pisanych programach. Instrukcje te nazywamy mianem nagłówków — zawierają one zarówno listę najczęściej dołączanych do programów bibliotek, jak i często wykorzystywane makra. Listing 1.1 przedstawia zbiór podstawowych nagłówków pochodzących z biblioteczki algorytmicznej mojego zespołu wraz z komentarzami opisującymi ich wykorzystanie. Na listingu 1.2 przedstawione zostały te same nagłówki po usunięciu komentarzy.

Listing 1.1: Nagłówki z komentarzami

```
01 #include <cstdio>
02 #include <iostream>
03 #include <algorithm>
04 #include <string>
05 #include <vector>
06 using namespace std;
// Dwa z najczęściej używanych typów o długich nazwach - ich skrócenie jest bardzo
// istotne
07 typedef vector<int> VI;
08 typedef long long LL;
// W programach bardzo rzadko można znaleźć w pełni zapisaną instrukcję pętli.
// Zamiast niej, wykorzystywane są trzy następujące makra:
// FOR - pętla zwiększająca zmienną x od b do e włącznie
09 #define FOR(x, b, e) for(int x = b; x <= (e); ++x)
// FORD - pętla zmniejszająca zmienną x od b do e włącznie
10 #define FORD(x, b, e) for(int x = b; x >= (e); --x)
// REP - pętla zwiększająca zmienną x od 0 do n. Jest ona bardzo
// często wykorzystywana do konstruowania i przeglądania struktur danych
11 #define REP(x, n) for(int x = 0; x < (n); ++x)
// Makro VAR(v,n) deklaruje nową zmienną o nazwie v oraz typie i wartości
// zmiennej n. Jest ono często wykorzystywane podczas operowania na iteratorach
// struktur danych z biblioteki STL, których nazwy typów są bardzo długie
12 #define VAR(v, n) __typeof(n) v = (n)
// ALL(c) reprezentuje parę iteratorów wskazujących odpowiednio na pierwszy i
// za ostatni element w strukturach danych STL. Makro to jest bardzo przydatne
// chociażby w przypadku korzystania z funkcji sort, która jako parametry
// przyjmuje parę iteratorów reprezentujących przedział elementów do posortowania.
13 #define ALL(c) (c).begin(), (c).end()
// Poniższe makro służy do wyznaczania rozmiaru struktur danych STL. Używa się go
// w programach, zamiast pisać po prostu x.size() z uwagi na fakt, iż wyrażenie
// x.size() jest typu unsigned int i w przypadku porównywania z typem
// int, w procesie kompilacji generowane jest ostrzeżenie.
14 #define SIZE(x) ((int)(x).size())
// Bardzo pożyteczne makro, służące do iterowania po wszystkich elementach w
```

Listing 1.1: (c.d. listingu z poprzedniej strony)

```
// strukturach danych STL.
#define FOREACH(i, c) for(VAR(i, (c).begin()); i != (c).end(); ++i)
// Skrót - zamiast pisać push_back podczas wstawiania elementów na koniec
// struktury danych, takiej jak vector, wystarczy napisać PB
#define PB push_back
// Podobnie - zamiast first będziemy pisali po prostu ST
#define ST first
// a zamiast second - ND.
#define ND second
```

Listing 1.2: Nagłówki bez komentarzy

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
typedef vector<int> VI;
typedef long long LL;
#define FOR(x, b, e) for(int x = b; x <= (e); ++x)
#define FORD(x, b, e) for(int x = b; x >= (e); --x)
#define REP(x, n) for(int x = 0; x < (n); ++x)
#define VAR(v, n) __typeof(n) v = (n)
#define ALL(c) (c).begin(), (c).end()
#define SIZE(x) ((int)(x).size())
#define FOREACH(i, c) for(VAR(i, (c).begin()); i != (c).end(); ++i)
#define PB push_back
#define ST first
#define ND second
```

Oprócz nagłówków podstawowych, istnieje również wiele innych skrótów, które okazują się pomocne podczas rozwiązywania zadań. Omawiana w tej książce biblioteczka, oprócz nagłówków podstawowych, zawiera również dodatkowe makra, umieszczane tylko w tych programach, które z nich korzystają. Ich lista przedstawiona jest na listingu 1.3. W prezentowanych w książce programach będziemy odwoływali się zarówno do tych podstawowych, jak i dodatkowych nagłówków, bez informowania o konieczności dopisywania ich definicji do implementowanych programów. Dzięki takiemu podejściu, kody źródłowe przedstawianych w książce algorytmów są krótsze, a czytelnik nie musi analizować za każdym razem powtarzających się fragmentów kodu.

Listing 1.3: Dodatkowe nagłówki

```
01 #include <complex>
02 #include <iterator>
03 #include <set>
04 #include <bitset>
05 #include <map>
06 #include <stack>
07 #include <list>
08 #include <queue>
09 #include <deque>
// Stała INF jest wykorzystywana jako reprezentacja nieskończoności. Ma ona
// wartość 1000000001, a nie 2147483647 (największa wartość typu int) ze
// względu na dwa fakty - prosty zapis, oraz brak przepełnienia wartości zmiennej
// w przypadku dodawania dwóch nieskończoności do siebie
// ((int) 2147483647 + (int) 2147483647 = -2).
10 const int INF = 1000000001;
// Stała EPS jest używana w wielu algorytmach geometrycznych do porównywania
// wartości bliskich zera (w zadaniach tego typu pojawia się wiele problemów
// związanych z błędami zaokrągleń)
11 const double EPS = 10e-9;
// Skrócone nazwy różnych typów i operatorów o długich nazwach
12 typedef vector<VI> VVI;
13 typedef vector<LL> VLL;
14 typedef vector<double> VD;
15 typedef vector<string> VS;
16 typedef pair<int, int> PII;
17 typedef vector<PII> VPII;
18 #define PF push_front
19 #define MP make_pair
```

Znane są przypadki, że niektórzy zawodnicy tworzyli przy użyciu makr własne, specyficzne języki programowania, pozwalające na szybkie pisanie programów. Takie podejście wymaga niestety przepisania znacznej ilości kodu, zanim przystąpi się do właściwego rozwiązywania problemu. W przypadku konkursów organizowanych przez Internet podejście takie okazuje się jednak bardzo wygodne. W dodatku A znajduje się przykładowe „środowisko" programistyczne, wykorzystywane w konkursie *TopCoder* przez Eryka Kopczyńskiego. Jest ono dość rozbudowane, dlatego też w innych konkursach (takich jak *ACM ICPC*) okazuje się nieprzydatne.

Rozdział 2

Algorytmy grafowe

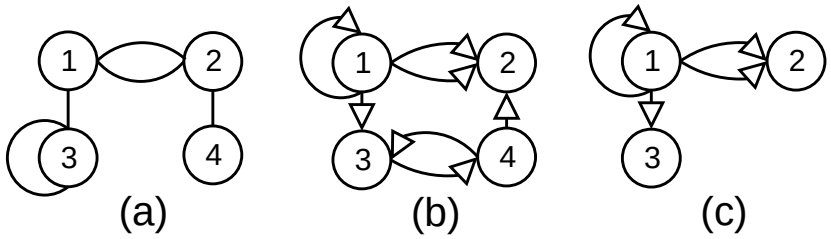
Literatura
[WDA] - 23
[ASD] - 7
[MD] - 3.2, 6

Graf G definiowany jest jako para (V, E), gdzie V to zbiór wierzchołków, natomiast E to multizbiór krawędzi łączących wierzchołki z V. Dla grafu $G = (V, E)$ z czterema wierzchołkami oraz pięcioma krawędziami, przedstawionego na rysunku 2.1.a, zbiór $V = \{1, 2, 3, 4\}$, natomiast $E = \{(1, 2), (1, 2), (1, 3), (2, 4), (3, 3)\}$. Krawędzie grafu mogą być skierowane (jak na rysunku 2.1.b), bądź nieskierowane (rysunek 2.1.a). Zarówno z wierzchołkami, jak i krawędziami można wiązać dodatkowe informacje, takie jak waga, kolor czy długość. Grafy znajdują wiele odpowiedników w świecie rzeczywistym, co sprawia, że są one bardzo przydatnym narzędziem podczas rozwiązywania praktycznych problemów algorytmicznych.

Rozpatrzmy dla przykładu zbiór skrzyżowań w mieście połączonych drogami. Naturalnym jest utożsamienie wierzchołków grafu ze skrzyżowaniami, a krawędzi grafu z drogami. Drogi mogą być jednokierunkowe (równoważne krawędziom skierowanym) lub dwukierunkowe (równoważne krawędziom nieskierowanym), mają określoną długość (którą możemy również przypisać krawędziom). Skrzyżowania natomiast mogą mieć określone położenie geograficzne, które możemy przechowywać w reprezentujących je wierzchołkach. Na potrzeby tego rozdziału wprowadzimy kilka często używanych pojęć:

Definicja 2.0.1 Pętla *to dowolna krawędź (skierowana lub nieskierowana), prowadząca do wierzchołka, z którego wychodzi. Innymi słowy, jest to krawędź postaci* (v, v), $v \in V$.

Definicja 2.0.2 Multikrawędzie *to co najmniej dwie krawędzie prowadzące między tymi samymi wierzchołkami. W przypadku grafów nieskierowanych są to krawędzie łączące tę samą parę wierzchołków, natomiast w przypadku grafów skierowanych — wychodzące z tego samego wierzchołka oraz wchodzące do tego samego wierzchołka.*



Rysunek 2.1: **(a)** graf nieskierowany — krawędzie łączące wierzchołki 1 i 2 są multikrawędziami, natomiast $(3, 3)$ to pętla. **(b)** Graf skierowany zawierający pętlę $(1, 1)$ oraz multikrawędzie $(1, 2)$. **(c)** Graf indukowany przez graf z rysunku **(b)** przez zbiór wierzchołków $\{1, 2, 3\}$

Definicja 2.0.3 Podgraf $G' = (V', E')$ *grafu* $G = (V, E)$ *jest to taki graf, dla którego* $V' \subseteq V$, *oraz* $E' \subseteq E$.

Definicja 2.0.4 Grafem indukowanym $G' = G[V']$ *przez zbiór* V' *z grafu* $G = (V, E)$, $V' \subseteq V$ *nazywamy podgraf grafu* G, *którego zbiorem wierzchołków jest* V', *natomiast do zbioru krawędzi należą wszystkie krawędzie* $(u, v) \in E$, *dla których* $u, v \in V'$.

Krawędź, łączącą wierzchołki u i v, będziemy oznaczać przez (u, v) (ewentualnie $u \to v$), natomiast ścieżkę prowadzącą z wierzchołka u do wierzchołka v przez $u \rightsquigarrow v$. Cyklem będziemy nazywali ścieżkę mającą swój początek i koniec w tym samym wierzchołku ($u \rightsquigarrow u$). Szczególnym przypadkiem cyklu jest cykl prosty, który przechodzi przez każdy wierzchołek grafu co najwyżej raz. Odwołując się do poszczególnych wierzchołków grafu, czasem będzie używane pojęcie ich stopnia. Stopień wejściowy wierzchołka, jest to liczba krawędzi do niego wchodzących, stopień wyjściowy natomiast, to liczba krawędzi wychodzących z wierzchołka.

W wielu miejscach — w szczególności podczas omawiania złożoności algorytmów, będziemy odwoływać się do liczby wierzchołków oraz krawędzi w grafie. W sytuacjach, w których jednoznacznie z kontekstu będzie wynikało o jaki graf chodzi, liczbę wierzchołków oznaczać będziemy przez n, natomiast liczbę krawędzi przez m.

W aktualnym rozdziale omówionych jest wiele algorytmów związanych z teorią grafów. Przedstawione zostaną między innymi metody przeszukiwania grafów, topologiczne sortowanie wierzchołków, sprawdzanie czy graf jest acykliczny, wyznaczanie maksymalnego przepływu oraz wiele innych zagadnień. Zanim jednak przejdziemy do analizy tych algorytmów, musimy najpierw omówić sposób reprezentowania grafu w programach.

2.1. Reprezentacja grafu

Literatura
[WDA] - 23.1
[ASD] - 1.5.3, 7
[KDP] - 2.1
[MD] - 3.3

Graf będziemy reprezentować przy użyciu sparametryzowanej struktury (ang. template) `Graph<V,E>`, gdzie `V` jest typem określającym dodatkowe informacje przechowywane w wierzchołkach, natomiast `E` jest typem określającym dodatkowe informacje przechowywane w krawędziach grafu. W ten sposób, jeśli chcemy stworzyć graf modelujący sieć dróg w mieście, to wystarczy wzbogacić strukturę `V` o informacje związane ze skrzyżowaniami, natomiast strukturę `E` — z ulicami.

Implementacja grafu udostępnia dwa operatory — `Graph<V,E>::EdgeD(int, int, E)`, oraz `Graph<V,E>::EdgeU(int, int, E)`, służące odpowiednio do dodawania do grafu krawędzi skierowanych oraz nieskierowanych (ang. directed — skierowane, undirected — nieskierowane). Pierwszy oraz drugi parametr tych funkcji określają numery wierzchołków połączonych dodawaną krawędzią, natomiast ostatni parametr definiuje dodatkowe informacje związane z tą krawędzią. Dla większości algorytmów, korzystających ze struktury `Graph<V,E>`, te dwa operatory są w zupełności wystarczające, ale w razie potrzeby można w prosty sposób wzbogacić implementację o nowe operacje — chociażby funkcję usuwającą krawędzie z grafu.

Listing 2.1 prezentuje implementację struktury `Graph<V,E>` wraz z komentarzami wyjaśniającymi przyjęte rozwiązania. Implementacja ta wykorzystuje bibliotekę STL (dokładniej, kontener `vector` do reprezentowania listy wierzchołków oraz krawędzi).

Listing 2.1: Implementacja struktury `Graph<V,E>`

```
01 template <class V, class E> struct Graph {
// Typ krawędzi (Ed) dziedziczy po typie zawierającym dodatkowe informacje
// związane z krawędzią (E). Zawiera on również pole v, określające numer
// wierzchołka, do którego prowadzi krawędź. Zaimplementowany konstruktor
// pozwala na skrócenie zapisu wielu funkcji korzystających ze struktury grafu.
02   struct Ed : E {
03     int v;
04     Ed(E p, int w) : E(p), v(w) { }
05   };
// Typ wierzchołka (Ve) dziedziczy po typie zawierającym dodatkowe informacje
// z nim związane (V) oraz po wektorze krawędzi. To drugie dziedziczenie może
// wydawać się na pierwszy rzut oka stosunkowo dziwne, lecz jest ono przydatne -
// umożliwia łatwe iterowanie po wszystkich krawędziach wychodzących z
// wierzchołka v: FOREACH(it, g[v])
06   struct Ve : V, vector<Ed> { };
// Wektor wierzchołków w grafie
07   vector<Ve> g;
// Konstruktor grafu - przyjmuje jako parametr liczbę wierzchołków
08   Graph(int n = 0) : g(n) { }
// Funkcja dodająca do grafu nową krawędź skierowaną z wierzchołka b do e,
// zawierającą dodatkowe informacje określone przez zmienną d.
09   void EdgeD(int b, int e, E d = E()) {
10     g[b].PB(Ed(d, e));
11   }
// Funkcja dodająca do grafu nową krawędź nieskierowaną, łączącą wierzchołki
// b i e oraz zawierającą dodatkowe informacje określone przez zmienną
// d. Krawędź nieskierowana jest reprezentowana przez dwie krawędzie
// skierowane - jedną prowadzącą z wierzchołka b do wierzchołka e, oraz
// drugą z wierzchołka e do b. Struktura E w grafach nieskierowanych
// musi dodatkowo zawierać element int rev. Dla danej krawędzi skierowanej
// (b,e), pole to przechowuje pozycję krawędzi (e,b) na liście incydencji
// wierzchołka e. Dzięki temu, dla dowolnej krawędzi w grafie w czasie stałym
// można znaleźć krawędź o przeciwnym zwrocie.
12   void EdgeU(int b, int e, E d = E()) {
13     Ed eg(d, e);
14     eg.rev = SIZE(g[e]) + (b == e);
15     g[b].PB(eg);
16     eg.rev = SIZE(g[eg.v = b]) - 1;
17     g[e].PB(eg);
18   }
19 };
```

W kolejnych rozdziałach dotyczących teorii grafów, przedstawione zostaną implementacje różnych algorytmów wykorzystujących strukturę `Graph<V,E>`. Wszystkie funkcje realizujące te algorytmy są metodami grafu, a co za tym idzie, ich treść musi zostać umieszczona w obrębie struktury `Graph<V,E>`. Faktu tego w dalszej części książki nie będziemy więcej przywoływać — zdajemy się na dobrą pamięć czytelnika.

Jako zadanie „rozgrzewkowe”, napiszemy prostą funkcję wypisującą strukturę grafu na standardowe wyjście. Na początku każdego wiersza opisu powinien znaleźć się numer kolejnego wierzchołka (zaczynając numerację od 0), a następnie lista wierzchołków, z którymi ten wierzchołek jest połączony krawędzią. Przykładowa implementacja tej funkcji przedstawiona jest na listingu 2.2.

Listing 2.2: Implementacja funkcji **void** `Graph<V,E>::Write()`

```
1 void Write() {
// Dla wszystkich wierzchołków w grafie zaczynając od 0...
2   REP(x, SIZE(g)) {
// Wypisz numer wierzchołka
3     cout << x << ":";
// Dla każdej krawędzi wychodzącej z przetwarzanego wierzchołka o numerze
// x, wypisz numer wierzchołka, do którego ona prowadzi
4     FOREACH(it, g[x]) cout << " " << it->v;
5     cout << endl;
6   }
7 }
```

Załóżmy, że w pliku znajduje się opis grafu skierowanego w następującym formacie: w pierwszym wierszu zapisane są dwie liczby naturalne — n i m, oznaczające odpowiednio liczbę wierzchołków, oraz krawędzi w grafie (wierzchołki numerowane są od 0 do $n-1$). W kolejnych m wierszach umieszczone są po dwie liczby naturalne b i e, oznaczające, że z wierzchołka o numerze b prowadzi krawędź skierowana do wierzchołka o numerze e. Chcemy napisać program, który przekonwertuje opis tego grafu na format, jaki jest generowany przez funkcję **void** `Graph<V,E>::Write()`.

Dla następującego opisu wejściowego:

```
5 8
0 1
3 4
4 1
4 0
1 2
2 4
3 2
4 3
```

Program wygeneruje następujący wynik:

```
0: 1
1: 2
2: 4
3: 4 2
4: 1 0 3
```

Rozwiązanie jest proste — wystarczy wykorzystać strukturę `Graph<V,E>`, na podstawie opisu z pliku skonstruować odpowiedni graf, a następnie wypisać jego strukturę przy użyciu funkcji **void** `Graph::Write()`. Główna część programu, realizującego to zadanie, przedstawiona jest na listingu 2.3. Pominięte w nim zostały standardowe nagłówki pochodzącego z biblioteczki. Właśnie w ten sposób, w dalszej części książki, będą prezentowane przykładowe programy. Z uwagi na fakt, iż omówiony tu przykład jest pierwszym kompletnym programem, jaki do tej pory udało nam się stworzyć, zatem na listingu 2.4 zamieszczony jest pełny kod źródłowy tego programu. W dalszych rozdziałach książki po pełny kod źródłowy programów należy sięgać do załączonej płyty.

Listing 2.3: Główna część programu służącego do konwertowania reprezentacji grafu

```
// Zarówno dla wierzchołków, jak i dla krawędzi nie potrzebne są żadne dodatkowe
// informacje
struct Empty { };
int main() {
   int n, m, b, e;
// Wczytaj liczbę wierzchołków i krawędzi w grafie
   cin >> n >> m;
// Skonstruuj graf o odpowiednim rozmiarze, nie zawierający dodatkowych
// informacji dla wierzchołków ani krawędzi
   Graph<Empty, Empty> gr(n);
   REP(x, m) {
// Wczytaj początek i koniec kolejnej krawędzi
      cin >> b >> e;
// Dodaj do grafu krawędź skierowaną z wierzchołka b do e
      gr.EdgeD(b, e);
   }
// Wypisz graf
   gr.Write();
   return 0;
}
```

Listing 2.4: Pełny kod źródłowy programu konwertującego reprezentację grafu

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <string>
#include <vector>
using namespace std;
typedef vector<int> VI;
typedef long long LL;
#define FOR(x, b, e) for(int x = b; x <= (e); ++x)
#define FORD(x, b, e) for(int x = b; x >= (e); --x)
#define REP(x, n) for(int x = 0; x < (n); ++x)
#define VAR(v, n) __typeof(n) v = (n)
#define ALL(c) (c).begin(), (c).end()
#define SIZE(x) ((int)(x).size())
#define FOREACH(i, c) for(VAR(i, (c).begin()); i != (c).end(); ++i)
#define PB push_back
#define ST first
#define ND second
template <class V, class E> struct Graph {
   struct Ed : E {
      int v;
      Ed(E p, int w) : E(p), v(w) { }
   };
   struct Ve : V, vector<Ed> { };
```

Listing 2.4: (c.d. listingu z poprzedniej strony)

```
25    vector<Ve> g;
26    Graph(int n = 0) : g(n) { }
27    void EdgeD(int b, int e, E d = E()) {
28       g[b].PB(Ed(d, e));
29    }
30    void Write() {
31       REP(x, SIZE(g)) {
32          cout << x << ":";
33          FOREACH(it, g[x]) cout << " " << it->v;
34          cout << endl;
35       }
36    }
37 };
38 struct Empty { };
39 int main() {
40    int n, m, b, e;
41    cin >> n >> m;
42    Graph<Empty, Empty> gr(n);
43    REP(x, m) {
44       cin >> b >> e;
45       gr.EdgeD(b, e);
46    }
47    gr.Write();
48    return 0;
49 }
```

2.2. Przeszukiwanie grafu wszerz

Literatura
[WDA] - 23.2
[ASD] - 1.5.3
[KDP] - 2.3

BFS — czyli przeszukiwanie grafu wszerz (ang. breadth-first search), jest metodą systematycznego badania struktury grafu. Zakłada ona, że dany jest graf $G = (V, E)$ (skierowany bądź nieskierowany), oraz wyróżniony wierzchołek s, zwany źródłem przeszukiwania. Dla każdego wierzchołka $v \in V$, algorytm BFS oblicza odległość $t(v)$ od wierzchołka s (rozumianą jako najmniejszą liczbę krawędzi na ścieżce od wierzchołka s do v). Wyznaczany jest również wierzchołek $s(v)$ połączony krawędzią z v, przez który przechodzi najkrótsza ścieżka ze źródła wyszukiwania do v. Zasada działania algorytmu BFS jest prosta — na początku za odwiedzony wierzchołek uważa się tylko s — wyznaczona dla niego odległość to 0. Następnie przetwarzane są wierzchołki v, dla których odległość została już wyznaczona, w kolejności niemalejących odległości — dla każdego nie odwiedzonego sąsiada u wierzchołka v, przypisywane są wartości $t(u) = t(v) + 1$, $s(u) = v$. Na skutek wykonania algorytmu BFS, konstruowany jest las przeszukiwania wszerz, w skład którego wchodzą wszystkie wierzchołki osiągalne ze źródła wyszukiwania, oraz krawędzie konstruujące najkrótsze ścieżki (reprezentowane przez zmienne $s(v)$). Ze względu na fakt, iż nie wszystkie krawędzie grafu G muszą znajdować się w lesie przeszukiwania BFS, istnieje możliwość klasyfikacji krawędzi grafu G. Rozróżniane są następujące typy krawędzi, których definicja ma sens w przypadku rozpatrywania konkret-

nego lasu przeszukiwania (definicje te są używane nie tylko w ramach przeszukiwania wszerz):

Definicja 2.2.1 Krawędź drzewowa, *jest to krawędź grafu G należąca do wyznaczonego lasu przeszukiwań.*

Definicja 2.2.2 Krawędź niedrzewowa, *jest to krawędź grafu G, która nie należy do wyznaczonego lasu przeszukiwań.*

Definicja 2.2.3 Krawędź powrotna, *jest to krawędź niedrzewowa, która prowadzi z wierzchołka v do jego przodka w lesie przeszukiwania.*

Definicja 2.2.4 Krawędź w przód, *jest to krawędź niedrzewowa, która prowadzi z wierzchołka v do jego potomka w lesie przeszukiwania.*

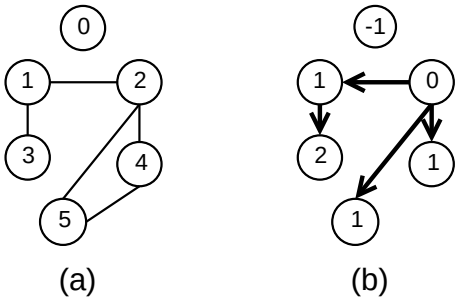
Definicja 2.2.5 Krawędź poprzeczna, *jest to krawędź niedrzewowa, która prowadzi z wierzchołka v do wierzchołka nie będącego ani jego poprzednikiem, ani następnikiem w wyznaczonym lesie przeszukiwań.*

W przypadku przeszukiwania wszerz grafów nieskierowanych nie zawierających multikrawędzi, krawędzie w przód oraz krawędzie powrotne nie występują, natomiast dla grafów skierowanych nie występują krawędzie w przód. W zależności od rodzaju stosowanego przeszukiwania, krawędzie poszczególnych klas mogą mieć ciekawe własności, które są wykorzystywane w różnego rodzaju algorytmach. Najciekawszą własnością, w przypadku przeszukiwania grafu wszerz, jest fakt, iż dla każdej krawędzi $(u, v) \in E$ zachodzi $d(u) = d(v)$ lub $d(u) = d(v) \pm 1$. Dzięki tej własności, algorytm BFS może być wykorzystywany do wyznaczania najkrótszych ścieżek między zadaną parą wierzchołków.

Algorytm BFS zrealizowany został jako funkcja **void** `Graph<V,E>::Bfs(int)`, która przyjmuje jeden parametr — numer wierzchołka będącego źródłem przeszukiwania. Implementacja wymaga wzbogacenia struktury wierzchołków o dwie dodatkowe zmienne — **int** `t` oraz **int** `s`. Po wykonaniu algorytmu na grafie, zmienna **int** `t` jest równa odległości wierzchołka od źródła wyszukiwania, natomiast zmienna **int** `s` zawiera numer wierzchołka, z którego prowadzi znaleziona najkrótsza ścieżka. Zmienne **int** `s` wyznaczają tzw. las przeszukiwania grafu wszerz — przykład takiego lasu przedstawiony jest na rysunku 2.2. Implementacja funkcji **void** `Graph<V,E>::Bfs(int)` znajduje się na listingu 2.5.

Listing 2.5: Implementacja funkcji **void** `Graph<V,E>::Bfs(int)`

```
// Po wykonaniu algorytmu BFS, pole int t wierzchołka zawiera odległość od
// źródła (-1 w przypadku, gdy wierzchołek jest nieosiągalny ze źródła), pole
// int s zawiera numer ojca w drzewie BFS (dla wierzchołka będącego źródłem
// wyszukiwania oraz wierzchołków nieosiągalnych jest to -1)
01 void Bfs(int s) {
// Dla każdego wierzchołka w grafie ustawiana jest początkowa wartość zmiennych
// t oraz s na -1. Źródło wyszukiwania ma czas równy 0
02   FOREACH(it, g) it->t = it->s = -1;
03   g[s].t = 0;
// Algorytm BFS jest realizowany przy użyciu kolejki FIFO, do której wstawiane
// są kolejne wierzchołki oczekujące na przetworzenie
04   int qu[SIZE(g)], b, e;
// Do kolejki wstawione zostaje źródło
05   qu[b = e = 0] = s;
```



Rysunek 2.2: **(a)** nieskierowany graf o sześciu wierzchołkach: $0, 1, 2, 3, 4, 5$, **(b)** las BFS dla grafu z rysunku **(a)** i wierzchołka źródłowego 2 (w wierzchołkach wpisane są wyznaczone czasy). Krawędź $(4, 5)$ jest jedyną krawędzią niedrzewową — jest ona jednocześnie krawędzią poprzeczną.

Listing 2.5: (c.d. listingu z poprzedniej strony)

```
// Dopóki w kolejce są jakieś wierzchołki...
06    while (b <= e) {
07       s = qu[b++];
// Dla każdej krawędzi wychodzącej z aktualnie przetwarzanego wierzchołka,
// jeśli wierzchołek do którego ona prowadzi nie był jeszcze wstawiony do
// kolejki, wstaw go i wyznacz wartości zmiennych int t i int s
08       FOREACH(it, g[s]) if (g[it->v].t == -1) {
09          g[qu[++e] = it->v].t = g[s].t + 1;
10          g[it->v].s = s;
11       }
12    }
13 }
```

Dla grafu przedstawionego na rysunku 2.2.a, po wykonaniu funkcji `Graph<V,E>::Bfs(2)`, wartości zmiennych wyliczone przez algorytm, przedstawione są na listingu 2.6, natomiast na listingu 2.7 umieszczony jest kod źródłowy programu użytego do wygenerowania tego wyniku.

Algorytm przeszukiwania grafów metodą BFS ma złożoność czasową $O(n + m)$. Wynika to z faktu, iż każdy wierzchołek w grafie odwiedzany jest co najwyżej raz, a podczas jego przetwarzania, wszystkie krawędzie z niego wychodzące są analizowane jednokrotnie (w przypadku krawędzi nieskierowanych są one analizowane w obie strony niezależnie).

Listing 2.6: Przykład użycia algorytmu BFS dla grafu z rysunku 2.2 i wierzchołka źródłowego 2.

```
Wierzcholek 0: czas = -1, ojciec w lesie BFS = -1
Wierzcholek 1: czas = 1, ojciec w lesie BFS = 2
Wierzcholek 2: czas = 0, ojciec w lesie BFS = -1
Wierzcholek 3: czas = 2, ojciec w lesie BFS = 1
Wierzcholek 4: czas = 1, ojciec w lesie BFS = 2
Wierzcholek 5: czas = 1, ojciec w lesie BFS = 2
```

Listing 2.7: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.6. Pełny kod źródłowy programu znajduje się w pliku `bfs_str.cpp`

```
// Krawędzie grafu nieskierowanego wymagają dodatkowego pola int rev
01 struct Ve {
02    int rev;
03 };
// Wzbogacenie wierzchołków musi zawierać pola wymagane przez algorytm BFS
04 struct Vs {
05    int t, s;
06 };
07 int main() {
08    int n, m, s, b, e;
// Wczytaj liczbę wierzchołków, krawędzi oraz numer wierzchołka startowego
09    cin >> n >> m >> s;
10    Graph<Vs, Ve> g(n);
// Dodaj do grafu wszystkie krawędzie
11    REP(x,m) {
12       cin >> b >> e;
13       g.EdgeU(b, e);
14    }
// Wykonaj algorytm BFS
15    g.Bfs(s);
// Wypisz wynik
16    REP(x, n) cout << "Wierzcholek " << x << ": czas = " << g.g[x].t <<
17       ", ojciec w lesie BFS = " << g.g[x].s << endl;
18    return 0;
19 }
```

Zadanie: Mrówki i biedronka

Pochodzenie: VIII Olimpiada Informatyczna

Rozwiązanie: `ants.cpp`

Jak wiadomo, mrówki potrafią hodować mszyce. Mszyce wydzielają słodką rosę miodową, którą spijają mrówki. Mrówki zaś bronią mszyc przed ich największymi wrogami — biedronkami.

Na drzewie obok mrowiska znajduje się właśnie taka hodowla mszyc. Mszyce żerują na liściach oraz w rozgałęzieniach drzewa. W niektórych z tych miejsc znajdują się również mrówki patrolujące drzewo. Dla ustalenia uwagi, mrówki są ponumerowane od jeden w górę. Hodowli zagraża biedronka, która zawsze siada na drzewie tam, gdzie są mszyce, czyli na liściach lub w rozgałęzieniach. W chwili, gdy gdzieś na drzewie usiądzie biedronka, mrówki patrolujące drzewo ruszają w jej stronę, aby ją przegonić. Kierują się przy tym następującymi zasadami:

- z każdego miejsca na drzewie (liścia lub rozgałęzienia) można dojść do każdego innego miejsca (bez zawracania) tylko na jeden sposób; każda mrówka wybiera właśnie taką drogę do miejsca lądowania biedronki,
- jeżeli w miejscu lądowania biedronki znajduje się mrówka, biedronka natychmiast odlatuje,

- jeżeli na drodze, od aktualnego położenia mrówki do miejsca lądowania biedronki, znajdzie się inna mrówka, to ta położona dalej od biedronki kończy wędrówkę i zostaje w miejscu swojego aktualnego położenia,
- jeżeli dwie lub więcej mrówek próbuje wejść na to samo rozgałęzienie drzewa, to robi to tylko jedna mrówka — ta z najmniejszym numerem, a reszta mrówek pozostaje na swoich miejscach (liściach lub rozgałęzieniach),
- mrówka, która dociera do miejsca lądowania biedronki, przegania ją i pozostaje w tym miejscu.

Biedronka jest uparta i znowu ląduje na drzewie. Wówczas mrówki ponownie ruszają, aby przegonić intruza.

Dla uproszczenia przyjmujemy, że przejście gałązki łączącej liść z rozgałęzieniem lub łączącej dwa rozgałęzienia, zajmuje wszystkim mrówkom jednostkę czasu.

Zadanie

Napisz program, który:

- wczyta opis drzewa, początkowe położenia mrówek oraz miejsca, w których kolejno siada biedronka,
- dla każdej mrówki znajdzie jej końcowe położenie i wyznaczy liczbę mówiącą, ile razy przegoniła ona biedronkę.

Wejście

W pierwszym wierszu wejścia znajduje się jedna liczba całkowita n, równa łącznej liczbie liści i rozgałęzień w drzewie, $1 \leqslant n \leqslant 5\,000$. Przyjmujemy, że liście i rozgałęzienia są ponumerowane od 1 do n. W kolejnych $n-1$ wierszach są opisane gałązki — w każdym z tych wierszy są zapisane dwie liczby całkowite a i b oznaczające, że dana gałązka łączy miejsca a i b. Gałązki pozwalają na przejście z każdego miejsca na drzewie, do każdego innego miejsca. W $n+1$-szym wierszu jest zapisana jedna liczba całkowita k, $1 \leqslant k \leqslant 1\,000$, $k \leqslant n$, równa liczbie mrówek patrolujących drzewo. W każdym z kolejnych k wierszy zapisana jest jedna liczba całkowita z przedziału od 1 do n. Liczba zapisana w wierszu $n+1+i$ oznacza początkowe położenie mrówki nr i. W każdym miejscu (liściu lub rozgałęzieniu) może znajdować się co najwyżej jedna mrówka. W wierszu $n+k+2$ zapisana jest jedna liczba całkowita l, $1 \leqslant l \leqslant 500$, mówiąca ile razy biedronka siada na drzewie. W każdym z kolejnych l wierszy zapisana jest jedna liczba całkowita z zakresu od 1 do n. Liczby te opisują kolejne miejsca, w których siada biedronka.

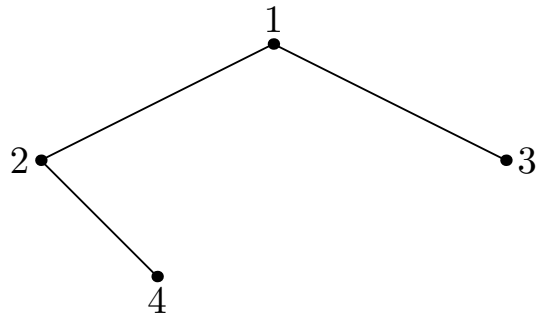
Wyjście

Twój program powinien wypisać k wierszy. W i-tym wierszu powinny zostać zapisane dwie liczby całkowite oddzielone pojedynczym odstępem — końcowa pozycja i-tej mrówki (numer rozgałęzienia lub liścia) i — liczba mówiąca, ile razy przegoniła ona biedronkę.

Przykład

Dla następującego wejścia:

```
4
1 2
1 3
2 4
2
1
2
2
2
4
```



Poprawnym rozwiązaniem jest:

```
1 0
4 2
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10187	acm.uva.es - zadanie 321	acm.uva.es - zadanie 589
acm.uva.es - zadanie 10959	acm.uva.es - zadanie 10067	acm.uva.es - zadanie 10097
spoj.sphere.pl - zadanie 206	spoj.sphere.pl - zadanie 135	spoj.sphere.pl - zadanie 212
acm.sgu.ru - zadanie 226	acm.sgu.ru - zadanie 213	

2.3. Przeszukiwanie grafu w głąb

Literatura
[WDA] - 23.3
[KDP] - 2.2
[MD] - 7.3

Inną metodą systematycznego przeszukiwania grafów jest algorytm DFS — przeszukiwanie w głąb (ang. depth - first search). Metoda ta — podobnie jak BFS: działa prawidłowo zarówno dla grafów skierowanych, jak i nieskierowanych. Rozpoczyna ona przeszukiwanie grafu od pewnego startowego wierzchołka s, a następnie rekurencyjnie odwiedza wszystkich jeszcze nie odwiedzonych sąsiadów przetwarzanego wierzchołka. Podczas wykonywania algorytmu, wyznaczane są dla każdego wierzchołka trzy wartości — d, f oraz s. Zmienne d i f reprezentują odpowiednio czasy wejścia oraz wyjścia z wierzchołka, natomiast s to numer wierzchołka-ojca w wyznaczonym lesie DFS. W algorytmie zakłada się, że każda wykonywana operacja (wejście oraz wyjście z wierzchołka) zajmuje jednostkę czasu. Czas wejścia do wierzchołka v jest rozumiany jako moment, w którym rozpoczęto przetwarzanie tego wierzchołka (przeszukiwanie DFS po raz pierwszy odwiedziło ten wierzchołek), natomiast czas wyjścia z wierzchołka v, to czas, w którym przetwarzanie wierzchołka v zostało zakończone. Ze względu na rekurencyjne odwiedzanie wszystkich sąsiadów wierzchołka v przed zakończeniem jego przetwarzania, czasy wejścia oraz wyjścia reprezentują poprawną strukturę nawiasową, w której wejście do wierzchołka reprezentowane jest przez nawias otwierający, natomiast wyjście — nawias zamykający. Na rysunkach 2.3.d oraz 2.3.e przedstawione są przykładowe kolejności przetwarzania wierzchołków grafu.

Czasy wyznaczane przez algorytm DFS mają wiele zastosowań — służą między innymi do znajdowania mostów, punktów artykulacji, czy silnie spójnych składowych w grafach. Zmienna s dla wierzchołka v (podobnie jak w algorytmie BFS) reprezentuje numer wierzchołka, z którego wierzchołek v został odwiedzony — wszystkie wartości s wyznaczają las przeszukiwań

w głąb (przykłady takich lasów są przedstawione na rysunkach 2.3.b oraz 2.3.c).

Implementacja algorytmu DFS, przedstawiona na listingu 2.8, jest zrealizowana przez funkcję **void** Graph<V,E>::DfsR(**int**), przyjmującą jako parametr numer wierzchołka, z którego należy rozpocząć przeszukiwanie. W przypadku, gdy parametr ten nie został podany, wykonywane jest przeszukiwanie ze wszystkich, jeszcze nie odwiedzonych wierzchołków grafu, w kolejności rosnących numerów (przykład takiej sytuacji przedstawiono na rysunku 2.3.c).

Listing 2.8: Implementacja rekurencyjna przeszukiwania DFS jako funkcja **void** Graph <V,E> ::DfsR(**int**)

```
// W polu int d każdego wierzchołka algorytm umieszcza czas wejścia, w polu
// int f - czas wyjścia, natomiast w polu int s - numer ojca w lesie DFS
// (wartości te dla wierzchołków nieodwiedzonych są równe -1)
// Zmienna t jest używana do pamiętania aktualnego czasu przetwarzania
int t;
// Funkcja rekurencyjna, przeszukująca poddrzewo wierzchołka v metodą DFS
void DfsV(int v) {
// Ustaw czas wejścia do wierzchołka
   g[v].d = ++t;
// Dla każdej krawędzi wychodzącej z wierzchołka v, jeśli wierzchołek, do
// którego prowadzi krawędź nie był jeszcze odwiedzony, to go odwiedź
   FOREACH(it, g[v]) if (g[it->v].s == -1) {
      g[it->v].s = v;
      DfsV(it->v);
   }
// Ustaw czas wyjścia z wierzchołka
   g[v].f = ++t;
}
void DfsR(int e = -1) {
   t = -1;
   int b = 0;
// Dla wierzchołków z przedziału [b..e] zostanie wywołana funkcja
// void DfsV(int). Jeśli do funkcji void DfsR(int) nie przekazano
// parametru, to przedziałem tym jest [0..SIZE(g)-1] (wszystkie wierzchołki w
// grafie), w przeciwnym przypadku [e..e] (tylko jeden wierzchołek)
   e == -1 ? e = SIZE(g) - 1 : b = e;
   REP(x, SIZE(g)) g[x].d = g[x].f = g[x].s = -1;
// Wykonaj algorytm DFS dla wszystkich wierzchołków z przedziału [b..e]
   FOR(x, b, e) if (g[x].s == -1) DfsV(x);
}
```

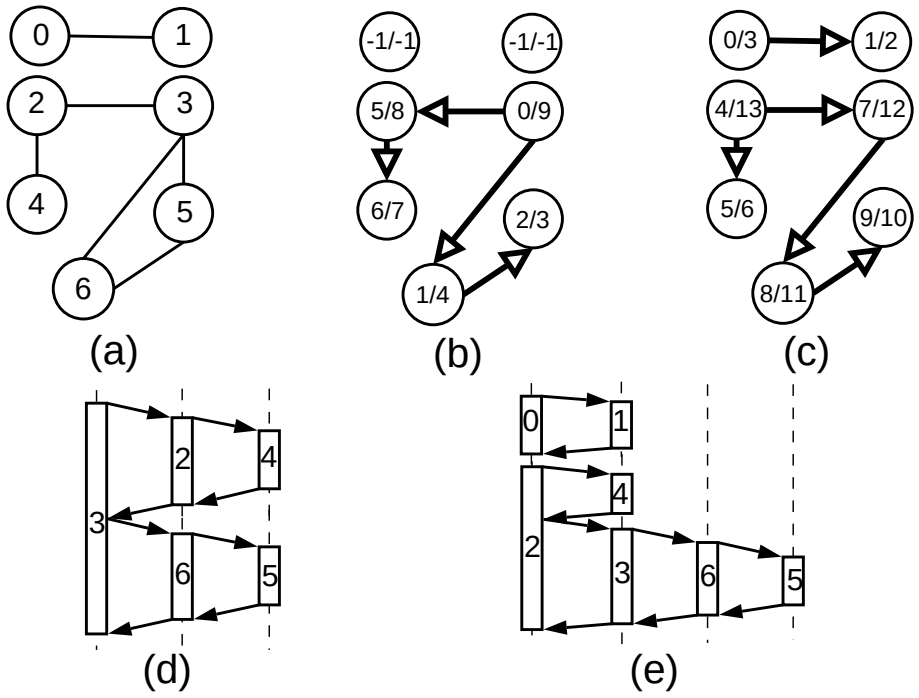
Implementacja przeszukiwania grafu metodą DFS, realizowana przez funkcję **void** Graph <V,E>::DfsR(**int**), jest rekurencyjna. Podejście takie jest stosunkowo proste w realizacji (nie jest wymagane implementowanie własnego stosu aktualnie przetwarzanych wierzchołków), jednak podczas operowania na dużych grafach, pojawia się ryzyko przepełnienia stosu programu, a co za tym idzie — doprowadzenie do błędu wykonania. W przypadku przeszukiwania dużych grafów, należy upewnić się, że w systemie operacyjnym, w którym będzie wykonywany program, nie ma dodatkowego ograniczenia na stos lub jest ono na tyle duże, że nie zagraża działaniu programu. W przypadku, gdy nie istnieje możliwość skorzystania z rekurencyjnej

implementacji algorytmu, można użyć implementacji iteracyjnej, realizowanej przez funkcję **void** Graph<V,E>::Dfs(**int**). Kod źródłowy tej funkcji przedstawiony został na listingu 2.9.

Listing 2.9: Implementacja iteracyjna przeszukiwania grafu w głąb jako funkcja **void** Graph<V,E>::Dfs(**int**)

```
01 void Dfs(int e = -1) {
02   VI st(SIZE(g));
03   int t = -1, i = 0, b = 0;
04   e == -1 ? e = SIZE(g) - 1 : b = e;
05   REP(x, SIZE(g)) g[x].d = g[x].f = g[x].s = -1;
// Dla wszystkich wierzchołków z przedziału [b..e], jeśli wierzchołek nie był
// jeszcze odwiedzony...
06   FOR(s, b, e) if (g[s].d == -1) {
// Wstaw wierzchołek na stos i ustaw dla niego odpowiedni czas wejścia.
// Zmienna f wierzchołka służy tymczasowo jako licznik nieprzetworzonych
// krawędzi wychodzących z wierzchołka.
07     g[st[i++] = s].d = ++t;
08     g[s].f = SIZE(g[s]);
// Dopóki stos jest niepusty...
09     while (i) {
10       int s = st[i - 1];
// Przetwórz kolejną krawędź wychodzącą z aktualnego wierzchołka, lub usuń
// go ze stosu (gdy nie ma już więcej krawędzi)
11       if (!g[s].f) {
12         g[s].f = ++t;
13         --i;
14       } else {
// Jeśli wierzchołek, do którego prowadzi krawędź nie był jeszcze
// odwiedzony, to...
15         if (g[s = g[s][--g[s].f].v].d == -1) {
// Ustaw numer wierzchołka-ojca w drzewie DFS oraz ustaw liczbę
// nieprzetworzonych krawędzi wychodzących z wierzchołka
16           g[s].s = st[i - 1];
17           g[s].f = SIZE(g[s]);
// Wstaw wierzchołek na stos i ustaw czas wejścia
18           g[st[i++] = s].d = ++t;
19         }
20       }
21     }
22   }
23 }
```

Algorytm przeszukiwania grafów metodą DFS ma złożoność $O(n + m)$. Każdy wierzchołek w grafie odwiedzany jest co najwyżej raz, a w ramach jego przetwarzania, sprawdzane są wszystkie krawędzie z niego wychodzące — operacja ta również wykonywana jest jednokrotnie.



Rysunek 2.3: **(a)** Przykładowy nieskierowany graf o siedmiu wierzchołkach. **(b)** Drzewo DFS dla źródła wyszukiwania w wierzchołku 3 (wewnątrz wierzchołków wpisane są odpowiednio czasy wejścia oraz wyjścia). **(c)** Drzewo DFS dla przeszukiwania całego grafu w kolejności rosnących numerów wierzchołków (odpowiednik wywołania funkcji **void** `Graph<V,E>::Dfs()`). **(d)** Kolejność przeszukiwania wierzchołków dla drzewa DFS z rysunku **(b)**. **(e)** Kolejność przeszukiwania wierzchołków dla drzewa DFS z rysunku **(c)**.

Dla nieskierowanego grafu przedstawionego na rysunku 2.3.a, po wykonaniu procedury `Graph<V,E>::Dfs(3)`, wartości wyliczonych przez algorytm zmiennych są przedstawione na listingu 2.10. Jeśli dla tego samego przykładu wykonać przeszukiwanie całego grafu metodą DFS — odpowiadające wywołaniu funkcji `Graph<V,E>::Dfs()`, to wyznaczone wartości zmiennych będą takie, jak na listingu 2.11.

Listing 2.10: Wynik wykonania funkcji **void** `Graph<V,E>::Dfs(3)` dla grafu z rysunku 2.3.a (rezultat jest zależny od kolejności wstawiania krawędzi do grafu)

```
Wierzcholek 0: czas wejscia = -1, czas wyjscia = -1, ojciec w lesie DFS = -1
Wierzcholek 1: czas wejscia = -1, czas wyjscia = -1, ojciec w lesie DFS = -1
Wierzcholek 2: czas wejscia = 5, czas wyjscia = 8, ojciec w lesie DFS = 3
Wierzcholek 3: czas wejscia = 0, czas wyjscia = 9, ojciec w lesie DFS = -1
Wierzcholek 4: czas wejscia = 6, czas wyjscia = 7, ojciec w lesie DFS = 2
Wierzcholek 5: czas wejscia = 2, czas wyjscia = 3, ojciec w lesie DFS = 6
Wierzcholek 6: czas wejscia = 1, czas wyjscia = 4, ojciec w lesie DFS = 3
```

Listing 2.11: Wynik wykonania **void** `Graph<V,E>::Dfs()` (rezultat jest zależny od kolejności wstawiania krawędzi do grafu)

```
Wierzcholek 0: czas wejscia = 0, czas wyjscia = 3, ojciec w lesie DFS = -1
Wierzcholek 1: czas wejscia = 1, czas wyjscia = 2, ojciec w lesie DFS = 0
Wierzcholek 2: czas wejscia = 4, czas wyjscia = 13, ojciec w lesie DFS = -1
Wierzcholek 3: czas wejscia = 7, czas wyjscia = 12, ojciec w lesie DFS = 2
Wierzcholek 4: czas wejscia = 5, czas wyjscia = 6, ojciec w lesie DFS = 2
Wierzcholek 5: czas wejscia = 9, czas wyjscia = 10, ojciec w lesie DFS = 6
Wierzcholek 6: czas wejscia = 8, czas wyjscia = 11, ojciec w lesie DFS = 3
```

Listing 2.12: Program użyty do wyznaczenia wyniku z listingów 2.10 i 2.11. Pełny kod źródłowy tego programu znajduje się w pliku `dfs_str.cpp`

```
// Krawędzie grafu nieskierowanego wymagają dodatkowego pola int rev
01 struct Ve {
02    int rev;
03 };
// Wzbogacenie wierzchołków przechowujące wynik generowany przez algorytm DFS
04 struct Vs {
05    int d, f, s;
06 };
07 int main() {
08    int n, m, s, b, e;
// Wczytaj liczbę wierzchołków, krawędzi oraz numer wierzchołka startowego
09    cin >> n >> m >> s;
// Skonstruuj odpowiedni graf
10    Graph<Vs, Ve> g(n);
// Dodaj do grafu wszystkie krawędzie
11    REP(x,m) {
12       cin >> b >> e;
13       g.EdgeU(b, e);
14    }
```

Listing 2.12: (c.d. listingu z poprzedniej strony)

```
// Wykonaj algorytm DFS i wypisz wynik
15   g.Dfs(s);
16   REP(x, SIZE(g.g)) cout << "Wierzcholek " << x << ": czas wejscia = " <<
17      g.g[x].d << ", czas wyjscia = " << g.g[x].f <<
18      ", ojciec w lesie DFS = " << g.g[x].s << endl;
19   return 0;
20 }
```

Zadanie: Komiwojażer Bajtazar

Pochodzenie: IX Olimpiada Informatyczna

Rozwiązanie: `kom.cpp`

Komiwojażer Bajtazar ciężko pracuje podróżując po Bajtocji. W dawnych czasach komiwojażerowie sami mogli wybierać miasta, które chcieli odwiedzić i kolejność w jakiej to czynili, jednak te czasy minęły już bezpowrotnie. Z chwilą utworzenia Centralnego Urzędu d/s Kontroli Komiwojażerów, każdy komiwojażer otrzymuje z Urzędu listę miast, które może odwiedzić i kolejność w jakiej powinien to uczynić. Jak to zazwyczaj bywa z centralnymi urzędami, narzucona kolejność odwiedzania miast nie ma zbyt dużo wspólnego z kolejnością optymalną. Przed wyruszeniem w trasę Bajtazar chciałby przynajmniej dowiedzieć się, ile czasu zajmie mu odwiedzenie wszystkich miast — obliczenie tego jest Twoim zadaniem.

Miasta w Bajtocji są ponumerowane od 1 do n. Numer 1 ma stolica Bajtocji, z niej właśnie rozpoczyna podróż Bajtazar. Miasta połączone są siecią dróg dwukierunkowych. Podróż między dwoma miastami bezpośrednio połączonymi drogą zawsze zajmuje 1 jednostkę czasu. Ze stolicy można dotrzeć do wszystkich pozostałych miast Bajtocji. Jednak sieć dróg została zaprojektowana bardzo oszczędnie, stąd drogi nigdy nie tworzą cykli.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis sieci dróg Bajtocji oraz listę miast, które musi odwiedzić Bajtazar
- obliczy sumaryczny czas podróży Bajtazara,
- wypisze wynik na standardowe wyjście

Wejście

W pierwszym wierszu wejścia zapisana jest jedna liczba całkowita n równa liczbie miast w Bajtocji ($1 \leqslant n \leqslant 30\,000$). W kolejnych $n-1$ wierszach opisana jest sieć dróg — w każdym z tych wierszy są zapisane dwie liczby całkowite a i b ($1 \leqslant a, b \leqslant n$, $a \neq b$), oznaczające, że miasta a i b połączone są drogą. W wierszu o numerze $n+1$ zapisana jest jedna liczba całkowita m równa liczbie miast, które powinien odwiedzić Bajtazar, ($1 \leqslant m \leqslant 50\,000$). W następnych m wierszach zapisano numery kolejnych miast na trasie podróży Bajtazara — po jednej liczbie w wierszu.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna zostać zapisana jedna liczba całkowita równa łącznemu czasowi podróży Bajtazara.

Przykład

Dla następującego wejścia:

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

Poprawnym rozwiązaniem jest:

```
7
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 260	acm.uva.es - zadanie 871	acm.uva.es - zadanie 10410
acm.uva.es - zadanie 280	acm.uva.es - zadanie 10802	spoj.sphere.pl - zadanie 287
acm.uva.es - zadanie 459	spoj.sphere.pl - zadanie 38	
acm.uva.es - zadanie 776	spoj.sphere.pl - zadanie 372	

2.4. Silnie spójne składowe

Literatura
[WDA] - 23.5
[ASD] - 7.3

Graf skierowany $G = (V, E)$ nazywany jest silnie spójnym, jeżeli dla każdej pary wierzchołków $u, v \in V$ z tego grafu, istnieje ścieżka prowadząca z wierzchołka u do wierzchołka v oraz z wierzchołka v do wierzchołka u ($u \rightsquigarrow v$ oraz $v \rightsquigarrow u$).

Dla każdego skierowanego grafu G można dokonać takiego podziału jego wierzchołków na maksymalne (w sensie zawierania) rozłączne zbiory, że grafy indukowane przez te zbiory stanowią silnie spójne składowe (ang. strongly connected components). Wyznaczanie silnie spójnych składowych jest często istotne przy rozwiązywaniu różnych problemów grafowych, nie tylko ze względu na możliwość zredukowania wielkości problemu (poprzez niezależne analizowanie poszczególnych, silnie spójnych składowych), ale również ze względu na możliwość sprowadzenia grafu do postaci acyklicznej poprzez wyznaczenie grafu silnie spójnych składowych (przykład takiego grafu przedstawiony jest na rysunku 2.4.b).

Do wyznaczania silnie spójnych składowych wykorzystuje się informacje wygenerowane przez algorytm DFS. W pierwszym kroku, dla wszystkich wierzchołków wyznacza się ich czasy przetworzenia (f) przez algorytm DFS, a następnie odwraca się skierowanie wszystkich krawędzi w grafie i wykonuje się algorytm DFS jeszcze raz, tym razem rozpoczynając przeszukiwanie grafu z wierzchołków w kolejności malejących wartości f. Każde pojedyncze drzewo przeszukiwania w głąb z fazy drugiej stanowi jedną silnie spójną składową.

Poprawność takiego algorytmu wynika z następujących trzech faktów:

- jeśli w grafie istnieją ścieżki $u \rightsquigarrow v$ oraz $v \rightsquigarrow u$, to wierzchołki u i v należą do tej samej silnie spójnej składowej, a przeszukiwanie w głąb umieści oba wierzchołki w tym samym drzewie DFS, gdyż bez względu na to który wierzchołek zostanie odwiedzony wcześniej podczas drugiej fazy algorytmu, drugi będzie jego potomkiem.

- jeśli w grafie nie istnieje ścieżka $u \rightsquigarrow v$, ani $v \rightsquigarrow u$, to wierzchołki nie mogą znaleźć się w

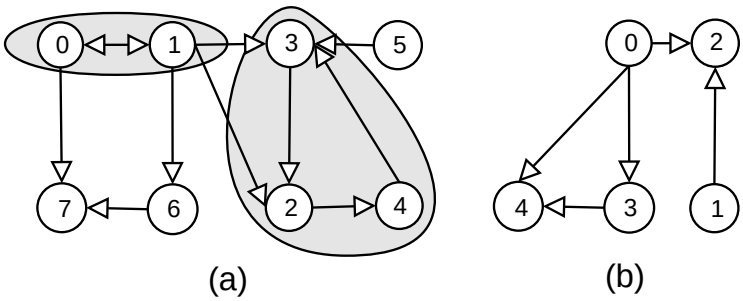
tym samym drzewie przeszukiwań DFS oraz nie są w tej samej silnie spójnej składowej.

- jeśli istnieje tylko jedna ścieżka (dla ustalenia uwagi niech to będzie $u \rightsquigarrow v$), to pierwsza faza algorytmu DFS przypisze mniejszą wartość f wierzchołkowi v, co spowoduje, że w drugiej fazie, jako pierwszy, zostanie odwiedzony wierzchołek v. Ze względu na brak ścieżki $v \rightsquigarrow u$, wierzchołki u i v trafią do różnych drzew przeszukiwań DFS.

Funkcja **void** `Graph<V,E>::SccS()` dla każdego wierzchołka wyznacza numer silnie spójnej składowej do której on należy — informacja ta jest umieszczana w dodatkowej zmiennej wierzchołka **int** `t`. Implementacja tego algorytmu przedstawiona została na listingu 2.13.

Listing 2.13: Implementacja funkcji **void** `Graph<V,E>::SccS()`

```
// Zmienna nr w pierwszej fazie algorytmu używana jest do pamiętania czasu
// odwiedzania wierzchołków, natomiast w drugiej fazie algorytmu do numerowania
// silnie spójnych składowych
int nr;
// Funkcja rekurencyjna, przeszukująca poddrzewo wierzchołka v. Jest ona
// używana do realizacji obu faz przeszukiwania DFS
void SccSDfs(int v) {
// Jeśli wierzchołek nie był odwiedzony, to go odwiedź
  if (g[v].t == -1) {
    g[v].t = nr;
// Odwiedź wszystkie wierzchołki, do których prowadzi krawędź z v
    FOREACH(it, g[v]) SccSDfs(it->v);
// Jeśli wykonywana jest pierwsza faza algorytmu, to ustaw zmienną t
// wierzchołka na czas przetworzenia (odejmowanie wartości 3 gwarantuje
// przydzielanie numerów poczynając od 0 wzwyż)
    if (nr < 0) g[v].t = -(--nr) - 3;
  }
}
// Właściwa funkcja, wykonująca wyznaczanie silnie spójnych składowych
void SccS() {
// Zbuduj graf transponowany gt oraz ustaw wartości zmiennych t
// wierzchołków na -1 (nieodwiedzone)
  Graph<V, E> gt(SIZE(g));
  REP(x, SIZE(g)) {
    g[x].t = gt.g[x].t = -1;
    FOREACH(it, g[x]) gt.EdgeD(it->v, x);
  }
  gt.nr = -2;
  nr = 0;
  VI v(SIZE(g));
// Wykonaj pierwszą fazę przeszukiwania w głąb na grafie transponowanym oraz
// wypełnij wektor v numerami wierzchołków w kolejności rosnących czasów
// przetworzenia DFS
  REP(x, SIZE(g)) {
    gt.SccSDfs(x);
    v[gt.g[x].t] = x;
  }
```



Rysunek 2.4: **(a)** przykładowy graf skierowany o ośmiu wierzchołkach. **(b)** graf silnie spójnych składowych wyznaczony dla grafu z rysunku **(a)**. Wierzchołek 0 reprezentuje zbiór wierzchołków $\{0, 1\}$, 1 — $\{5\}$, 2 — $\{2, 3, 4\}$, 3 — $\{6\}$, 4 — $\{7\}$

Listing 2.13: (c.d. listingu z poprzedniej strony)

```
// Wykonaj drugą fazę przeszukiwania w głąb na oryginalnym grafie.
22    FORD(x, SIZE(g) - 1, 0) {
23       SccSDfs(v[x]);
24       nr++;
25    }
26 }
```

W wielu przypadkach, wyliczenie dla każdego wierzchołka v z grafu G numeru silnie spójnej składowej, do której on należy, nie jest wystarczające. Często przydaje się również skonstruowanie grafu silnie spójnych składowych G', w którym każdy wierzchołek reprezentuje jedną silnie spójną składową. Niech u' oraz v' będą wierzchołkami w grafie G' silnie spójnych składowych grafu G. Wierzchołki te są połączone skierowaną krawędzią (u', v') wtedy, gdy w grafie G istnieją wierzchołki u oraz v połączone krawędzią (u, v), należące odpowiednio do silnie spójnych składowych, reprezentowanych przez wierzchołki u' oraz v'. Graf silnie spójnych składowych dla przykładowego grafu z rysunku 2.4.a przedstawiony jest na rysunku 2.4.b. Każdy graf silnie spójnych składowych jest acykliczny — gdyby bowiem istniał w nim cykl, to wszystkie wierzchołki na nim leżące reprezentowałyby tę samą silnie spójną składową.

Funkcja `Graph<V,E> Graph<V,E>::Scc()`, której implementacja została przedstawiona na listingu 2.14, oprócz wyznaczania wartości zmiennych **int** `t`, jako wynik swojego działania zwraca dodatkowo graf silnie spójnych składowych. Wierzchołki tego grafu są numerowane zgodnie z porządkiem topologicznym (od wierzchołka o większym numerze nie prowadzi krawędź do wierzchołka o numerze mniejszym), co jest bardzo pożyteczną własnością podczas rozwiązywania wielu zadań.

Listing 2.14: Implementacja funkcji `Graph<V,E> Graph<V,E>::Scc()`

```
// Wektor służący do odznaczania odwiedzonych wierzchołków
01 VI vis;
// Wskaźnik do konstruowanego grafu silnie spójnych składowych
02 Graph<V, E> *sccRes;
// Funkcja przechodząca graf algorytmem DFS. Jest ona wykorzystywana dwukrotnie -
// w pierwszej fazie podczas wyznaczania kolejności wierzchołków dla drugiej fazy,
// oraz podczas drugiej fazy - do wyznaczania silnie spójnych składowych oraz
// konstrukcji grafu silnie spójnych składowych
03 void SccDfs(int v, int nr, bool phase) {
// Zaznacz wierzchołek jako odwiedzony
04   g[v].t = 1;
// Jeśli wykonywana jest druga faza przeszukiwania, to ustaw dla wierzchołka
// numer silnie spójnej składowej
05   if (!phase) vis[v] = nr;
// Odwiedź kolejne wierzchołki oraz (jeśli wykonywana jest druga faza) dodaj
// krawędź do konstruowanego grafu silnie spójnych składowych
06   FOREACH(it, g[v]) if (g[it->v].t == -1) SccDfs(it->v, nr, phase);
07   else if (!phase && nr > vis[it->v])
08     sccRes->EdgeD(g[it->v].t, vis[it->v] = nr);
// Jeśli wykonywana jest pierwsza faza, to wstaw wierzchołek do listy, jeśli
// natomiast druga, to zaktualizuj jego czas
09   if (phase) vis.PB(v);
10   else g[v].t = nr;
11 }
// Funkcja wyznaczająca silnie spójne składowe w grafie
12 Graph<V, E> Scc() {
// Graf gt to graf transponowany, natomiast res to konstruowany graf
// silnie spójnych składowych
13   Graph<V, E> gt(SIZE(g)), res(SIZE(g)), *tab[] = {
14   this, &gt};
15   gt.sccRes = &res;
16   gt.vis.resize(SIZE(g), -1);
17   vis.clear();
// Budowa grafu transponowanego
18   REP(i, SIZE(g)) FOREACH(it, g[i]) gt.EdgeD(it->v, i);
// Przeprowadź dwie fazy algorytmu DFS...
19   REP(i, 2) {
// Zaznacz wierzchołki jako nieodwiedzone
20     FOREACH(it, tab[i]->g) it->t = -1;
21     int comp = 0, v;
// Dla kolejnych, nieodwiedzonych wierzchołków, wykonaj przeszukiwanie
22     FORD(j, SIZE(g) - 1, 0)
23     if (tab[i]->g[v = (i ? vis[j] : j)].t == -1)
24       tab[i]->SccDfs(v, comp++, 1 - i);
25     if (i) res.g.resize(comp);
26   }
27   REP(i, SIZE(g)) g[i].t = gt.g[i].t;
```

Listing 2.14: (c.d. listingu z poprzedniej strony)

```
28    return res;
29 }
```

Złożoność czasowa obu przedstawionych funkcji — **void** `Graph<V,E>::SccS()` oraz `Graph<V,E> Graph<V,E>::Scc()` jest liniowa ze względu na wielkość wejściowego grafu ($O(n+m)$). Wynika to z faktu, że obie funkcje wykonują dwukrotnie zmodyfikowany algorytm DFS. Funkcja **void** `Graph<V,E>::SccS()` nie tylko jest krótsza w implementacji, ale w praktyce okazuje się być szybsza, ze względu na brak konieczności konstruowania grafu wynikowego, dlatego też, jeśli rezultat przez nią wyznaczany jest wystarczający, nie należy korzystać z `Graph<V,E> Graph<V,E>::Scc()`.

Listing 2.15: Wynik wyznaczony przez funkcję `Graph<V,E>::Scc()` dla grafu z rysunku 2.4.a

```
Wierzcholek 0 nalezy do silnie spojnej skladowej numer 0
Wierzcholek 1 nalezy do silnie spojnej skladowej numer 0
Wierzcholek 2 nalezy do silnie spojnej skladowej numer 2
Wierzcholek 3 nalezy do silnie spojnej skladowej numer 2
Wierzcholek 4 nalezy do silnie spojnej skladowej numer 2
Wierzcholek 5 nalezy do silnie spojnej skladowej numer 1
Wierzcholek 6 nalezy do silnie spojnej skladowej numer 3
Wierzcholek 7 nalezy do silnie spojnej skladowej numer 4
Graf silnie spojnych skladowych:
0: 2 3 4
1: 2
2:
3: 4
4:
```

Listing 2.16: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.15. Pełny kod źródłowy programu znajduje się w pliku `scc_str.cpp`

```
01 struct Ve {};
// Wzbogacenie wierzchołków zawiera pole, w którym umieszczany jest numer
// silnie spójnej składowej
02 struct Vs {
03    int t;
04 };
05 int main() {
06    int n, m, s, b, e;
07    Ve ed;
// Wczytaj liczbę wierzchołków oraz krawędzi
08    cin >> n >> m;
09    Graph<Vs, Ve> g(n);
// Dodaj do grafu wszystkie krawędzie
10    REP(x, m) {
11       cin >> b >> e;
12       g.EdgeD(b, e);
```

Listing 2.16: (c.d. listingu z poprzedniej strony)

```
13    }
// Wykonaj algorytm Scc oraz wypisz wyznaczony wynik
14    Graph<Vs, Ve> scc = g.Scc();
15    REP(x,n) cout << "Wierzcholek " << x <<
16       " nalezy do silnie spojnej skladowej numer " << g.g[x].t << endl;
17    cout << "Graf silnie spojnych skladowych: " << endl;
18    scc.Write();
19    return 0;
20 }
```

Zadanie: Drogi

Pochodzenie: Potyczki Algorytmiczne 2005

Rozwiązanie: `road.cpp`

W Bajtocji jest n miast. Miasta są połączone jednokierunkowymi drogami. Każda droga łączy tylko dwa miasta i nie przechodzi przez żadne inne. Niestety, nie zawsze z każdego miasta da się dojechać do każdego innego. Król Bajtazar postanowił rozwiązać ten problem. Król ma świadomość, że budowanie nowych dróg jest bardzo kosztowne, a budżet Bajtocji nie jest zbyt zasobny. Dlatego też poprosił Cię o pomoc. Należy obliczyć minimalną liczbę jednokierunkowych dróg, które trzeba zbudować, żeby z każdego miasta dało się dojechać do każdego innego miasta.

Zadanie

Napisz program, który:

- wczyta opis istniejącej sieci dróg,
- obliczy minimalną liczbę dróg, które trzeba dobudować tak, aby z każdego miasta w Bajtocji dało się dojechać do każdego innego,
- wypisze wynik.

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite n i m ($2 \leqslant n \leqslant 10\,000, 0 \leqslant m \leqslant 100\,000$) oddzielone pojedynczym odstępem i oznaczające, odpowiednio, liczbę miast i liczbę dróg w Bajtocji. Miasta są ponumerowane od 1 do n. W każdym z kolejnych m wierszy znajdują się dwie liczby całkowite, oddzielone pojedynczym odstępem. W $i+1$ wierszu znajdują się liczby a_i i b_i ($1 \leqslant a_i, b_i \leqslant n$ dla $1 \leqslant i \leqslant m$), reprezentują one jednokierunkową drogę prowadzącą z miasta a_i do b_i.

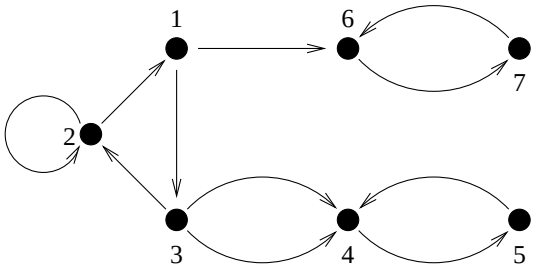
Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać dokładnie jedną nieujemną liczbę całkowitą — minimalną liczbę dróg, które trzeba zbudować w Bajtocji tak, aby z każdego miasta dało się dojechać do każdego innego miasta.

Przykład

Dla następującego wejścia:

```
7 11
1 3
3 2
2 1
2 2
3 4
4 5
5 4
3 4
1 6
6 7
7 6
```



Poprawnym rozwiązaniem jest:

```
2
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10731 spoj.sphere.pl - zadanie 51	acm.uva.es - zadanie 247	acm.uva.es - zadanie 125 acm.uva.es - zadanie 10510

2.5. Sortowanie topologiczne

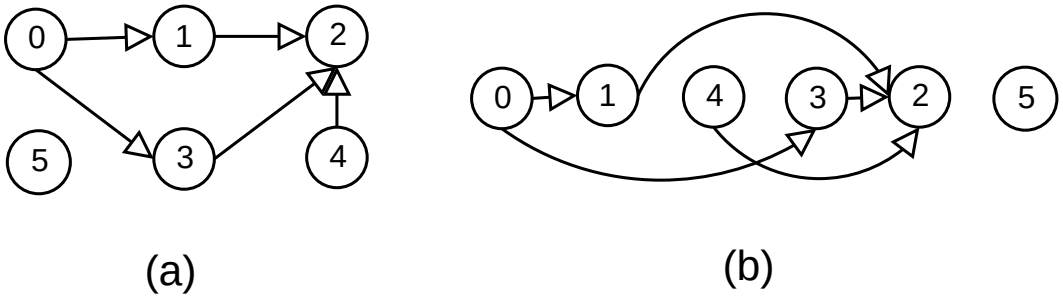
Literatura
[WDA] - 23.4
[ASP] - 4.3.3

Skierowany graf acykliczny (ang. directed acyclic graph), jest to graf, który nie posiada cykli. Innymi słowy, w grafie takim nie istnieje para wierzchołków u i v połączonych ścieżkami $u \rightsquigarrow v$ i $v \rightsquigarrow u$.

Sortowanie topologiczne skierowanego grafu $G = (V, E)$ polega na takim uporządkowaniu wierzchołków ze zbioru V, aby dla każdej krawędzi $(u, v) \in E$, wierzchołek u znajdował się przed wierzchołkiem v. Sortowanie topologiczne daje się wyznaczyć dla wszystkich grafów skierowanych nie zawierających cykli. Do wykonania tego zadania można wykorzystać algorytm DFS — jedyne, co należy zrobić, to posortować wierzchołki w kolejności malejących czasów przetworzenia f.

Załóżmy, że mamy dany skierowany acykliczny graf G oraz dwa wierzchołki u oraz v. Niech wartości f wyznaczone dla tych wierzchołków przez algorytm DFS będą równe odpowiednio u_f i v_f, oraz załóżmy bez straty ogólności, że $u_f < v_f$. Zgodnie z naszym sposobem wyznaczania porządku topologicznego, wierzchołek v zostanie umieszczony przed wierzchołkiem u. Postępowanie takie jest zgodne z definicją porządku topologicznego, gdyż z warunku $u_f < v_f$ oraz z acykliczności grafu G wynika, że w grafie tym nie ma krawędzi (u, v).

Algorytm DFS, oprócz zmiennych **int** `f`, wyznacza również inne, niepotrzebne z punktu widzenia sortowania topologicznego wartości, dlatego też implementacja funkcji **void** `Graph<V,E>::TopoSort()` nie korzysta z przedstawionej wcześniej implementacji algorytmu DFS. Wyznaczony porządek topologiczny umieszczany jest w dodatkowych polach wierzchołków **int** `t` — pierwszy wierzchołek ma tą wartość równą 0, drugi — 1... Listing 2.17 przedstawia implementację funkcji **void** `Graph<V,E>::TopoSort()`.



Rysunek 2.5: **(a)** Przykładowy skierowany graf acykliczny **(b)** jeden z możliwych porządków topologicznych określonych na zbiorze wierzchołków grafu z rysunku **(a)**

Listing 2.17: Implementacja funkcji **void** `Graph<V,E>::TopoSort()`

```
01 int topo;
// Funkcja wykonująca algorytm DFS z wierzchołka v i aktualizująca wartości
// zmiennych t
02 void TopoDfs(int v) {
// Jeśli wierzchołek nie był odwiedzony, to należy go odwiedzić
03   if (!g[v].t) {
// Zaznacz wierzchołek jako odwiedzony
04     g[v].t = 1;
// Odwiedź wszystkie wierzchołki, do których prowadzi krawędź z v
05     FOREACH(it, g[v]) TopoDfs(it->v);
// Zaktualizuj wartość t przetwarzanego wierzchołka
06     g[v].t = --topo;
07   }
08 }
// Właściwa funkcja implementująca sortowanie topologiczne
09 void TopoSort() {
10   FOREACH(it, g) it->t = 0;
11   topo = SIZE(g);
// Odwiedź wszystkie wierzchołki w grafie
12   FORD(x, topo - 1, 0) TopoDfs(x);
13 }
```

W wielu przypadkach, forma wyniku obliczana przez funkcję **void** `Graph<V,E>::TopoSort()` jest nieporęczna w użyciu — często oczekiwanym rezultatem jest posortowana lista wierzchołków grafu. Funkcja `VI Graph<V,E>::TopoSortV()`, której implementacja przedstawiona jest na listingu 2.18, jako wynik działania zwraca wektor liczb reprezentujących numery kolejnych wierzchołków w porządku topologicznym.

Listing 2.18: Implementacja funkcji `VI Graph<V,E>::TopoSortV()`

```
1 VI TopoSortV() {
2   VI res(SIZE(g));
// Wyznacz sortowanie topologiczne
3   TopoSort();
// Na podstawie wartości zmiennych t wierzchołków, wyznacz wektor z wynikiem
4   REP(x, SIZE(g)) res[g[x].t] = x;
```

Listing 2.18: (c.d. listingu z poprzedniej strony)

```
5     return res;
6 }
```

Czas działania algorytmu sortowania topologicznego to $O(n+m)$, co wynika bezpośrednio z faktu, iż jest on modyfikacją przeszukiwania grafu w głąb.

Listing 2.19: Wynik wyznaczony przez funkcję `VI Graph<V,E>::TopoSortV()` dla grafu z rysunku 2.5.a

```
Kolejnosc topologiczna wierzcholkow: 0 1 3 4 2 5
Wierzcholek 0 ma pozycje 0 w porzadku topologicznym
Wierzcholek 1 ma pozycje 1 w porzadku topologicznym
Wierzcholek 2 ma pozycje 4 w porzadku topologicznym
Wierzcholek 3 ma pozycje 2 w porzadku topologicznym
Wierzcholek 4 ma pozycje 3 w porzadku topologicznym
Wierzcholek 5 ma pozycje 5 w porzadku topologicznym
```

Listing 2.20: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.19. Pełny kod źródłowy programu znajduje się w pliku `toposort_str.cpp`

```
01 struct Ve {};
// Wzbogacenie wierzchołka o pole t, w którym umieszczany jest wynik
02 struct Vs {
03    int t;
04 };
05 int main() {
06    int n, m, b, e;
// Wczytaj liczbę wierzchołków oraz krawędzi, stwórz graf o odpowiedniej wielkości
07    cin >> n >> m;
08    Graph<Vs, Ve> g(n);
// Dodaj do grafu wszystkie krawędzie skierowane
09    REP(x,m) {
10       cin >> b >> e;
11       g.EdgeD(b, e);
12    }
// Wyznacz porządek topologiczny oraz wypisz wynik
13    VI res = g.TopoSortV();
14    cout << "Kolejnosc topologiczna wierzcholkow: ";
15    FOREACH(it, res) cout << *it << " ";
16    cout << endl;
17    REP(x, SIZE(g.g)) cout << "Wierzcholek " << x << " ma pozycje " <<
18       g.g[x].t << " w porzadku topologicznym" << endl;
19    return 0;
20 }
```

Zadanie: Spokojna komisja

Pochodzenie: VIII Olimpiada Informatyczna

Rozwiązanie: `comm.cpp`

W parlamencie Demokratycznej Republiki Bajtocji, zgodnie z Bardzo Ważną Ustawą, należy ukonstytuować Komisję Poselską do Spraw Spokoju Publicznego. Niestety sprawę utrudnia fakt, iż niektórzy posłowie wzajemnie się nie lubią.

Komisja musi spełniać następujące warunki:

- każda partia ma dokładnie jednego reprezentanta w Komisji,
- jeśli dwaj posłowie się nie lubią, to nie mogą jednocześnie być w Komisji.

Każda partia ma w parlamencie dokładnie dwóch posłów. Wszyscy posłowie są ponumerowani liczbami od 1 do $2n$. Posłowie o numerach $2i - 1$ i $2i$ należą do partii o numerze i.

Zadanie

Napisz program, który:

- wczyta liczbę partii oraz pary posłów, którzy się wzajemnie nie lubią,
- wyznaczy skład Komisji, lub stwierdzi, że nie da się jej ukonstytuować,
- wypisze wynik

Wejście

W pierwszym wierszu wejścia znajdują się dwie nieujemne liczby całkowite n i m. Liczba n, spełniająca warunki $1 \leqslant n \leqslant 8\,000$, oznacza liczbę partii. Liczba m, spełniająca warunki $0 \leqslant m \leqslant 20\,000$, oznacza liczbę par nielubiących się posłów. W każdym z kolejnych m wierszy zapisana jest para liczb naturalnych a i b, $1 \leqslant a \neq b \leqslant 2n$, oddzielonych pojedynczym odstępem. Oznacza ona, że posłowie o numerach a i b wzajemnie się nie lubią.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinno znaleźć się pojedyncze słowo NIE, jeśli utworzenie Komisji nie jest możliwe. W przypadku, gdy utworzenie Komisji jest możliwe, program powinien wypisać n liczb całkowitych z przedziału od 1 do $2n$, zapisanych w kolejności rosnącej i oznaczających numery posłów zasiadających w Komisji. Każda z tych liczb powinna zostać zapisana w osobnym wierszu. Jeśli Komisję można utworzyć na wiele sposobów, Twój program może wypisać dowolny z nich.

Przykład

Dla następującego wejścia:

```
3 2
1 3
2 4
```

Poprawnym rozwiązaniem jest:

```
1
4
6
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 124	acm.uva.es - zadanie 200	acm.uva.es - zadanie 10319
acm.uva.es - zadanie 10305	acm.sgu.ru - zadanie 230	spoj.sphere.pl - zadanie 44
spoj.sphere.pl - zadanie 70		

2.6. Acykliczność

Mając dany graf $G = (V, E)$ (skierowany bądź nieskierowany), można zadać pytanie, czy jest on acykliczny. Acykliczność grafu jest pożądaną własnością w wielu algorytmach. Przykładowo, wyznaczanie topologicznego porządku wierzchołków grafu skierowanego jest możliwe pod warunkiem, że graf, dla którego wyznaczany jest ten porządek, jest acykliczny. Opisany w poprzednim rozdziale algorytm zakłada prawdziwość tego faktu i nie sprawdza acykliczności grafu.

Literatura
[WDA] - 5.4

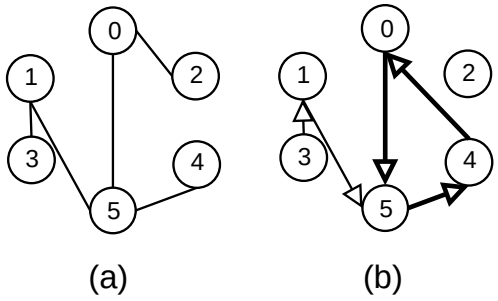
W przypadku grafów skierowanych, stosunkowo łatwo można zmodyfikować algorytm służący do sortowania topologicznego grafu, wzbogacając go o dodatkowy test na acykliczność. Wystarczy pod koniec algorytmu sprawdzić, czy wszystkie krawędzie prowadzą od wierzchołków znajdujących się wcześniej, do wierzchołków znajdujących się później w porządku topologicznym. Implementacja tej metodologii jest zrealizowana w funkcji **bool** `Graph<V,E>::AcyclicD()` — do działania wymaga ona zaimplementowania funkcji **void** `Graph<V,E>::TopoSort()`, a zatem wykorzystuje również dodatkowe pole **int** `t` wierzchołków. Implementacja tej funkcji znajduje się na listingu 2.21.

Listing 2.21: Implementacja funkcji **bool** `Graph<V,E>::AcyclicD()`

```
// Funkcja sprawdzająca, czy dany graf skierowany jest acykliczny
1 bool AcyclicD() {
// Wyznacz sortowanie topologiczne
2   TopoSort();
// Dla każdej krawędzi w grafie sprawdź, czy prowadzi ona od wierzchołka
// wcześniejszego do wierzchołka późniejszego w porządku topologicznym
3   FOREACH(it, g) FOREACH(it2, *it) if (it->t >= g[it2->v].t) return false;
4   return true;
5 }
```

Algorytm realizowany przez funkcję **bool** `Graph<V,E>::AcyclicD()` nie działa prawidłowo w przypadku grafów nieskierowanych — reprezentacja krawędzi (u, v) w takich grafach składa się z dwóch krawędzi skierowanych: (u, v), oraz (v, u). W podejściu realizowanym przez funkcję **bool** `Graph<V,E>::AcyclicD()`, każdy graf nieskierowany z co najmniej jedną krawędzią uważany jest za zawierający cykl.

Dla grafów acyklicznych potrzebny jest inny algorytm. Funkcja **bool** `Graph<V,E>::AcyclicU()`, której kod źródłowy został przedstawiony na listingu 2.22, jest implementacją pomysłu działającego właśnie dla takich grafów. Wykonuje ona algorytm DFS dla wszystkich wierzchołków w grafie i w momencie powtórnego wejścia do tego samego wierzchołka stwierdza, że zawiera on cykl. Problem z reprezentacją krawędzi nieskierowanych w postaci pary krawędzi skierowanych został rozwiązany poprzez nieprzetwarzanie krawędzi, po której algorytm wszedł do wierzchołka. Funkcja **bool** `Graph<V,E>::AcyclicU()` traktuje każdą pętlę oraz multikrawędzie jako cykl.



Rysunek 2.6: **(a)** Nieskierowany graf acykliczny **(b)** Skierowany graf zawierający cykl $5 \to 4 \to 0 \to 5$

Listing 2.22: Implementacja funkcji **bool** `Graph<V,E>::AcyclicU()`

```
// Wskaźnik na tablicę służącą do odznaczania wierzchołków odwiedzonych
01 bool *vis;
// Zmienna, w której umieszczany jest wynik
02 bool acyc;
// Funkcja przeszukująca poddrzewo wierzchołka v w celu znalezienia cyklu
03 void AcDfs(int v, Ed * p) {
// Jeśli wierzchołek nie był jeszcze odwiedzony...
04   if (!vis[v]) {
05     vis[v] = 1;
// Przetwórz jego wszystkie krawędzie za wyjątkiem tej, po której przyszedłeś
// od ojca
06     FOREACH(it, g[v]) if (&(*it) != p) AcDfs(it->v, &g[it->v][it->rev]);
07   } else acyc = 0;
08 }
// Funkcja sprawdzająca, czy graf jest acykliczny - działa prawidłowo dla grafów
// nieskierowanych
09 bool AcyclicU() {
// Inicjalizacja zmiennych
10   acyc = 1;
11   vis = new bool[SIZE(g)];
12   REP(x, SIZE(g)) vis[x] = 0;
// Dla każdego wierzchołka w grafie dokonaj przeszukiwania
13   REP(x, SIZE(g)) if (!vis[x]) AcDfs(x, 0);
14   delete[]vis;
15   return acyc;
16 }
```

Złożoność czasowa obu przedstawionych w tym rozdziale funkcji to $O(n+m)$. Tak samo jak w przypadku funkcji przedstawionych w poprzednich rozdziałach, implementacja jest pewną modyfikacją algorytmu DFS, która przetwarza każdy wierzchołek, oraz każdą krawędź grafu dokładnie raz. Dla grafu, przedstawionego na rysunku 2.6.a, funkcja **bool** `Graph<V,E>::AcyclicU()` zwróci fałsz, natomiast funkcja **bool** `Graph<V,E>::AcyclicD()` dla grafu z rysunku 2.6.b — prawdę.

Zadanie: Wirusy

Pochodzenie: VII Olimpiada Informatyczna

Rozwiązanie: `viruses.cpp`

Komisja Badania Wirusów Binarnych wykryła, że pewne ciągi zer i jedynek są kodami wirusów. Komisja wyodrębniła zbiór wszystkich kodów wirusów. Ciąg zer i jedynek nazywamy bezpiecznym, gdy żaden jego segment (tj. ciąg kolejnych znaków) nie jest kodem wirusa. Komisja dąży do ustalenia, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek.

Dla zbioru kodów $\{011, 11, 00000\}$ nieskończonym, bezpiecznym ciągiem jest $010101\ldots$ Dla zbioru kodów $\{01, 11, 00000\}$ nie istnieje nieskończony, bezpieczny ciąg zer i jedynek.

Zadanie

Napisz program, który:

- wczyta kody wirusów,
- stwierdzi, czy istnieje nieskończony, bezpieczny ciąg zer i jedynek,
- wypisze wynik.

Wejście

W pierwszym wierszu znajduje się jedna liczba całkowita n, będąca liczbą wszystkich kodów wirusów. W każdym z kolejnych n wierszy znajduje się jedno niepuste słowo złożone ze znaków 0 i 1 — kod wirusa. Sumaryczna długość wszystkich słów nie przekracza 30 000.

Wyjście

W pierwszym i jedynym wierszu powinno znajdować się słowo:

- TAK — jeżeli istnieje nieskończony, bezpieczny ciąg zer i jedynek,
- NIE — w przeciwnym przypadku.

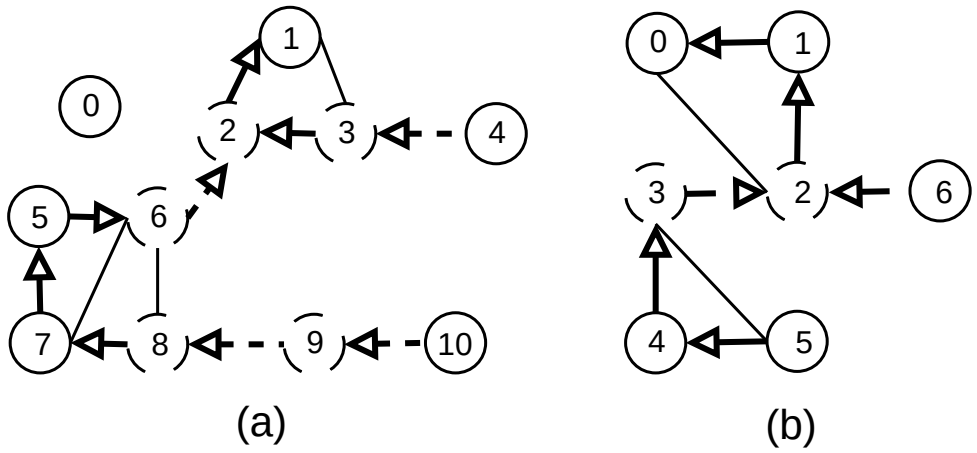
Przykład

Dla następującego wejścia:

```
3
01
11
00000
```

Poprawnym rozwiązaniem jest:

```
NIE
```



Rysunek 2.7: Dwa przykładowe grafy. Przerywane krawędzie reprezentują mosty, wierzchołki o przerywanym obwodzie to punkty artykulacji. Pogrubione krawędzie przedstawiają wyznaczone drzewa przeszukiwania algorytmu DFS. **(a)** Graf o jedenastu wierzchołkach i 12 krawędziach. Krawędzie $(8, 9)$, $(9, 10)$, $(2, 6)$ oraz $(3, 4)$ to mosty, wierzchołki 2, 3, 6 , 8 i 9 stanowią punkty artykulacji. **(b)** Graf zawiera dwa mosty — $(2, 3)$ i $(2, 6)$, oraz dwa punkty artykulacji — 2 i 3.

2.7. Mosty, punkty artykulacji i dwuspójne składowe

Niech $G = (V, E)$ będzie nieskierowanym, spójnym grafem. Na wstępie zdefiniujmy obiekty, którymi będziemy się interesowali w obrębie aktualnego rozdziału:

Literatura
[KDP] - 2.6
[ASD] - 7.2

Definicja 2.7.1 Punkt artykulacji *jest to taki wierzchołek grafu G, którego usunięcie zwiększa liczbę spójnych składowych grafu G.*

Definicja 2.7.2 Most *jest to taka krawędź $(u, v) \in E$, której usunięcie z grafu powoduje zwiększenie liczby spójnych składowych grafu G.*

Definicja 2.7.3 Dwuspójną składową *w grafie G jest maksymalny (ze względu na zawieranie) zbiór krawędzi, takich, że dowolne dwie z nich leżą na wspólnym cyklu prostym.*

Dwuspójne składowe wyznaczają jednoznaczny podział zbioru krawędzi grafu E. Zauważmy, że jeżeli krawędź (u, v) należy do jednoelementowej dwuspójnej składowej, to jest ona mostem — jej usunięcie powoduje, że wierzchołki u i v nie należą już do tej samej spójnej składowej grafu.

W aktualnym rozdziale przedstawione zostanie kilka implementacji funkcji, służących do wyznaczania dwuspójnych składowych, mostów oraz punktów artykulacji.

Wyznaczanie dwuspójnych składowych oraz elementów z nimi związanych (mostów oraz punktów artykulacji) w grafie jest możliwe dzięki zastosowaniu przeszukiwania grafu w głąb. Załóżmy zatem, że dla całego grafu $G = (V, E)$ wykonaliśmy już algorytm DFS, na skutek czego zostały wyznaczone czasy wejść do wierzchołków d oraz skonstruowany został las przeszukiwań w głąb. Do dalszej analizy zagadnienia będziemy potrzebowali funkcji low, określonej na zbiorze wierzchołków grafu G:

$$low(v) = min(d(v), min(d(w) : w \in W), min(low(q) : q \in Q)), \text{ dla } v \in V$$

gdzie W jest zbiorem wierzchołków, dla których istnieje krawędź $(v, w) \in E$ nie będąca krawędzią drzewową w lesie DFS, natomiast zbiór Q składa się z następników v w lesie przeszukiwań DFS. Bardziej intuicyjnie — $low(v)$ jest równe wartości najmniejszej zmiennej d dla wierzchołka osiągalnego z v przy użyciu krawędzi z poddrzewa DFS tego wierzchołka oraz co najwyżej jednej krawędzi niedrzewowej.

Charakterystyka mostów oraz punktów artykulacji, wykorzystująca funkcję low, jest następująca:

Wierzchołek v jest punktem artykulacji wtedy i tylko wtedy, gdy:

- jest korzeniem drzewa DFS oraz posiada co najmniej dwa następniki, lub
- nie jest korzeniem drzewa DFS oraz dla każdego następnika u wierzchołka v w drzewie DFS, $low(u) \geqslant d(v)$

Mostami w grafie są krawędzie drzewowe (u, v), dla których $d(u) \leqslant low(v)$. Wszystkie inne krawędzie w grafie należą do pewnych dwuspójnych składowych, które są rozdzielone między sobą przez mosty oraz punkty artykulacji.

Charakterystyka punktów artykulacji, mostów oraz dwuspójnych składowych jest silnie powiązana ze sobą, co pozwala na prosty sposób wyznaczania wszystkich tych obiektów na raz przy użyciu jednej funkcji. Ze względu jednak na fakt, iż niewiele zadań wymaga wyznaczania wszystkich obiektów — mostów, punktów artykulacji i dwuspójnych składowych, w dalszej części rozdziału przedstawionych zostanie kilka funkcji, służących do wyznaczania poszczególnych elementów niezależnie.

Funkcją służącą do wyznaczania mostów w grafie, jest **void** `Graph<V,E>::Bridges(VPII &res)`. Wypełnia ona przekazany przez referencję wektor parami numerów wierzchołków, które połączone są mostem. Funkcja ta wymaga wzbogacenia wierzchołków grafu o dwa dodatkowe pola — **int** `d` oraz **int** `low`. Jej implementacja przedstawiona jest na listingu 2.23.

Listing 2.23: Implementacja funkcji **void** `Graph<V,E>::Bridges(VII&)`

```
01 int s;
02 VPII *X;
// Funkcja dla każdego mostu w grafie, wstawia do wektora res parę numerów
// wierzchołków połączonych tym mostem
03 void Bridges(VPII & res) {
// Odpowiednia inicjalizacja poszczególnych zmiennych
04    res.clear();
05    X = &res;
06    s = 0;
07    FOREACH(it, g) it->d = -1;
// Przetworzenie wszystkich, jeszcze nieodwiedzonych, wierzchołków w grafie
08    REP(i, SIZE(g)) if (g[i].d == -1) BriSearch(i, -1);
09 }
// Funkcja realizuje przeszukiwanie grafu metodą DFS - odwiedza wierzchołek v,
// gdzie u jest ojcem v w drzewie DFS
10 void BriSearch(int v, int u) {
11    g[v].d = g[v].low = s++;
// Dla każdej krawędzi wychodzącej z wierzchołka v...
12    FOREACH(it, g[v]) {
13       int w = it->v;
// Jeśli wierzchołek w nie był jeszcze odwiedzony, to go odwiedź
```

Listing 2.23: (c.d. listingu z poprzedniej strony)

```
14      if (g[w].d == -1) {
15         BriSearch(w, v);
// Jeśli znaleziono w grafie most, to dodaj go do wyniku
16         if (g[w].low > g[v].d) {
17            X->PB(MP(min(v, w), max(v, w)));
18         } else g[v].low = min(g[v].low, g[w].low);
19      } else if (w != u) g[v].low = min(g[v].low, g[w].d);
20   }
21 }
```

Funkcją służącą do wyznaczania dwuspójnych składowych, jest **void** `Graph<V,E>::Dcc()`. Podobnie jak poprzednio, omawiana funkcja, wymaga wzbogacenia struktury wierzchołka o pola **int** `d` oraz **int** `low`. Dodatkowo, do struktury krawędzi musi zostać dodane pole **int** `dcc` oraz **bool** `bridge`. Dla każdej krawędzi w grafie, algorytm określa, czy jest ona mostem (pole **bool** `bridge`), oraz w przypadku, gdy krawędź nie jest mostem, wylicza numer dwuspójnej składowej, do której ta krawędź należy (pole **int** `dcc`).

Listing 2.24: Implementacja funkcji **void** `Graph<V,E>::Dcc()`

```
// Zmienna id używana jest do numerowania dwuspójnych składowych, natomiast
// l do pamiętania aktualnego czasu odwiedzenia wierzchołków metodą DFS
01 int id, l;
02 typedef typename vector<Ed>::iterator EIT;
03 vector<EIT> qu;
// Makro ustawia odpowiednie wartości zmiennych dcc i bri dla krawędzi
// nieskierowanej, wskazywanej przez iterator e
04 #define DccMark(bri) e->dcc = g[e->v][e->rev].dcc = id, \
05 e->bridge = g[e->v][e->rev].bridge = bri
06 void Dcc() {
// Odpowiednia inicjalizacja zmiennych
07   id = l = 0;
08   qu.clear();
09   FOREACH(it, g) it->d = -1;
// Przetwórz wszystkie, jeszcze nieodwiedzone, wierzchołki w grafie
10   REP(i, SIZE(g)) if (g[i].d == -1) DccSearch(i, -1);
11 }
12 void DccSearch(int v, int u) {
13   EIT e;
// Ustawienie na odpowiednie wartości pól d i low rekordu wierzchołka v
14   g[v].d = g[v].low = l++;
// Dla wszystkich krawędzi, wychodzących z wierzchołka v
15   FOREACH(it, g[v]) {
16      int w = it->v;
// Jeśli wierzchołek docelowy nie został jeszcze odwiedzony...
17      if (g[w].d == -1) {
// Wstaw iterator wskazujący na aktualnie przetwarzaną krawędź na stos
18         qu.PB(it);
// Odwiedź wierzchołek, do którego prowadzi krawędź
```

Listing 2.24: (c.d. listingu z poprzedniej strony)

```
19          DccSearch(w, v);
20          if (g[w].low >= g[v].d) {
// Znaleziono dwuspójną składową - dla każdej krawędzi z tej składowej
// ustaw odpowiednie wartości zmiennych bri oraz dcc
21            int cnt = 0;
22            do {
23              e = qu.back();
24              DccMark(0);
25              qu.pop_back();
26              cnt++;
27            } while (e != it);
// Znaleziony został most
28            if (cnt == 1) DccMark(1);
29            id++;
30          } else g[v].low = min(g[v].low, g[w].low);
31        } else if (g[w].d < g[v].d && w != u)
32          qu.PB(it), g[v].low = min(g[v].low, g[w].d);
33      }
34 }
```

Ostatnią z tej serii funkcją jest **void** `Graph<V,E>::BriArt(VPII&)`, która wymaga wzbogacenia struktury wierzchołków grafu o pola **int** `d` oraz **bool** `art`, a wyznacza dla każdego z nich, czy jest on punktem artykulacji — informacja ta umieszczana jest w polu **bool** `art`. Dodatkowo, funkcja wypełnia wektor przekazany przez referencję parami numerów wierzchołków połączonych mostami (podobnie do funkcji **void** `Graph<V,E>::Bridges(VII&)`).

Listing 2.25: Implementacja funkcji **void** `Graph<V,E>::BriArt(VPII&)`

```
// Zmienna wykorzystywana do przechowywania czasu odwiedzenia wierzchołków przez
// algorytm DFS
01 int t;
// Wskaźnik na konstruowaną listę znalezionych mostów
02 VPII *br;
// Funkcja przeszukuje poddrzewo wierzchołka v metodą DFS (p jest
// wierzchołkiem-ojcem v w drzewie DFS)
03 int BriArtR(int v, int p) {
04   int l = g[v].d = ++t;
// Dla każdej krawędzi wychodzącej z wierzchołka v i nie prowadzącej do
// wierzchołka-ojca w drzewie DFS...
05   FOREACH(it, g[v]) if (it->v != p)
// Aktualizacja wartości funkcji low dla wierzchołka v.
06   l = min(l, !g[it->v].d ? BriArtR(it->v, v) : g[it->v].d);
// Zaktualizowanie informacji o znalezionym punkcie artykulacji
07   if (g[p].d <= l) g[p].art = 1;
// Jeśli został znaleziony most, to jest on dodawany do wyniku
08   if (g[p].d < l) br->PB(MP(min(v, p), max(v, p)));
09   return l;
10 }
```

Listing 2.25: (c.d. listingu z poprzedniej strony)

```
11 void BriArt(VPII & res) {
// Odpowiednia inicjalizacja zmiennych
12   res.clear();
13   br = &res;
14   t = 0;
15   REP(x, SIZE(g)) g[x].art = g[x].d = 0;
// Od każdego, jeszcze nieodwiedzonego wierzchołka, rozpocznij przeszukiwanie w
// głąb
16   REP(x, SIZE(g)) if (!g[x].d) {
17     g[x].d = ++t;
18     int c = 0;
19     FOREACH(it, g[x]) if (!g[it->v].d) {
20       c++;
21       BriArtR(it->v, x);
22     }
// Jeśli z korzenia drzewa DFS wychodzi więcej niż jedna krawędź drzewowa, to
// jest on punktem artykulacji
23     g[x].art = (c > 1);
24   }
25 }
```

Tabela 2.26: Wyniki wyznaczone dla dwóch przykładowych grafów z rysunków 2.7.a oraz 2.7.b przez omówione funkcje

Funkcja	Graf z rysunku 2.7.a	Graf z rysunku 2.7.b
void Graph<V,E>::Bridges(VPII&)	E(3, 4) E(9, 10) E(8, 9) E(2, 6)	E(2, 3) E(2, 6)
void Graph<V,E>::Dcc()	E(1, 2) – 5 E(1, 3) – 5 E(2, 3) – 5 E(2, 6) – 4 (most) E(3, 4) – 0 (most) E(5, 6) – 3 E(5, 7) – 3 E(6, 7) – 3 E(6, 8) – 3 E(7, 8) – 3 E(8, 9) – 2 (most) E(9, 10) – 1 (most)	E(0, 1) – 3 E(0, 2) – 3 E(1, 2) – 3 E(2, 3) – 1 (most) E(2, 6) – 2 (most) E(3, 4) – 0 E(3, 5) – 0 E(4, 5) – 0
void Graph<V,E>::BriArt(VPII&)	E(3, 4) – most E(9, 10) – most E(8, 9) – most E(2, 6) – most 2 – punkt art 3 – punkt art 6 – punkt art 8 – punkt art 9 – punkt art	E(2, 3) – most E(2, 6) – most 2 – punkt art 3 – punkt art

Listing 2.27: Program prezentujący sposób użycia funkcji **void** `Graph<V,E>::Bridges(VII&)`, **void** `Graph<V,E>::Dcc()` oraz **void** `Graph<V,E>::BriArt(VPII&)`. Pełny kod źródłowy tego programu znajduje się w pliku `bri_art_dcc.cpp`

```
// Wzbogacenie struktur wierzchołków oraz krawędzi wymagane
// przez funkcję Bridges
01 struct VsBridges {
02    int d, low;
03 };
04 struct VeBridges {
05    int rev;
06 };
// Wzbogacenie struktury wierzchołków i krawędzi o elementy wymagane
// przez funkcję Dcc
07 struct VsDcc {
08    int d, low;
09 };
10 struct VeDcc {
11    int rev, dcc;
12    bool bridge;
13 };
// Wzbogacenie struktury wierzchołków oraz krawędzi o elementy wymagane
// przez funkcję BriArt
14 struct VsBriArt {
15    int d;
16    bool art;
17 };
18 struct VeBriArt {
19    int rev;
20 };
21 int main() {
22    int n, m, b, e;
// Wczytaj liczbę wierzchołków oraz krawędzi
23    cin >> n >> m;
24    Graph<VsBridges, VeBridges> g1(n);
25    Graph<VsDcc, VeDcc> g2(n);
26    Graph<VsBriArt, VeBriArt> g3(n);
// Dodaj do grafów wszystkie krawędzie
27    REP(x, m) {
28       cin >> b >> e;
29       g1.EdgeU(b, e);
30       g2.EdgeU(b, e);
31       g3.EdgeU(b, e);
32    }
// Zastosuj funkcję Bridges
33    VPII res1;
34    g1.Bridges(res1);
35    FOREACH(it, res1)
36       cout << "E(" << it->ST << "," << it->ND << ")" << endl;
```

Listing 2.27: (c.d. listingu z poprzedniej strony)

```
// Zastosuj funkcję Dcc
37    g2.Dcc();
38    REP(x, SIZE(g2.g)) FOREACH(it, g2.g[x]) if (x < it->v) {
39       if (it->bridge)
40          cout << "(" << x << "," << it->v << ") - most" << endl;
41       cout << "E(" << x << "," << it->v << ") - " <<
42          it->dcc << endl;
43    }
// Zastosuj funkcję BriArt
44    VPII res3;
45    g3.BriArt(res3);
46    FOREACH(it, res3)
47       cout << "E(" << it->ST << ", " << it->ND << ") - most" << endl;
48    REP(x, SIZE(g3.g))
49       if (g3.g[x].art) cout << x << " - punkt art." << endl;
50    return 0;
51 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 315	acm.uva.es - zadanie 10199	spoj.sphere.pl - zadanie 185 spoj.sphere.pl - zadanie 208

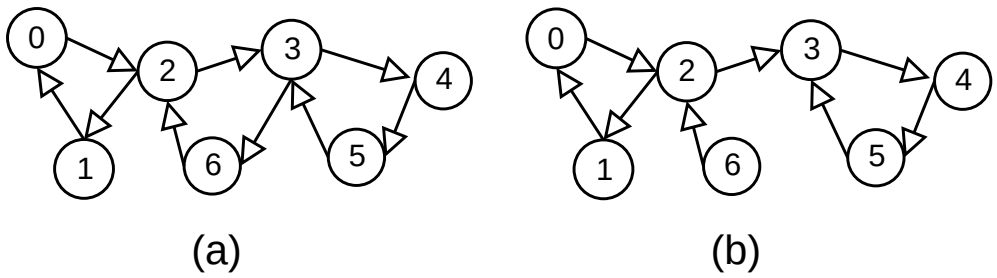
2.8. Ścieżka i cykl Eulera

Literatura
[KDP] - 2.7
[ASD] - 7.4
[MD] - 6.2

Ścieżka Eulera w grafie $G = (V, E)$, jest to dowolna ścieżka przechodząca przez wszystkie krawędzie tego grafu dokładnie raz. W przypadku, gdy pierwszy wierzchołek na ścieżce jest równy wierzchołkowi ostatniemu, to ścieżkę Eulera nazywamy cyklem Eulera. W zależności od przyjętych założeń, ścieżka Eulera musi przechodzić przez wszystkie wierzchołki w grafie, lub też nie przechodzi przez wierzchołki izolowane (czyli takie, z których nie wychodzą żadne krawędzie). W tym rozdziale zakładamy, że wierzchołki izolowane nie muszą być odwiedzone — dzięki takiemu podejściu, przed rozpoczęciem wyszukiwania ścieżki Eulera, nie musimy się martwić o eliminowanie z grafu wierzchołków izolowanych, które często pojawiają się przypadkowo przy szybkim implementowaniu programów podczas zawodów.

Ścieżki oraz cykle Eulera można wyznaczać zarówno dla grafów skierowanych jak i nieskierowanych. Warunki na ich istnienie w grafie (oprócz spójności grafu z pominięciem ewentualnych wierzchołków izolowanych), są następujące:

- W grafie nieskierowanym cykl Eulera istnieje wtedy i tylko wtedy, gdy stopień każdego wierzchołka jest parzysty.
- W grafie skierowanym cykl Eulera istnieje wtedy i tylko wtedy, gdy stopień wejściowy każdego wierzchołka jest równy stopniowi wyjściowemu.



Rysunek 2.8: **(a)** Graf skierowany zawierający cykl Eulera $0 \to 2 \to 3 \to 4 \to 5 \to 3 \to 6 \to 2 \to 1 \to 0$. **(b)** Graf skierowany nieposiadający cyklu Eulera (stopień wejściowy wierzchołka 6 nie jest równy stopniu wyjściowemu). Graf ten zawiera ścieżkę Eulera $6 \to 2 \to 1 \to 0 \to 2 \to 3 \to 4 \to 5 \to 3$.

- W grafie nieskierowanym, ścieżka Eulera, niebędąca cyklem Eulera, istnieje wtedy i tylko wtedy, gdy dwa wierzchołki mają stopień nieparzysty, natomiast wszystkie inne wierzchołki — parzysty.
- W grafie skierowanym, ścieżka Eulera niebędąca cyklem Eulera istnieje wtedy i tylko wtedy, gdy jeden wierzchołek w grafie ma o jeden większy stopień wejściowy od wyjściowego, jeden wierzchołek ma o jeden mniejszy stopień wejściowy od wyjściowego, natomiast reszta wierzchołków ma równe stopnie wejściowe i wyjściowe.

Prezentowane algorytmy wyszukiwania ścieżki Eulera bazują na przeszukiwaniu grafu w głąb. Główna różnica w podejściu, w stosunku do oryginalnego przeszukiwania DFS jest taka, że podczas wyznaczania ścieżki Eulera, wierzchołki mogą być odwiedzane wielokrotnie, natomiast krawędzie — tylko raz. Dla kolejno odwiedzanych wierzchołków, wybierane są krawędzie jeszcze nieodwiedzone. W momencie znalezienia się w wierzchołku, z którego nie da się wyjść (gdyż wszystkie krawędzie z niego wychodzące zostały już przetworzone), algorytm wycofuje się z tego wierzchołka, jednocześnie dodając krawędzie, po których się cofa, do wyznaczanej ścieżki Eulera. Jedyna różnica w sposobie realizacji algorytmu dla grafów nieskierowanych polega na tym, że przechodząc krawędzią (v, u), należy również zaznaczyć krawędź (u, v) jako odwiedzoną.

Wyznaczanie ścieżek Eulera w prosty sposób można zrealizować przy użyciu funkcji rekurencyjnej. Takie podejście jednak nie jest bezpieczne ze względu na możliwość przepełnienia stosu. Prezentowane w tym rozdziale funkcje — **bool** `Graph<V,E>EulerD::(VI&)` oraz **bool** `Graph<V,E>::EulerU(VI&)` są iteracyjną realizacją przedstawionej wyżej koncepcji. Obie funkcje przyjmują jako parametr referencję na wektor zmiennych typu **int**. W przypadku, gdy w grafie nie istnieje ścieżka Eulera, omawiane funkcje zwracają fałsz. Gdy ścieżka istnieje, zwracana jest prawda, natomiast przekazany przez referencję wektor wypełniony zostaje numerami, kolejno odwiedzanych na wyznaczonej ścieżce wierzchołków. W przypadku gdy ścieżka Eulera jest cyklem, pierwszy i ostatni element wektora są równe (ścieżka kończy się w tym samym wierzchołku, co zaczyna). Odpowiednie implementacje przedstawione są na listingach 2.28 oraz 2.29.

Listing 2.28: Implementacja funkcji **bool** `Graph<V,E>::EulerD(VI&)`

```
01 bool EulerD(VI &r) {
// Inicjalizacja wymaganych zmiennych
02    int v = -1, kr = 1, h;
03    r.clear();
```

Listing 2.28: (c.d. listingu z poprzedniej strony)

```
04    VI l, st(SIZE(g), 0);
// Dla wszystkich wierzchołków wyliczany jest stopień wejściowy
05    FOREACH(it, g) FOREACH(it2, *it) ++st[it2->v];
// Należy wyznaczyć wierzchołek v, od którego rozpoczęte
// zostanie wyszukiwanie ścieżki Eulera.
06    REP(x, SIZE(g)) {
// Jeśli wierzchołek ma większy stopień wyjściowy od wejściowego,
// to jest wierzchołkiem startowym
07       if ((h = SIZE(g[x])) > st[x]) v = x; else
// Jeśli wierzchołek ma jakieś wychodzące krawędzie oraz nie znaleziono
// jeszcze wierzchołka początkowego, to użyj go jako wierzchołek startowy
08       if (h && v == -1) v = x;
09       kr += st[x] = h;
10    }
// Konstrukcja ścieżki Eulera jest rozpoczynana w wierzchołku v
11    if (v != -1) l.PB(v);
// Dopóki istnieją wierzchołki na stosie, przeszukuj graf metodą DFS
12    while(SIZE(l)) if (!st[v = l.back()]) {
13       l.pop_back();
14       r.PB(v);
15    } else l.PB(v = g[v][--st[v]].v);
// Wyznaczona ścieżka Eulera została skonstruowana w odwrotnym kierunku,
// więc trzeba ją odwrócić
16    reverse(ALL(r));
// Algorytm zwraca prawdę, jeśli wykorzystane zostały wszystkie krawędzie
// w grafie
17    return SIZE(r) == kr;
18 }
```

Listing 2.29: Implementacja funkcji **bool** `Graph<V,E>::EulerU(VI&)`

```
01 bool EulerU(VI & ce) {
// Inicjalizacja wymaganych zmiennych
02    int v = -1;
03    ce.clear();
04    VI l, st(SIZE(g), 0), of(SIZE(g) + 1, 0);
// Należy wyznaczyć wierzchołek v, od którego rozpoczęte zostanie
// wyszukiwanie ścieżki Eulera
05    REP(x, SIZE(g)) {
06       of[x + 1] = of[x] + (st[x] = SIZE(g[x]));
07       if ((st[x] & 1) || (v == -1 && st[x])) v = x;
08    }
// Wektor służący do odznaczania wykorzystanych krawędzi.
09    vector<bool> us(of[SIZE(g)], 0);
// Konstrukcja ścieżki Eulera jest rozpoczynana w wierzchołku v
10    if (v != -1) l.PB(v);
// Dopóki istnieją wierzchołki na stosie, przeszukuj graf metodą DFS
11    while (SIZE(l)) {
```

Listing 2.29: (c.d. listingu z poprzedniej strony)

```
12      v = l.back();
// Dopóki kolejne krawędzie zostały już przetworzone, pomiń je
13      while (st[v] && us[of[v] + st[v] - 1]) --st[v];
// Jeśli nie ma już więcej krawędzi, to wyjdź z wierzchołka i dodaj krawędź,
// po której się cofasz do wyniku
14      if (!st[v]) {
15         l.pop_back();
16         ce.PB(v);
17      } else {
// Przejdź po jeszcze niewykorzystanej krawędzi
18         int u = g[v][--st[v]].v;
19         us[of[u] + g[v][st[v]].rev] = 1;
20         l.PB(v = u);
21      }
22   }
// Algorytm zwraca prawdę, jeśli wykorzystane zostały wszystkie krawędzie w
// grafie
23   return 2 * (SIZE(ce) - 1) == of[SIZE(g)];
24 }
```

Zarówno w przypadku grafów skierowanych, jak i nieskierowanych, złożoność czasowa procesu wyznaczania ścieżki Eulera to $O(n+m)$ — każda krawędź w grafie przetwarzana jest jednokrotnie.

Dla grafu przedstawionego na rysunku 2.8.a, wywołanie funkcji **bool** `Graph<V,E>::EulerD(VI &l)` spowoduje wypełnienie wektora `l` następującymi liczbami, reprezentującymi numery kolejno odwiedzanych wierzchołków na wyznaczonym cyklu Eulera:

$$\{0, 2, 3, 4, 5, 3, 6, 2, 1, 0\}$$

Gdyby ten sam graf potraktować jako nieskierowany, to zawartość wektora `l` po wywołaniu funkcji **bool** `Graph<V,E>::EulerU(VI &l)` byłaby następująca:

$$\{0, 1, 2, 3, 5, 4, 3, 6, 2, 0\}$$

Po usunięciu z grafu krawędzi łączącej wierzchołki 3 i 6 (graf ten przedstawiony jest na rysunku 2.8.b), wywołanie **bool** `Graph<V,E>::EulerD(VI &l)` spowoduje wypełnienie wektora `l` następującymi liczbami:

$$\{6, 2, 1, 0, 2, 3, 4, 5, 3\}$$

Gdyby ten sam graf potraktować jako nieskierowany, to zawartość wektora `l` po wywołaniu funkcji **bool** `Graph<V,E>::EulerU(VI &l)` byłaby następująca:

$$\{6, 2, 0, 1, 2, 3, 5, 4, 3\}$$

Podane wyniki działania obu funkcji są przykładowe i zależą od kolejności wstawiania krawędzi do grafu.

Listing 2.30: Program prezentujący użycie funkcji **bool** `Graph<V,E>::EulerD(VI &l)`. Pełny kod źródłowy tego programu znajduje się w pliku `eulerpath.cpp`.

```
// Zarówno krawędzie, jak i wierzchołki nie wymagają dodatkowych wzbogaceń
01 struct Ve { };
02 struct Vs { };
03 int main() {
04    VI res;
05    int n, m, b, e;
// Wczytaj liczbę wierzchołków oraz krawędzi w grafie
06    cin >> n >> m;
07    Graph<Vs, Ve> g(n);
// Dodaj do grafu odpowiednie krawędzie
08    REP(x, m) {
09       cin >> b >> e;
10       g.EdgeD(b, e);
11    }
// Jeśli graf zawiera ścieżkę Eulera - wypisz ją, jeśli nie - poinformuj o tym
12    if (g.EulerD(res)) {
13       FOREACH(it, res) cout << *it << " ";
14       cout << endl;
15    } else {
16       cout << "No Euler Path" << endl;
17    }
18    return 0;
19 }
```

Listing 2.31: Program prezentujący użycie funkcji **bool** `Graph<V,E>::EulerU(VI &l)`. Pełny kod źródłowy tego programu znajduje się w pliku `eulerupath.cpp`

```
// Wymagane wzbogacenie krawędzi w grafie nieskierowanym
01 struct Ve {
02    int rev;
03 };
// Wierzchołki nie wymagają dodatkowych wzbogaceń
04 struct Vs { };
05 int main() {
06    VI res;
07    int n, m, b, e;
// Wczytaj liczbę wierzchołków oraz krawędzi w grafie
08    cin >> n >> m;
09    Graph<Vs, Ve> g(n);
// Dodaj do grafu odpowiednie krawędzie
10    REP(x, m) {
11       cin >> b >> e;
12       g.EdgeU(b, e);
13    }
// Jeśli graf zawiera ścieżkę Eulera - wypisz ją, jeśli nie - poinformuj o tym
14    if (g.EulerU(res)) {
15       FOREACH(it, res) cout << *it << " ";
```

Listing 2.31: (c.d. listingu z poprzedniej strony)

```
16        cout << endl;
17    } else {
18        cout << "No Euler Path" << endl;
19    }
20    return 0;
21 }
```

Zadanie: Linie autobusowe

Pochodzenie: Potyczki Algorytmiczne 2005

Rozwiązanie: `buses.cpp`

W Bajtocji jest n miast, połączonych dwukierunkowymi drogami, przy których leżą liczne wioski. Król Bajtazar zdecydował się utworzyć sieć linii autobusowych obsługujących miasta i wioski. Każda linia może się zaczynać i kończyć w dowolnym mieście oraz przebiegać przez dowolne miasta. Miasta na trasie linii mogą się powtarzać, jednak żadna linia nie może przebiegać wielokrotnie tą samą drogą.

Aby wszystkim mieszkańcom zapewnić transport, a jednocześnie zminimalizować koszty inwestycji, król Bajtazar postanowił, że każdą drogę będzie przebiegała dokładnie jedna linia autobusowa, a także, że liczba linii autobusowych będzie minimalna.

Zadanie

Napisz program, który:

- wczyta opis sieci dróg,
- zaprojektuje sieć linii autobusowych spełniającą podane wymagania,
- wypisze wynik.

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite n i m, oddzielone pojedynczym odstępem, $2 \leqslant n \leqslant 10\,000$, $n - 1 \leqslant m \leqslant 200\,000$; n jest liczbą miast, a m liczbą dróg. Miasta są ponumerowane od 1 do n. Kolejnych m wierszy zawiera opis sieci dróg. Każdy z tych wierszy zawiera dwie liczby całkowite a i b, oddzielone pojedynczym odstępem, $1 \leqslant a \leqslant b \leqslant n$ — numery miast połączonych drogą. Każda droga jest podana na wejściu dokładnie raz. Możesz założyć, że dowolne dwa miasta są połączone co najwyżej jedną drogą (chociaż może być wiele tras łączących dwa miasta) i że istnieje możliwość przejazdu pomiędzy dowolnymi dwoma miastami.

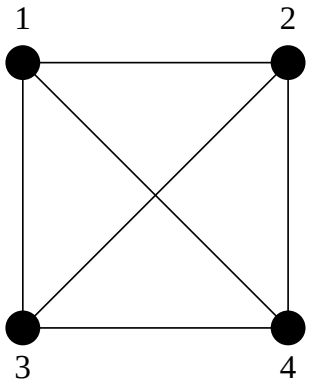
Wyjście

Pierwszy wiersz powinien zawierać liczbę c, równą minimalnej liczbie linii autobusowych. Kolejnych c wierszy powinno zawierać opisy kolejnych linii: $i+1$-szy wiersz powinien zawierać liczbę l_i równą liczbie miast na trasie i-tej linii, a następnie l_i numerów tych miast, podanych w kolejności przebiegu linii. Liczby w wierszach powinny być pooddzielane pojedynczymi odstępami. Jeżeli linia ma swój początek i koniec w tym samym mieście, jego numer powinien się znaleźć na początku i na końcu opisu trasy.

Przykład

Dla następującego wejścia:

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



Poprawnym rozwiązaniem jest:

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

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 117	acm.uva.es - zadanie 10054	acm.uva.es - zadanie 10441
acm.uva.es - zadanie 10129	acm.sgu.ru - zadanie 121	spoj.sphere.pl - zadanie 41
acm.sgu.ru - zadanie 101		acm.sgu.ru - zadanie 286

2.9. Minimalne drzewo rozpinające

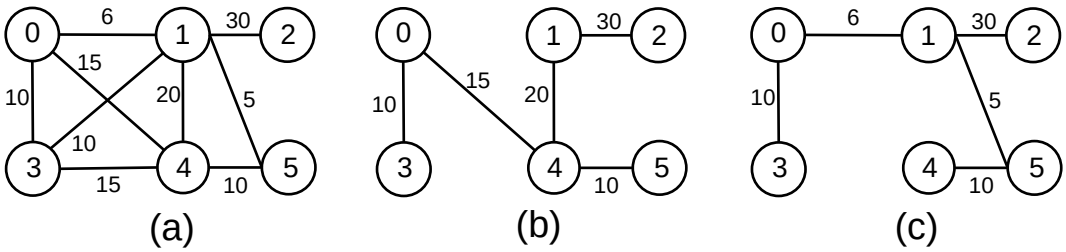
Literatura
[WDA] - 24
[KDP] - 2.4
[ASD] - 7.6
[MD] - 6.6

Rozpatrzmy spójny graf nieskierowany $G = (V, E)$, w którym każdej krawędzi przypisana jest pewna nieujemna waga. Drzewo rozpinające dla grafu G jest to spójny podgraf $G' = (V, E')$, $E' \subseteq E$ grafu G, zawierający dokładnie $|V| - 1$ krawędzi. Minimalne drzewo rozpinające, to takie drzewo rozpinające, którego suma wag krawędzi jest minimalna.

Na rysunku 2.9 przedstawiony jest przykładowy graf oraz dwa jego drzewa rozpinające. Na listingu 2.32 znajduje się implementacja funkcji **int** `Graph<V,E>::MinSpanTree()`, która dla danego spójnego grafu wyznacza minimalne drzewo rozpinające i zwraca jako wynik sumę wag krawędzi należących do tego drzewa. Funkcja ta wymaga, aby dla każdej krawędzi w grafie, jej waga znajdowała się w dodatkowej zmiennej krawędzi **int** `l`. Po zakończeniu działania, pole **bool** `span` każdej krawędzi zawiera wartość prawda, gdy odpowiadająca jej krawędź należy do wyznaczonego minimalnego drzewa rozpinającego.

Funkcja **int** `Graph<V,E>::MinSpanTree()` realizuje algorytm Prima. Na początku konstruowane jest jednowierzchołkowe drzewo rozpinające (wybierany jest w tym celu dowolny wierzchołek grafu wejściowego), a następnie dokładane są kolejne krawędzie o najmniejszej wadze, łączące wierzchołek należący do konstruowanego drzewa, z wierzchołkiem jeszcze do niego nie należącym. Poprawność algorytmu wynika z faktu, iż w każdym kroku algorytm konstruuje minimalne drzewo rozpinające (wybierane są najlżejsze możliwe krawędzie) dla coraz większego zbioru wierzchołków, aż w końcu uzyskiwane jest drzewo rozpinające dla całego grafu G. Złożoność czasowa takiego algorytmu to $O(m * log(n))$.

Innym, równie popularnym jak algorytm Prima, jest algorytm Kruskala, który na początku konstruuje graf składający się tylko z wierzchołków oryginalnego grafu G, a następnie przetwarza wszystkie krawędzie grafu G w kolejności niemalejących wag. Dodaje do drzewa rozpinającego te krawędzie, które łączą różne spójne składowe konstruowanego grafu.



Rysunek 2.9: **(a)** Graf o sześciu wierzchołkach $[0 \ldots 5]$ oraz dziewięciu krawędziach. Wzdłuż krawędzi zaznaczone są ich wagi. **(b)** Drzewo rozpinające dla grafu z rysunku **(a)** o sumarycznej wadze krawędzi 85. **(c)** Minimalne drzewo rozpinające dla grafu z rysunku **(a)** o sumarycznej wadze krawędzi 61. Nie jest to jedyne minimalne drzewo rozpinające — drugie można uzyskać poprzez wymianę krawędzi $(0, 3)$ na $(1, 3)$.

Listing 2.32: Implementacja funkcji **int** `Graph<V,E>::MinSpanTree()`

```
// W polu bool span krawędzi algorytm wstawia wartość prawda gdy krawędź należy
// do wyznaczonego minimalnego drzewa rozpinającego. Funkcja zwraca wagę
// znalezionego drzewa.
01 int MinSpanTree() {
// Tablica d dla każdego wierzchołka, nienależącego jeszcze do drzewa
// rozpinającego, zawiera długość najkrótszej krawędzi łączącej go z dowolnym
// wierzchołkiem drzewa
02   int r = 0, d[SIZE(g)];
// Tablica służąca do odznaczania wierzchołków dodawanych do drzewa
03   bool o[SIZE(g)];
04   REP(x, SIZE(g)) {
05     d[x] = INF;
06     o[x] = 0;
07   }
// Kolejka priorytetowa wierzchołków, osiągalnych z budowanego drzewa, w
// kolejności niemalejących kosztów krawędzi
08   set<PII> s;
09   s.insert(MP(d[0] = 0, 0));
// Dopóki istnieją wierzchołki nie należące do drzewa
10   while (!s.empty()) {
// Wybierz wierzchołek, którego dodanie jest najtańsze
11     int v = (s.begin())->ND;
12     s.erase(s.begin());
13     bool t = 0;
// Zaznacz wierzchołek jako dodany do drzewa oraz zwiększ sumaryczną wagę
// drzewa
14     o[v] = 1;
15     r += d[v];
// Dla wszystkich krawędzi wychodzących z dodawanego wierzchołka...
16     FOREACH(it, g[v]) {
17       it->span = 0;
// Jeśli jest to krawędź, którą dodano do drzewa, to zaznacz ten fakt
18       if (!t && o[it->v] && it->l == d[v])
```

Listing 2.32: (c.d. listingu z poprzedniej strony)

```
19            t = it->span = g[it->v][it->rev].span = 1;
20          else
// Próba zaktualizowania odległości od drzewa dla wierzchołków jeszcze nie
// dodanych...
21          if (!o[it->v] && d[it->v] > it->l) {
22            s.erase(MP(d[it->v], it->v));
23            s.insert(MP(d[it->v] = it->l, it->v));
24          }
25        }
26    }
// Zwróć wagę skonstruowanego drzewa rozpinającego
27    return r;
28 }
```

Listing 2.33: Wynik wygenerowany przez funkcję **int** `Graph<V,E>::MinSpanTree()` dla grafu z rysunku 2.9.a

```
Waga minimalnego drzewa rozpinajacego: 61
Krawedzie nalezace do drzewa: (0,1) (1,2) (1,3) (1,5) (4,5)
```

Listing 2.34: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.33. Pełny kod źródłowy programu znajduje się w pliku `minspantree_str.cpp`

```
// Wzbogacenie krawędzi wymagane przez funkcję wyznaczającą minimalne drzewo
// rozpinające
01 struct Ve {
02    int rev, l;
03    bool span;
04 };
05 struct Vs {};
06 int main() {
07    int n, m, b, e;
// Wczytaj liczbę wierzchołków oraz krawędzi w grafie
08    cin >> n >> m;
09    Ve l;
// Skonstruuj graf o odpowiedniej wielkości i dodaj do niego krawędzie
10    Graph<Vs, Ve> g(n);
11    REP(x,m) {
12       cin >> b >> e >> l.l;
13       g.EdgeU(b, e, l);
14    }
// Wyznacz minimalne drzewo rozpinające
15    cout << "Waga minimalnego drzewa rozpinajacego: " <<
16       g.MinSpanTree() << endl << "Krawedzie nalezace do drzewa:";
// Wypisz wszystkie krawędzie należące do drzewa rozpinającego
17    REP(x, SIZE(g.g)) FOREACH(it, g.g[x]) if (it->span && it->v < x)
18       cout << " (" << it->v << "," << x << ")";
19    cout << endl;
```

Listing 2.34: (c.d. listingu z poprzedniej strony)

```
20    return 0;
21 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 534	acm.uva.es - zadanie 10462	acm.uva.es - zadanie 10147
acm.uva.es - zadanie 10034	acm.uva.es - zadanie 10600	spoj.sphere.pl - zadanie 148
spoj.sphere.pl - zadanie 368	spoj.sphere.pl - zadanie 30	acm.sgu.ru - zadanie 206

2.10. Algorytm Dijkstry

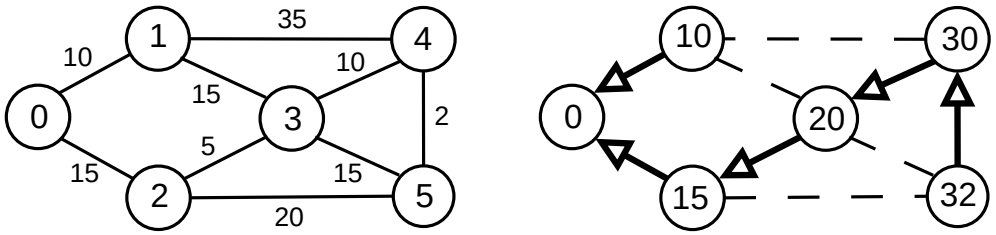
Literatura
[WDA] - 25.2
[KDP] - 3.3
[MD] - 8.3
[CON] - 3.5

Kolejnym problemem jakim się zajmiemy, będzie znajdowanie odległości od zadanego źródła $s \in V$ do wszystkich wierzchołków w ważonym grafie $G = (V, E)$ (skierowanym lub nieskierowanym). Załóżmy, że każdej krawędzi grafu przypisana jest pewna nieujemna waga, reprezentująca jej długość. Odległość między wierzchołkami v i w w takim grafie rozumiana jest jako długość najkrótszej ścieżki $v \to \ldots \to w$, czyli suma wag krawędzi na niej występujących. Przy takiej definicji długości ścieżki, najkrótsza ścieżka między dwoma wierzchołkami nie musi być tą, która składa się z najmniejszej liczby krawędzi. Jako przykład wystarczy rozpatrzyć graf o trzech wierzchołkach v_1, v_2 oraz v_3, który zawiera krawędzie (v_1, v_2) o długości 1, (v_1, v_3) o długości 3, oraz (v_2, v_3) o długości 1. Ścieżka, jaka zostanie wyznaczona przez algorytm BFS dla pary wierzchołków v_1 oraz v_3, będzie składała się z jednej krawędzi (v_1, v_3) o długości 3. Tym czasem istnieje krótsza droga $v_1 \to v_2 \to v_3$ o długości 2.

Do rozwiązania naszego problemu zastosujemy algorytm Dijkstry. Na początku działania ustala on odległości dla wszystkich wierzchołków w grafie na nieskończoność. Wyjątkiem jest wierzchołek startowy, dla którego odległość ustalana jest na 0. W kolejnych krokach wybierany jest wierzchołek v, o którym wiadomo, że wyznaczona dla niego odległość już się nie zmieni (pierwszym takim wierzchołkiem jest v), oraz aktualizowane są odległości do wszystkich wierzchołków, do których istnieje krawędź z v.

Powstaje pytanie — dla jakich wierzchołków od danego momentu działania algorytmu, poszukiwana odległość nie ulegnie już zmianie? Algorytm Dijkstry zakłada, że jest to wierzchołek o najmniejszej aktualnie wyznaczonej odległości spośród wierzchołków jeszcze nie przetworzonych. Założenie takie sprawia, że algorytm Dijkstry działa prawidłowo tylko w przypadku grafów z nieujemnymi wagami krawędzi.

Algorytm Dijkstry zrealizowany jest przez funkcję **void** `Graph<V,E>::Dijkstra(int)`, której kod źródłowy przedstawiony jest na listingu 2.35. Funkcja ta przyjmuje jako parametr numer wierzchołka startowego `s` i dla każdego wierzchołka wyznacza odległość od `s`, która zostaje umieszczona w dodatkowej zmiennej wierzchołka **int** `t`. Funkcja wyznacza również numer wierzchołka-ojca w drzewie najkrótszych ścieżek — umieszcza go w zmiennej **int** `s`. Długości krawędzi grafu należy umieścić w zmiennych **int** `l` krawędzi. Złożoność czasowa to $O(m * log(n))$ — w przypadku szczególnych grafów istnieje możliwość poprawienia tej złożoności. Przykładowo, dla grafów gęstych, w których liczba krawędzi jest rzędu



Rysunek 2.10: **(a)** Przykładowy graf z zaznaczonymi wagami krawędzi. **(b)** Wyznaczone przez algorytm Dijkstry drzewo najkrótszych ścieżek dla źródła znajdującego się w wierzchołku 0. W wierzchołkach umieszczone są odległości.

$O(n^2)$, istnieje możliwość uzyskania złożoności $O(m)$ poprzez zamianę kolejki priorytetowej nieprzetworzonych wierzchołków na listę.

Listing 2.35: Implementacja funkcji **void** `Graph<V,E>::Dijkstra(int)`

```
// Operator określający porządek liniowy (w kolejności rosnących odległości od
// źródła) na wierzchołkach grafu
01 struct djcmp {
02    bool operator() (const Ve * a, const Ve * b) const {
03       return (a->t == b->t) ? a < b : a->t < b->t;
04    }
05 };
// Funkcja realizująca algorytm Dijkstry. Dla każdego wierzchołka wyznaczana jest
// odległość od źródła wyszukiwania s i umieszczana w zmiennej t oraz numer
// wierzchołka-ojca w drzewie najkrótszych ścieżek - umieszczany w zmiennej s.
// Dla wierzchołków nieosiągalnych ze źródła t = INF, s = -1
06 void Dijkstra(int s) {
// Kolejka priorytetowa służąca do wyznaczania najbliższych wierzchołków
07    set<Ve *, djcmp> k;
08    FOREACH(it, g) it->t = INF, it->s = -1;
// Na początku wstawiany jest do kolejki wierzchołek źródło
09    g[s].t = 0;
10    g[s].s = -1;
11    k.insert(&g[s]);
// Dopóki są jeszcze nieprzetworzone wierzchołki...
12    while (!k.empty()) {
// Wybierz wierzchołek o najmniejszej odległości i usuń go z kolejki
13       Ve *y = *(k.begin());
14       k.erase(k.begin());
// Dla każdej krawędzi, wychodzącej z aktualnie przetwarzanego wierzchołka,
// spróbuj zmniejszyć odległość do wierzchołka, do którego ta krawędź prowadzi
15       FOREACH(it, *y) if (g[it->v].t > y->t + it->l) {
16          k.erase(&g[it->v]);
17          g[it->v].t = y->t + it->l;
18          g[it->v].s = y - &g[0];
19          k.insert(&g[it->v]);
20       }
```

Listing 2.35: (c.d. listingu z poprzedniej strony)

```
21    }
22 }
```

Listing 2.36: Wynik wywołania funkcji **void** `Graph<V,E>::Dijkstra(0)` dla grafu przedstawionego na rysunku 2.10.a

```
Wierzcholek 0: odleglosc od zrodla = 0, ojciec w drzewie najkrotszych sciezek = -1
Wierzcholek 1: odleglosc od zrodla = 10, ojciec w drzewie najkrotszych sciezek = 0
Wierzcholek 2: odleglosc od zrodla = 15, ojciec w drzewie najkrotszych sciezek = 0
Wierzcholek 3: odleglosc od zrodla = 20, ojciec w drzewie najkrotszych sciezek = 2
Wierzcholek 4: odleglosc od zrodla = 30, ojciec w drzewie najkrotszych sciezek = 3
Wierzcholek 5: odleglosc od zrodla = 32, ojciec w drzewie najkrotszych sciezek = 4
```

Listing 2.37: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.36. Pełny kod źródłowy programu znajduje się w pliku `dijkstra_str.cpp`

```
// Wzbogacenie wierzchołków oraz krawędzi wymagane przez algorytm Dijkstry
01 struct Vs {
02    int t, s;
03 };
04 struct Ve {
05    int l, rev;
06 };
07 int main() {
08    int n, m, s, b, e;
09    Ve ed;
// Wczytaj liczbę wierzchołków, krawędzi oraz wierzchołek startowy
10    cin >> n >> m >> s;
// Skonstruuj graf o odpowiednim rozmiarze oraz dodaj do niego krawędzie
11    Graph<Vs, Ve> g(n);
12    REP(x,m) {
13       cin >> b >> e >> ed.l;
14       g.EdgeU(b, e, ed);
15    }
// Wykonaj algorytm Dijkstry
16    g.Dijkstra(s);
// Wypisz dla wszystkich wierzchołków znalezioną odległość od źródła
// oraz numer wierzchołka, z którego prowadzi ostatnia krawędź
// wyznaczonej najkrótszej ścieżka
17    REP(x, SIZE(g.g)) cout << "Wierzcholek " << x <<
18       ": odleglosc od zrodla = " << g.g[x].t <<
19       ", ojciec w drzewie najkrotszych sciezek = " << g.g[x].s << endl;
20    return 0;
21 }
```

Zadanie: Przemytnicy

Pochodzenie: X Olimpiada Informatyczna

Rozwiązanie: `alchemy.cpp`

Bajtocja słynie z bogatych złóż złota, dlatego przez długie lata kwitła sprzedaż tego kruszcu do sąsiedniego królestwa, Bitlandii. Niestety, powiększająca się ostatnio dziura budżetowa, zmusiła króla Bitlandii do wprowadzenia zaporowych ceł na metale i minerały. Handlarze przekraczający granicę muszą zapłacić cło w wysokości 50% wartości przewożonego ładunku. Bajtockim kupcom grozi bankructwo.

Na szczęście bajtoccy alchemicy opracowali sposoby pozwalające zamieniać pewne metale w inne. Pomysł kupców polega na tym, aby z pomocą alchemików zamieniać złoto w pewien tani metal, a następnie, po przewiezieniu go przez granicę i zapłaceniu niewielkiego cła, znowu otrzymywać z niego złoto. Niestety alchemicy nie znaleźli sposobu na zamianę dowolnego metalu w dowolny inny. Może się więc zdarzyć, że proces otrzymania danego metalu ze złota musi przebiegać wielostopniowo i że na każdym etapie uzyskiwany będzie inny metal. Alchemicy każą sobie słono płacić za swoje usługi i dla każdego znanego sobie procesu zamiany metalu A w metal B wyznaczyli cenę za przemianę 1 kg surowca. Handlarze zastanawiają się, w jakiej postaci należy przewozić złoto przez granicę oraz jaki ciąg procesów alchemicznych należy zastosować, aby zyski były możliwie największe. Pomóż uzdrowić bajtocką gospodarkę!

Zadanie

Napisz program, który:

- Wczyta tabelę cen wszystkich metali, a także ceny przemian oferowanych przez alchemików,
- Wyznaczy taki ciąg metali $m_0, m_1, \ldots, m_k$, że:
  - $m_0 = m_k$ to złoto,
  - dla każdego $i = 1, 2, \ldots, k$ alchemicy potrafią otrzymać metal m_i z metalu $m_i - 1$ oraz,
  - koszt wykonania całego ciągu procesów alchemicznych dla 1 kg złota, powiększony o płacone na granicy cło (50% ceny 1 kg najtańszego z metali m_i, dla $i = 0, 1, \ldots, k$) jest najmniejszy z możliwych.

  Zakładamy, że podczas procesów alchemicznych waga metali nie zmienia się.
- Wypisze koszt wykonania wyznaczonego ciągu procesów alchemicznych powiększony o płacone na granicy cło.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna dodatnia liczba całkowita n oznaczająca liczbę rodzajów metali, $1 \leqslant n \leqslant 5\,000$. W wierszu o numerze $k+1$, dla $1 \leqslant k \leqslant n$, znajduje się nieujemna parzysta liczba całkowita p_k — cena 1 kg metalu oznaczonego numerem k, $0 \leqslant p_k \leqslant 10^9$. Przyjmujemy, że złoto ma numer 1. W wierszu o numerze $n + 2$ znajduje się jedna nieujemna liczba całkowita m równa liczbie procesów przemiany znanych alchemikom, $0 \leqslant m \leqslant 100\,000$. W każdym z kolejnych m wierszy znajdują się po trzy liczby naturalne, pooddzielane pojedynczymi odstępami, opisujące kolejne procesy przemiany. Trójka liczb a, b, c oznacza, że alchemicy potrafią z metalu o numerze a otrzymywać metal o numerze b i za zamianę 1 kg surowca każą sobie płacić c bajtalarów, $1 \leqslant a, b \leqslant n, 0 \leqslant c \leqslant 10\,000$.

Uporządkowana para liczb a i b może się pojawić w danych co najwyżej jeden raz.

Wyjście

Twój program powinien pisać na standardowe wyjście. W pierwszym wierszu powinna zostać wypisana jedna liczba całkowita — koszt wykonania wyznaczonego ciągu procesów alchemicznych powiększony o płacone na granicy cło.

Przykład

Dla następującego wejścia:

```
4
200
100
40
2
6
1  2  10
1  3  5
2  1  25
3  2  10
3  4  5
4  1  50
```

Poprawnym rozwiązaniem jest:

```
60
```

Ćwiczenia

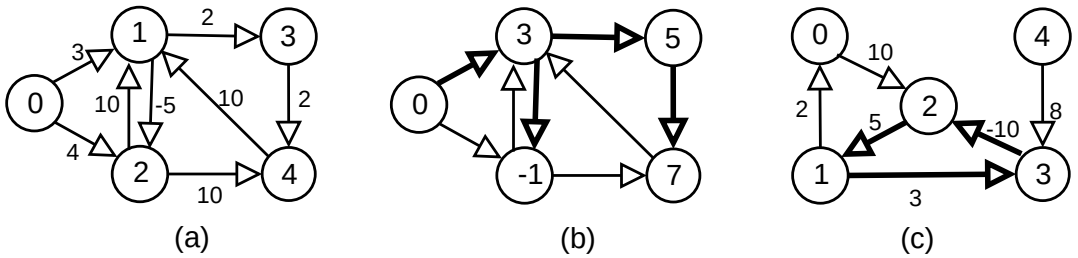
Proste	Średnie	Trudne
acm.uva.es - zadanie 336	acm.uva.es - zadanie 429	acm.uva.es - zadanie 10171
acm.uva.es - zadanie 10048	acm.uva.es - zadanie 10917	spoj.sphere.pl - zadanie 25
acm.uva.es - zadanie 10099	spoj.sphere.pl - zadanie 15	spoj.sphere.pl - zadanie 145
spoj.sphere.pl - zadanie 50	spoj.sphere.pl - zadanie 119	spoj.sphere.pl - zadanie 391

2.11. Algorytm Bellmana-Forda

Literatura
[WDA] - 25.3
[CON] - 3.6

Opisany w poprzednim rozdziale algorytm Dijkstry pozwala na wyznaczanie najkrótszych ścieżek z zadanego wierzchołka w grafie. Niestety, nie działa on poprawnie w przypadku pojawienia się krawędzi o ujemnych wagach, z którymi wiążą się dodatkowe problemy. Na rysunku 2.11.c przedstawiony jest graf zawierający cykl o ujemnej sumie wag krawędzi na nim leżących. W takim przypadku, dla pewnych wierzchołków w grafie nie istnieje najkrótsza ścieżka, gdyż dla dowolnej ścieżki można wyznaczyć krótszą (przechodzącą odpowiednią liczbę razy wzdłuż ujemnego cyklu).

W przypadku rozpatrywania grafów zawierających krawędzie o ujemnych wagach, należy zrezygnować z algorytmu Dijkstry i sięgnąć po inny — niestety wolniejszy — algorytm Bellmana-Forda. Algorytm ten, podobnie jak algorytm Dijkstry, na początku ustala odległość od źródła dla wszystkich wierzchołków na nieskończoność (za wyjątkiem wierzchołka startowego, dla którego odległość ta jest ustalana na 0). Następnie algorytm przetwarza wszystkie krawędzie (u, v) grafu, aktualizując odległości wierzchołków zgodnie z regułą: $t(v) = min(t(v), t(u) + l(u, v))$, gdzie $l(u, v)$ jest długością krawędzi, natomiast $t(v)$ — aktualną odległością wierzchołka v od źródła. Dopóki proces aktualizacji wprowadza jakieś zmiany,



Rysunek 2.11: **(a)** Przykładowy graf skierowany z zaznaczonymi wagami krawędzi. **(b)** Wyznaczone przez algorytm Bellmana-Forda najkrótsze ścieżki dla źródła znajdującego się w wierzchołku 0. W wierzchołkach umieszczone są odległości. **(c)** Graf zawierający cykl $1 \rightarrow 3 \rightarrow 2$ o ujemnej wadze. Funkcja **bool** `Graph<V,E>::BellmanFord(int)` zwróci dla tego grafu prawdę.

jest on powtarzany, ale nie więcej niż n razy. Jeśli za n-tym razem wartości $t(v)$ nadal ulegają zmianom, oznacza to że graf zawiera cykl o ujemnej wadze (w grafach nie zawierających takich cykli każda najkrótsza ścieżka składa się z co najwyżej $n-1$ krawędzi). Złożoność czasowa tego algorytmu to $O(n*(n+m))$ — cały graf jest przetwarzany co najwyżej n-krotnie.

Funkcja **bool** `Graph<V,E>::BellmanFord(int)`, której implementację można znaleźć na listingu 2.38, jest realizacją wyżej opisanej metody. Funkcja ta, w przypadku gdy graf, dla którego została ona wywołana zawiera cykl o ujemnej wadze, zwraca wartość prawda. W przypadku, gdy takiego cyklu nie ma, funkcja zwraca fałsz, oraz wyznacza dla każdego wierzchołka w grafie dwie wartości — **int** `t` oraz **int** `s`, reprezentujące odpowiednio obliczoną odległość od źródła oraz numer wierzchołka, z którego prowadzi ostatnia krawędź najkrótszej ścieżki. Algorytm wymaga, aby długości krawędzi zostały umieszczone w dodatkowych polach **int** `l` krawędzi.

Listing 2.38: Implementacja funkcji **bool** `Graph<V,E>::BellmanFord(int)`

```
01 bool BellmanFord(int v) {
// Inicjalizacja zmiennych
02   FOREACH(it, g) it->t = INF;
03   g[v].t = 0;
04   g[v].s = -1;
05   int change, cnt = 0;
// Dopóki przebieg pętli poprawia wyznaczane odległości, ale pętla nie została
// wykonana więcej niż SIZE(g) razy...
06   do {
07     change = 0;
// Dla każdej krawędzi (v,u) zaktualizuj odległość do wierzchołka u
08     REP(i, SIZE(g)) FOREACH(it, g[i])
09     if (g[i].t + it->l < g[it->v].t) {
10       g[it->v].t = g[i].t + it->l;
11       g[it->v].s = i;
12       change = 1;
13     }
14   } while (change && cnt++ < SIZE(g));
// Jeśli wagi cały czas ulegały zmianom, to oznacza, że w grafie istnieje cykl o
// ujemnej wadze
15   return change;
```

Listing 2.38: (c.d. listingu z poprzedniej strony)

```
16 }
```

Listing 2.39: Dla grafu przedstawionego na rysunku 2.11.a, wartości wyliczonych przez wywołanie funkcji **bool** `Graph<V,E>::BellmanFord(0)` zmiennych będą następujące

```
Wierzcholek 0: odleglosc od zrodla = 0, ojciec w drzewie najkrotszych sciezek = -1
Wierzcholek 1: odleglosc od zrodla = 3, ojciec w drzewie najkrotszych sciezek = 0
Wierzcholek 2: odleglosc od zrodla = -2, ojciec w drzewie najkrotszych sciezek = 1
Wierzcholek 3: odleglosc od zrodla = 5, ojciec w drzewie najkrotszych sciezek = 1
Wierzcholek 4: odleglosc od zrodla = 7, ojciec w drzewie najkrotszych sciezek = 3
```

Listing 2.40: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.39. Pełny kod źródłowy programu znajduje się w pliku `bellman_ford_str.cpp`

```
// Wzbogacenie struktury wierzchołków oraz krawędzi wymagane przez
// algorytm Bellmana-Forda
01 struct Ve {
02    int l;
03 };
04 struct Vs {
05    int t, s;
06 };
07
08 int main() {
09    int n, m, s, b, e;
10    Ve ed;
// Wczytaj liczbę wierzchołków, krawędzi oraz wierzchołek źródłowy
11    cin >> n >> m >> s;
// Skonstruuj graf o odpowiedniej wielkości oraz dodaj do niego
// wszystkie krawędzie
12    Graph<Vs, Ve> g(n);
13    REP(x, m) {
14       cin >> b >> e >> ed.l;
15       g.EdgeD(b, e, ed);
16    }
// Wykonaj algorytm Bellmana-Forda
17    g.BellmanFord(s);
// Wypisz wynik
18    REP(x, SIZE(g.g))
19       cout << "Wierzcholek " << x << ": odleglosc od zrodla = " <<
20       g.g[x].t << ", ojciec w drzewie najkrotszych sciezek = " <<
21       g.g[x].s << endl;
22    return 0;
23 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 104	acm.uva.es - zadanie 10746	acm.uva.es - zadanie 10449
acm.uva.es - zadanie 558	acm.uva.es - zadanie 10806	

2.12. Maksymalny przepływ

Literatura
[WDA] - 27
[KDP] - 4.1, 4.2
[CON] - 4

Rozpatrywaliśmy już przykład modelowania sieci miejskich dróg przy użyciu grafu. Wyobraźmy sobie teraz, że każda droga łączy ze sobą pewne dwa skrzyżowania, oraz ma określoną maksymalną przepustowość (liczbę samochodów, które mogą nią przejeżdżać w jednostce czasu). Nasuwa się oczywiste pytanie, jaka jest maksymalna liczba samochodów, które mogą przejeżdżać z pewnego startowego skrzyżowania do skrzyżowania docelowego w jednostce czasu. Innymi słowy, jaka jest maksymalna przepustowość sieci dróg pomiędzy tymi dwoma skrzyżowaniami. Należy na wstępnie zauważyć, że samochody nie muszą poruszać się po tej samej ścieżce — niektóre mogą podróżować po najkrótszej trasie, inne mogą jechać drogą okrężną. Naszkicowane tu zagadnienie nazywane jest problemem wyznaczania maksymalnego przepływu w grafie.

Istnieją różne modyfikacje problemu maksymalnego przepływu oraz różne metody pozwalające na jego wyznaczanie. Można rozważać przepływy, których każda krawędź ma jednostkową przepustowość, analizować maksymalny przepływ, którego koszt utrzymania jest minimalny pod pewnym względem... W kilku kolejnych rozdziałach przedstawimy różne algorytmy rozwiązujące te problemy.

Dla każdej krawędzi $k = (u, v)$, poprzez c_k będziemy oznaczali jej przepustowość, natomiast przez f_k — wielkość aktualnie przedostającego się przez nią przepływu. W analizie algorytmów przepływowych, ważną rolę odgrywają trzy pojęcia — krawędź nasycona i nienasycona oraz ścieżka powiększająca:

Definicja 2.12.1 Krawędź nasycona *k, to taka krawędź, której przepustowość c_k jest równa aktualnie przedostającemu się przez nią przepływowi f_k ($c_k = f_k$).*

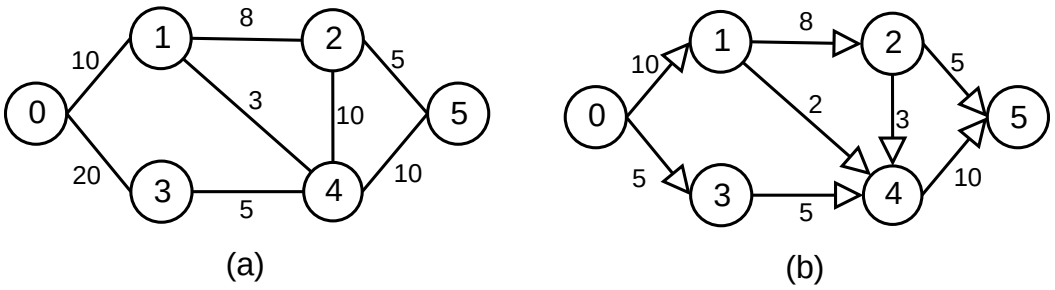
Definicja 2.12.2 Krawędź nienasycona *k, to taka krawędź, której przepustowość c_k jest większa od aktualnie przedostającego się przez nią przepływu f_k ($c_k > f_k$).*

Definicja 2.12.3 Ścieżka powiększająca *$u \rightsquigarrow v$, to taka ścieżka, której wszystkie krawędzie są nienasycone — istnieje możliwość przesłania wzdłuż niej dodatkowego przepływu z wierzchołka u do wierzchołka v.*

Podstawą, na jakiej bazuje wiele algorytmów wyznaczania maksymalnego przepływu jest to, że jeśli w grafie nie istnieje ścieżka powiększająca łącząca źródło i ujście, to aktualny przepływ jest maksymalny. Dopóki wyznaczony przepływ nie jest maksymalny, algorytmy starają się wyznaczać kolejne ścieżki powiększające, wzdłuż których można przesłać dodatkowy przepływ. W zależności od sposobu wyszukiwania kolejnych ścieżek powiększających, otrzymywane są algorytmy o różnych złożonościach czasowych.

2.12.1. Maksymalny przepływ metodą Dinica

Algorytm Dinica służy do wyznaczania maksymalnego przepływu w czasie $O(n^2 * m)$. Pomysł jest następujący — dopóki w grafie istnieje ścieżka powiększająca między źródłem a ujściem:



Rysunek 2.12: **(a)** Graf nieskierowany o sześciu wierzchołkach. Przy krawędziach zaznaczone są ich przepustowości. **(b)** Maksymalny przepływ wyznaczony dla sieci przedstawionej na rysunku **(a)**. Skierowanie krawędzi reprezentuje kierunek przepływu, natomiast wartość przy krawędzi reprezentuje wielkość przepływu. Jak widać z rysunku, krawędzie $(0, 3)$, $(1, 4)$ oraz $(2, 4)$ są nienasycone — ich przepustowość jest większa od wartości przepływu przez nie płynącego.

- dokonuje się podziału zbioru wierzchołków grafu na warstwy. W kolejnych warstwach umieszczane są wierzchołki o coraz większych odległościach od wierzchołka źródłowego (odległość między wierzchołkami u i v rozumiana jest jako liczba krawędzi znajdujących się na najkrótszej ścieżce $u \rightsquigarrow v$). Do warstwy zerowej należy wierzchołek źródłowy, do warstwy pierwszej — wierzchołki połączone krawędzią ze źródłem, ..., do warstwy k-tej — ujście. Podział na warstwy może być zrealizowany poprzez przeszukiwanie grafu wszerz w czasie $O(n + m)$.

- wyznacza się wszystkie ścieżki powiększające między źródłem a ujściem, których wierzchołki należą do kolejnych warstw (0, 1, ... k). W ten sposób wszystkie wyznaczone w jednej fazie algorytmu ścieżki powiększające mają taką samą długość k. Dla każdej krawędzi (u, v), przez którą został zwiększony przepływ, zmniejsza się o tyle samo przepływ w krawędzi (v, u) (w kolejnych fazach może się okazać, że wyznaczenie maksymalnego przepływu wymaga cofnięcia części aktualnego przepływu, co realizowane jest w algorytmie przez wprowadzenie ujemnego przepływu o przeciwnym zwrocie). Faza wyznaczania pojedynczej ścieżki powiększającej w przedstawionej tu implementacji realizowana jest poprzez przeszukiwanie grafu w głąb.

- usuwa się z grafu skierowane krawędzie nasycone, aby nie przetwarzać niepotrzebnie krawędzi, które nie mogą należeć do kolejnych ścieżek powiększających.

Funkcja **int** `Graph<V,E>::MaxFlow(`**int, int**`)` przyjmuje jako parametry odpowiednio numery wierzchołków stanowiących źródło oraz ujście wyznaczanego przepływu, a zwraca wielkość tego przepływu. Dodatkowo, dla każdej krawędzi w grafie wyznaczana jest wartość **int** `f`, która równa jest wielkości przepływu w tej krawędzi (w przypadku ujemnej wartości przepływu, jest to przepływ skierowany w przeciwną stronę). Do działania funkcja wymaga wzbogacenia struktury wierzchołków o pole **int** `t` (jest ono wykorzystywane przez fazę przeszukiwania wszerz oraz w głąb) oraz krawędzi o pole **int** `c` — przepustowości krawędzi, oraz **int** `f` — wyznaczona wartość przepływu. Grafy, na jakich operuje omawiana funkcja są nieskierowane, jednak istnieje możliwość ustawiania różnych przepustowości krawędzi w różne strony — przed przystąpieniem do wyznaczania przepływu należy jedynie zmodyfikować odpowiednio wartości pól **int** `c` krawędzi.

Listing 2.41: Implementacja funkcji **int** `Graph<V,E>::MaxFlow(int, int)`

```
// Zmienna out reprezentuje numer wierzchołka-źródła
int out;
#define ITER typename vector<Ed>::iterator
// Wektor itL zawiera dla każdego wierzchołka wskaźnik na aktualnie
// przetwarzaną krawędź
vector<ITER> itL;
VI vis;
// Funkcja wykorzystuje czasy odwiedzenia wierzchołków z tablicy vis do
// wyznaczania ścieżek poszerzających
int FlowDfs(int x, int fl) {
  int r = 0, f;
// Jeśli aktualny wierzchołek jest ujściem, lub nie można powiększyć przepływu,
// to zwróć aktualny przepływ
  if (x == out || !fl) return fl;
// Przetwórz kolejne krawędzie wierzchołka w celu znalezienia ścieżki
// poszerzającej
  for (ITER & it = itL[x]; it != g[x].end(); ++it) {
// Jeśli krawędź nie jest nasycona i prowadzi między kolejnymi warstwami...
    if (vis[x] + 1 == vis[it->v] && it->c - it->f) {
// Wyznacz wartość przepływu, który można przeprowadzić przez przetwarzaną
// krawędź oraz zaktualizuj odpowiednie zmienne
      it->f += f = FlowDfs(it->v, min(fl, it->c - it->f));
      g[it->v][it->rev].f -= f;
      r += f;
      fl -= f;
// Jeśli nie można powiększyć przepływu to przerwij
      if (!fl) break;
    }
  }
  return r;
}
// Funkcja wyznacza maksymalny przepływ między wierzchołkami s oraz f
int MaxFlow(int s, int f) {
// Inicjalizacja zmiennych
  int res = 0, n = SIZE(g);
  vis.resize(n);
  itL.resize(n);
  out = f;
  REP(x, n) FOREACH(it, g[x]) it->f = 0;
  int q[n], b, e;
  while (1) {
// Ustaw wszystkie wierzchołki jako nieodwiedzone
    REP(x, n) vis[x] = -1, itL[x] = g[x].begin();
// Wykonaj algorytm BFS zaczynając ze źródła s i analizując tylko
// nienasycone krawędzie
    for (q[vis[s] = b = e = 0] = s; b <= e; ++b)
    FOREACH(it, g[q[b]]) if (vis[it->v] == -1 && it->c > it->f)
```

Listing 2.41: (c.d. listingu z poprzedniej strony)

```
30          vis[q[++e] = it->v] = vis[q[b]] + 1;
// Jeśli nie istnieje ścieżka do ujścia f, to przerwij działanie
31      if (vis[f] == -1) break;
// Zwiększ aktualny przepływ
32      res += FlowDfs(s, INF);
33    }
34    return res;
35 }
```

Listing 2.42: Dla grafu przedstawionego na rysunku 2.12.a, wywołanie funkcji **int** `Graph<V,E>::MaxFlow(0, 5)` wyznaczy następującą wartość maksymalnego przepływu

```
Wielkosc calkowitego przeplywu: 15
Wartosci przeplywu dla kolejnych krawedzi:
f(0, 1) = 10
f(0, 3) = 5
f(1, 2) = 7
f(1, 4) = 3
f(2, 4) = 2
f(2, 5) = 5
f(3, 4) = 5
f(4, 5) = 10
```

Listing 2.43: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.42. Pełny kod źródłowy programu znajduje się w pliku `maxflow.cpp`

```
// Wzbogacenie struktury wierzchołków oraz krawędzi o elementy wymagane przez
// algorytm Dinica
01 struct Ve {
02    int rev, c, f;
03 };
04 struct Vs {
05    int t;
06 };
07 int main() {
08    int n, m, s, f, b, e;
// Wczytaj liczbę wierzchołków i krawędzi w grafie oraz źródło i ujście
// wyznaczanego przepływu
09    cin >> n >> m >> s >> f;
// Skonstruuj graf o odpowiedniej wielkości, a następnie dodaj do niego
// wszystkie krawędzie
10    Graph<Vs, Ve> g(n);
11    Ve l;
```

Listing 2.43: (c.d. listingu z poprzedniej strony)

```
12    REP(x, m) {
13       cin >> b >> e >> l.c;
14       g.EdgeU(b, e, l);
15    }
// Wypisz wielkość przepływu między źródłem a ujściem
16    cout << "Wielkosc calkowitego przeplywu: " << g.MaxFlow(s, f) << endl;
17    cout << "Wartosci przeplywu dla kolejnych krawedzi:" << endl;
18    REP(x, SIZE(g.g)) FOREACH(it, g.g[x]) if (it->f > 0)
19       cout << "f(" << x << ", " << it->v << ") = " << it->f << endl;
20    return 0;
21 }
```

2.12.2. Maksymalny przepływ dla krawędzi jednostkowych

W wielu zadaniach, związanych z wyznaczaniem maksymalnego przepływu, pojawiają się różne dodatkowe ograniczenia, które niekiedy ułatwiają bądź utrudniają rozwiązywany problem. Jednym z takich ograniczeń może być założenie, że przez każdą krawędź może przepływać co najwyżej jednostkowa wielkość przepływu. Problem tego typu nazywać będziemy poszukiwać maksymalnego jednostkowego przepływu (z krawędziami o jednostkowej przepustowości). Algorytm z poprzedniego rozdziału jak najbardziej nadaje się do rozwiązywania tego problemu, jednak istnieje nie tylko prostszy, ale również i szybszy algorytm.

Przedstawiony w tym rozdziale algorytm służy do wyznaczania maksymalnego jednostkowego przepływu w czasie $O(n^{8/3})$ dla grafów skierowanych. Podobnie jak w algorytmie Dinica, pomysł polega na wyszukiwaniu kolejnych ścieżek powiększających. Fakt, iż każda krawędź na wyznaczanych ścieżkach powiększających ma taką samą niewykorzystaną przepustowość, pozwala na prosty sposób usuwania krawędzi nasyconych, oraz dodawania ich odpowiedników o przeciwnym zwrocie. Wyznaczanie ścieżek powiększających realizowane jest przez dwie fazy — przeszukiwanie BFS, które dokonuje podziału wierzchołków na warstwy, a następnie przeszukiwanie w głąb, na skutek którego odwracane są skierowania krawędzi należących do znalezionych ścieżek powiększających. W przypadku, gdy w grafie nie istnieje już więcej ścieżek powiększających prowadzących ze źródła do ujścia, algorytm kończy działanie, zwracając jako wynik liczbę wyznaczonych dotychczas ścieżek. Przykład działania został przedstawiony na rysunku 2.13. Funkcja **int** `UnitFlow(`**int, int**`)` z listingu 2.44 realizuje opisany algorytm, przyjmując jako parametry numery wierzchołków stanowiących odpowiednio źródło oraz ujście wyznaczanego przepływu. Jako wynik zwracana jest wielkość maksymalnego jednostkowego przepływu. Funkcja wymaga wzbogacenia struktury wierzchołków o pola **int** `t` oraz **int** `s`.

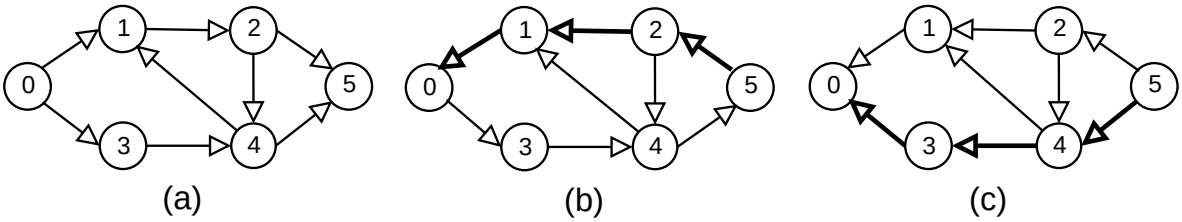
W przypadku korzystania z tej funkcji należy pamiętać, że struktura grafu ulega modyfikacji — zostaje zmienione skierowanie krawędzi wchodzących w skład ścieżek powiększających. Funkcja może być wykorzystywana zarówno dla grafów skierowanych, jak i nieskierowanych (w tym drugim przypadku jednak, modyfikacje wykonane na grafie zaburzają reprezentację krawędzi nieskierowanych przy użyciu pól **int** `rev`).

Listing 2.44: Implementacja funkcji **int** `Graph<V,E>::UnitFlow(`**int, int**`)`

```
// Funkcja odwraca skierowanie krawędzi e wychodzącej z wierzchołka v
01 void mvFlow(int v, Ed & e) {
02    int u = e.v;
```

Listing 2.44: (c.d. listingu z poprzedniej strony)

```
03    g[u].PB(e);
04    g[u].back().v = v;
05    swap(g[v].back(), e);
06    g[v].pop_back();
07 }
08 int Ue;
// Funkcja szuka ścieżki prowadzącej do wierzchołka Ue (ujścia) przy użyciu
// przeszukiwania w głąb
09 bool UFDfs(int v) {
// Jeśli wierzchołek jest ujściem, to została znaleziona ścieżka poszerzająca
10    if (v == Ue) return true;
11    g[v].s = 1;
// Dla każdej krawędzi wychodzącej z wierzchołka...
12    FOREACH(it, g[v])
// Jeśli łączy ona kolejne warstwy oraz wierzchołek docelowy nie był jeszcze
// odwiedzony, to odwiedź go...
13    if (g[it->v].t == 1 + g[v].t && !g[it->v].s && UFDfs(it->v)) {
// W przypadku znalezienia ścieżki poszerzającej, zamień skierowanie krawędzi
14       mvFlow(v, *it);
15       return true;
16    }
17    return false;
18 }
// Właściwa funkcja wyznaczająca maksymalny przepływ jednostkowy między
// wierzchołkami v1 i v2
19 int UnitFlow(int v1, int v2) {
20    int res = 0;
21    Ue = v2;
22    while (1) {
// Wyznacz drzewo przeszukiwania BFS
23       Bfs(v1);
// Jeśli ujście nie zostało odwiedzone, to nie da się powiększyć przepływu
24       if (g[v2].t == -1) break;
25       FOREACH(it, g) it->s = 0;
// Dla każdej krawędzi wychodzącej z wierzchołka źródłowego, jeśli istnieje
// ścieżka poszerzająca zawierająca tę krawędź, to powiększ przepływ
26       FOREACH(it, g[v1]) if (UFDfs(it->v)) {
27          res++;
28          mvFlow(v1, *it--);
29       }
30    }
31    return res;
32 }
```



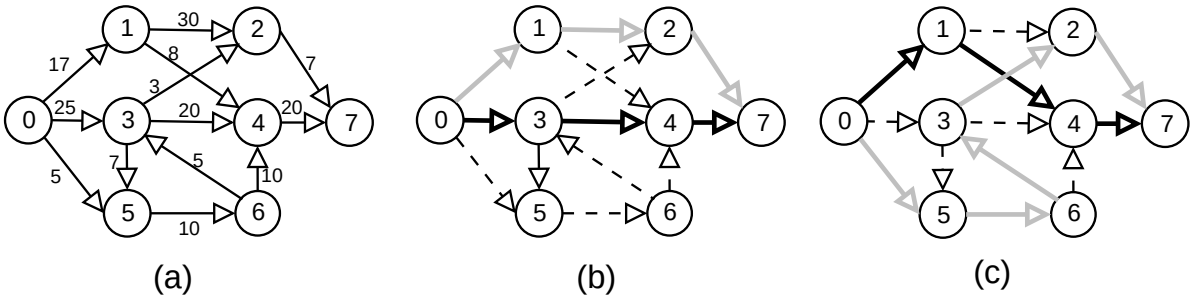
Rysunek 2.13: Przykład wyznaczania maksymalnego jednostkowego przepływu dla grafu z rysunku **(a)** pomiędzy wierzchołkiem numer 0, a wierzchołkiem numer 5. **(b)** Stan skierowania krawędzi grafu po wyznaczeniu pierwszej ścieżki powiększającej $0 \to 1 \to 2 \to 5$. **(c)** Stan skierowania krawędzi po wyznaczeniu drugiej ścieżki powiększającej $0 \to 3 \to 4 \to 5$. W grafie nie ma już więcej ścieżek $0 \rightsquigarrow 5$, zatem wyznaczony przepływ jest maksymalny.

Listing 2.45: Wartość wyznaczonego maksymalnego przepływu dla grafu z rysunku 2.13.a między wierzchołkami 0 i 5 przez funkcję `Graph<V,E>::UnitFlow(int, int)`

```
Przeplyw miedzy wierzcholkami 0 i 5 wynosi 2
```

Listing 2.46: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.45. Pełny kod źródłowy programu znajduje się w pliku `unitflow.cpp`

```
01 struct Ve {};
// Wzbogacenie struktury wierzchołków o pole wymagane przez funkcję UnitFlow
02 struct Vs {
03    int t,s;
04 };
05 int main() {
06    int n, m, s, f, b, e;
// Wczytaj liczbę wierzchołków i krawędzi w grafie oraz numer wierzchołka
// początkowego i końcowego
07    cin >> n >> m >> s >> f;
// Skonstruuj graf o odpowiednim rozmiarze oraz dodaj do niego wymagane krawędzie
08    Graph<Vs, Ve> g(n);
09    REP(x,m) {
10       cin >> b >> e;
11       g.EdgeD(b, e);
12    }
// Wypisz wielkość wyznaczonego maksymalnego przepływu
13    cout << "Przeplyw miedzy wierzcholkami " << s << " i " << f <<
14       " wynosi " << g.UnitFlow(s, f) << endl;
15    return 0;
16 }
```



Rysunek 2.14: **(a)** Skierowany graf z kosztami przypisanymi krawędziom. **(b)** Wyznaczony maksymalny przepływ o wielkości 2 i koszcie 109 dla krawędzi o wagach jednostkowych. **(c)** Maksymalny przepływ o wielkości 2 i minimalnym koszcie 75.

2.12.3. Najtańszy maksymalny przepływ dla krawędzi jednostkowych

Literatura
[CON] - 4.7

Na problem wyznaczania maksymalnego przepływu z poprzedniego rozdziału można narzucić dodatkowe wymaganie. Załóżmy, że dla wszystkich krawędzi analizowanej sieci dodane zostały nieujemne koszty. Koszt utrzymania krawędzi jest równy iloczynowi jej kosztu oraz wielkości przepływu. Kosztem utrzymania całej sieci jest suma kosztów utrzymania wszystkich krawędzi. Nowo postawiony problem polega na wyznaczeniu w sieci maksymalnego przepływu jednostkowego, którego koszt jest minimalny.

Algorytm, realizowany przez prezentowaną funkcję `PII Graph<V,E>::MinCostFlow(int, int)` z listingu 2.47 działa w czasie $O(n*m*u)$, gdzie u jest wielkością wyznaczanego maksymalnego przepływu. Pierwszym oraz drugim parametrem funkcji są odpowiednio źródło oraz ujście. Funkcja jako wynik swojego działania zwraca parę liczb naturalnych — wielkość wyznaczonego maksymalnego przepływu oraz jego koszt. Do działania funkcji, wymagane jest wzbogacenie struktury wierzchołków grafu o dodatkowe pola **int** `t` oraz **int** `s`, natomiast krawędzi o pole **int** `l`, w którym umieszcza się koszt krawędzi.

Zasada działania algorytmu jest podobna do innych algorytmów wyznaczających maksymalny przepływ. Tym razem jednak, w celu zapewnienia minimalności kosztu, do wyznaczania ścieżek powiększających wykorzystywany jest algorytm Bellmana-Forda. Zauważmy, że jeśli poszukujemy przepływu jednostkowego o minimalnym koszcie, którego wielkość wynosi 1 (przepływ taki składa się z dokładnie jednej ścieżki powiększającej), to ścieżkę powiększającą możnaby wyznaczyć przy użyciu algorytmu Dijkstry — sumaryczny koszt utrzymania krawędzi na ścieżce można potraktować jako jej długość. Zatem najkrótsza znaleziona ścieżka między źródłem a ujściem jest zgodnie z naszą definicją najtańsza.

Po wyznaczeniu pierwszej ścieżki powiększającej, należy zamienić zwroty oraz koszty krawędzi do niej należących na przeciwne, a następnie przystąpić do wyszukiwania kolejnych ścieżek powiększających. Ze względu jednak na zamianę kosztów krawędzi, w grafie pojawią się krawędzie z ujemnymi wagami, co powoduje, że algorytm Dijkstry nie może zostać już użyty — zamiast niego można jednak skorzystać z algorytmu Bellmana-Forda.

Korzystając z funkcji `PII Graph<V,E>::MinCostFlow(int, int)` należy pamiętać o tym, że dokonuje ona modyfikacji struktury grafu poprzez zamiany zwrotów krawędzi oraz zmiany ich kosztów.

Listing 2.47: Implementacja funkcji `PII Graph<V,E>::MinCostFlow(int, int)`

```
// Funkcja wyznacza wielkość oraz koszt maksymalnego, jednostkowego przepływu o
// minimalnym koszcie w sieci między wierzchołkami v1 oraz v2. Na skutek jej
// działania graf podlega modyfikacjom.
PII MinCostFlow(int v1, int v2) {
  int n = SIZE(g);
  PII res = MP(0, 0);
  vector < typename vector<Ed>::iterator > vit(SIZE(g));
  while (1) {
// Ustaw aktualne odległości dla wszystkich wierzchołków (poza źródłem) na
// INF oraz zaznacz wierzchołek v1 jako źródło wyszukiwania
    FOREACH(it, g) it->t = INF;
    g[v1].t = 0;
    g[v1].s = -1;
// Wykonaj co najwyżej SIZE(g) faz wyznaczania najkrótszych ścieżek metodą
// Bellmana-Forda
    for (int chg = 1, cnt = 0; chg && ++cnt < SIZE(g);) {
      chg = 0;
      REP(i, SIZE(g)) FOREACH(it, g[i])
      if (g[i].t + it->l < g[it->v].t) {
        g[it->v].t = g[i].t + it->l;
        g[it->v].s = i;
        vit[it->v] = it;
        chg = 1;
      }
    }
// Jeśli nie wyznaczono ścieżki między v1 a v2, to przerwij
    if (g[v2].t == INF) break;
// Zwiększ wynik
    res.ST++;
    res.ND += g[v2].t;
// Odwróć skierowanie krawędzi na wyznaczonej ścieżce oraz zmień znaki wag
// krawędzi na przeciwne
    int x = v2;
    while (x != v1) {
      int v = g[x].s;
      swap(*vit[x], g[v].back());
      Ed e = g[v].back();
      e.l *= -1;
      e.v = v;
      g[x].PB(e);
      g[x = v].pop_back();
    }
  }
  return res;
}
```

Listing 2.48: Wielkość oraz koszt maksymalnego przepływu o minimalnym koszcie wyznaczonego przez funkcję `PII Graph<V,E>::MinCostFlow(int, int)` wykonanej dla grafu z rysunku 2.14, źródła 0 i ujścia 7.

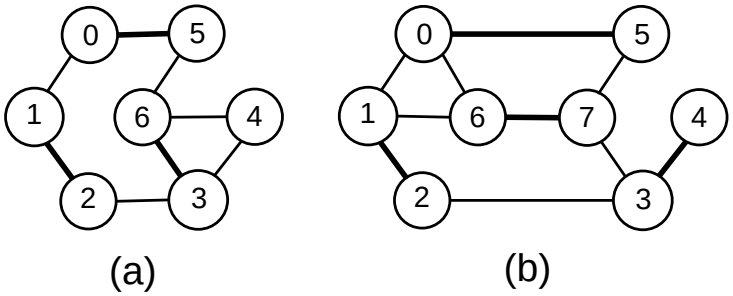
```
Wyznaczanie przeplywu z 0 do 7
Wielkosc przeplywu: 2, koszt przeplywu: 75
```

Listing 2.49: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.48. Pełny kod źródłowy programu znajduje się w pliku `mincostflow.cpp`

```
// Wzbogacenie dla wierzchołków oraz krawędzi wymagane przez funkcję
// MinCostFlow
01 struct Vs {
02    int t, s;
03 };
04 struct Ve {
05    int l;
06 };
07 int main() {
08    int n, m, s, f, b, e;
// Wczytaj liczbę wierzchołków i krawędzi oraz źródło i ujście wyznaczanego
// przepływu
09    cin >> n >> m >> s >> f;
// Skonstruuj graf o odpowiedniej liczbie wierzchołków oraz dodaj do niego
// wymagane krawędzie
10    Graph<Vs, Ve> g(n);
11    Ve l;
12    REP(x, m) {
13       cin >> b >> e >> l.l;
14       g.EdgeD(b, e, l);
15    }
// Wyznacz maksymalny, najtańszy przepływ oraz wypisz wynik
16    cout << "Wyznaczanie przeplywu z " << s << " do " << f << endl;
17    PII res = g.MinCostFlow(s, f);
18    cout << "Wielkosc przeplywu: " << res.ST;
19    cout << ", koszt przeplywu: " << res.ND << endl;
20    return 0;
21 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10092 acm.uva.es - zadanie 10330 acm.sgu.ru - zadanie 194	acm.uva.es - zadanie 10480 acm.sgu.ru - zadanie 176	acm.uva.es - zadanie 10546 acm.sgu.ru - zadanie 212



Rysunek 2.15: **(a)** Przykładowy graf o maksymalnym skojarzeniu wielkości 3. Krawędzie należące do przykładowego maksymalnego skojarzenia zostały pogrubione. **(b)** Graf o doskonałym skojarzeniu wielkości 4.

2.13. Maksymalne skojarzenie w grafie dwudzielnym

Literatura
[WDA] - 27.3
[KDP] - 4.3

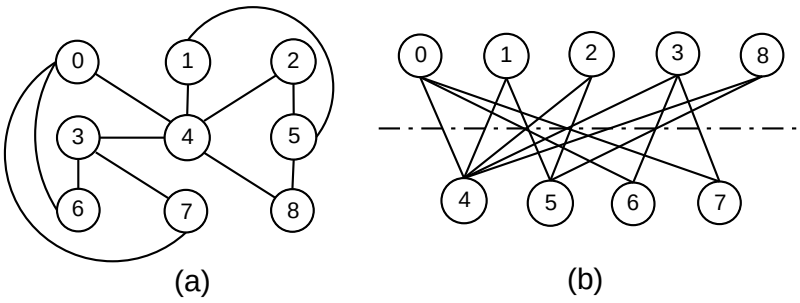
Problem znajdowania maksymalnego skojarzenia w grafie nieskierowanym $G = (V, E)$ polega na wyznaczeniu największego ze względu na liczność zbioru krawędzi $V' \subseteq V$, takiego aby żadne dwie krawędzie nie miały wspólnego końca. W przypadku, gdy wszystkie wierzchołki grafu są końcami pewnych krawędzi ze skojarzenia (wielkość skojarzenia jest wtedy równa $\frac{n}{2}$), skojarzenie takie nazywamy doskonałym. Na rysunku 2.15 przedstawione są przykładowe grafy wraz z wyznaczonymi maksymalnymi skojarzeniami.

Znalezienie takiego zbioru krawędzi w dowolnym grafie nie jest rzeczą łatwą (wprawdzie istnieje algorytm działający w czasie $O(m * \sqrt{n})$, jednak jego implementacja jest nie trywialna). Na szczęście istnieją dosyć ciekawe klasy grafów, dla których rozwiązanie okazuje się istotnie prostsze. Jedną z takich klas są grafy dwudzielne — w ich przypadku, problem maksymalnego skojarzenia można sprowadzić do wyznaczania maksymalnego przepływu jednostkowego.

W kolejnych podrozdziałach przedstawimy prosty algorytm służący do wyznaczania maksymalnego skojarzenia w grafach dwudzielnych w czasie $O(n * (n + m))$, a następnie pokażemy implementację algorytmu Hopcrofta-Karpa, wyznaczającego maksymalne skojarzenie w czasie $O((n + m) * \sqrt{n})$. Zanim jednak to nastąpi, przybliżymy pojęcie grafów dwudzielnych.

2.13.1. Dwudzielność grafu

Graf $G = (V, E)$ nazywamy dwudzielnym, gdy zbiór jego wierzchołków V można podzielić na dwa rozłączne zbiory V_1 oraz V_2, $V_1 \cup V_2 = V$, w taki sposób, aby wszystkie krawędzie grafu $(u, v) \in E$ prowadziły między wierzchołkami z różnych zbiorów ($u \in V_1$, $v \in V_2$ lub $v \in V_1$, $u \in V_2$). Podział taki jest możliwy dla każdego grafu niezawierającego cyklu o nieparzystej długości. Wiele algorytmów skonstruowanych z myślą o grafach dwudzielnych, przed przystąpieniem do rozwiązania właściwego zagadnienia, musi wyznaczyć podział (V_1, V_2). Na listingu 2.50 przedstawiona jest implementacja funkcji **bool** `Graph<V,E>::BiPart(vector<bool>&)`, służąca do wyznaczania tego podziału. Funkcja przyjmuje jako parametr referencję do wektora zmiennych logicznych, a zwraca prawdę, jeśli graf, dla którego została wywołana jest dwudzielny. Wtedy również przekazany przez referencję wektor wypełniany jest wartościami logicznymi — k-ta z tych wartości reprezentuje przynależność k-tego wierzchołka grafu do zbioru V_1. Działanie funkcji opiera się na algorytmie sortowania topologicznego grafu



Rysunek 2.16: **(a)** Graf dwudzielny o dziewięciu wierzchołkach i dwunastu krawędziach. **(b)** Podział zbioru wierzchołków grafu z rysunku **(a)** na dwa rozłączne zbiory $\{0, 1, 2, 3, 8\}$ oraz $\{4, 5, 6, 7\}$, taki że między dowolnymi dwoma wierzchołkami w tym samym zbiorze nie istnieje żadna krawędź.

— posortowane wierzchołki są przetwarzane w kolejności wyznaczonego porządku i zachłannie umieszczane w jednym ze zbiorów V_1 lub V_2. Złożoność czasowa algorytmu wynosi zatem $O(n + m)$. Ze względu na fakt, iż grafy dla których wyznaczany jest podział przy użyciu funkcji **bool** `Graph<V,E>::BiPart(vector <bool>&)` są nieskierowane, może pojawić się obawa o poprawność algorytmu — wykorzystywane jest bowiem sortowanie topologiczne działające poprawnie dla grafów skierowanych. Istotną obserwacją pozwalającą na uzasadnienie poprawności algorytmu jest fakt, iż faza rozdzielająca wierzchołki grafu na dwa zbiory V_1 i V_2 korzysta z tylko jednej własności porządku wyznaczonego przez sortowanie topologiczne — jeśli kolejno przetwarzany wierzchołek nie został jeszcze przydzielony do żadnego ze zbiorów, to jest on pierwszym wierzchołkiem analizowanym w obrębie swojej spójnej składowej, a zatem można go przydzielić do dowolnego ze zbiorów (algorytm przydziela go do zbioru V_2).

Listing 2.50: Implementacja funkcji **bool** `Graph<V,E>::BiPart(vector<bool>&)`

```
01 bool BiPart(vector<char> &v) {
// Inicjalizacja zmiennych
02   v.resize(SIZE(g), 2);
// Wykonaj sortowanie topologiczne grafu. W grafie mogą występować cykle, ale
// nie stanowi to problemu
03   VI r = TopoSortV();
// Dla każdego wierzchołka w grafie
04   FOREACH(x, r) {
// Jeśli wierzchołek nie był jeszcze odwiedzony, to przydzielany jest on do
// pierwszego zbioru wierzchołków
05     if (v[*x] == 2) v[*x] = 0;
// Przetwórz każdego sąsiada aktualnego wierzchołka - jeśli nie był on jeszcze
// odwiedzony, to przydziel go do innego zbioru niż wierzchołek x, a jeśli
// był odwiedzony i jest w tym samym zbiorze co x, to graf nie jest
// dwudzielny
06     FOREACH(it, g[*x]) if (v[it->v] == 2) v[it->v] = 1 - v[*x];
07     else if (v[it->v] == v[*x]) return 0;
08   }
09   return 1;
10 }
```

Przedstawiona funkcja **bool** `Graph<V,E>::BiPart(vector<bool>&)` będzie wykorzystywana przez algorytmy wyznaczania maksymalnego skojarzenia w grafach dwudzielnych. Oryginalne algorytmy, pochodzące z biblioteczki algorytmicznej, nie wymagają implementacji tej funkcji, gdyż wychodzą one z założenia, że wierzchołki w grafie o numerach $0 \ldots \frac{n}{2} - 1$ należą do zbioru V_1, podczas gdy wierzchołki $\frac{n}{2} \ldots n-1$ — do zbioru V_2. Takie założenie jest wygodne w przypadku większości zadań, gdyż podczas konstrukcji grafu zazwyczaj z góry wiadomo, które wierzchołki należą do którego zbioru. Odpowiednia numeracja wierzchołków pozwala na skrócenie kodu implementowanego rozwiązania zadania.

Listing 2.51: Podział wierzchołków grafu z rysunku 2.16.a wyznaczony przez funkcję **bool** `Graph<V,E>::BiPart(vector<bool>&l)`

```
Wierzcholek 0 nalezy do V0
Wierzcholek 1 nalezy do V0
Wierzcholek 2 nalezy do V0
Wierzcholek 3 nalezy do V0
Wierzcholek 4 nalezy do V1
Wierzcholek 5 nalezy do V1
Wierzcholek 6 nalezy do V1
Wierzcholek 7 nalezy do V1
Wierzcholek 8 nalezy do V0
```

Listing 2.52: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.51. Pełny kod źródłowy programu znajduje się w pliku `bipart_str.cpp`

```
01 struct Ve {
02    int rev;
03 };
// Wzbogacenie struktury wierzchołków wymagane przez funkcję Bipart
04 struct Vs {
05    int t, s;
06 };
07 int main() {
08    int n, m, b, e;
09    cin >> n >> m;
// Skonstruuj graf o n wierzchołkach i m krawędziach
10    Graph<Vs, Ve> g(n);
11    REP(x, m) {
12       cin >> b >> e;
13       g.EdgeU(b, e);
14    }
15    vector<char> l;
// Wypisz wynik wyznaczony przez funkjcę Bipart
16    if (g.BiPart(l)) {
17       REP(x, SIZE(l)) cout << "Wierzcholek " << x <<
18          " nalezy do V" << ((int) l[x]) << endl;
19    }
20    return 0;
21 }
```

2.13.2. Maksymalne skojarzenie w grafie dwudzielnym w czasie $O(n*(n+m))$

Bardzo ważnym pojęciem, w kontekście maksymalnego skojarzenia, jest ścieżka naprzemienna, która pełni podobną rolę co ścieżka powiększająca w przypadku maksymalnego przepływu. Niech dany będzie graf $G = (V, E)$ oraz zbiór $E' \subseteq E$, stanowiący skojarzenie w grafie G (niekoniecznie maksymalne). Ścieżką naprzemienną S nazywamy ścieżkę postaci $s_1 \to s_2 \to \ldots \to s_k$, taką że s_1 oraz s_k nie są wierzchołkami skojarzonymi, oraz krawędzie $(s_{2*l}, s_{2*l+1}) \in E'$, $l \in \{1, 2, \ldots, \frac{k}{2} - 1\}$. Przykład ścieżki naprzemiennej o długości 5 przedstawiony jest na rysunku 2.17.d.

Pierwszą, jaką zaprezentujemy funkcją służącą do wyznaczania maksymalnego skojarzenia w grafie dwudzielnym, jest **bool** `Graph<V,E>::BipMatching()`, której kod źródłowy przedstawiony jest na listingu 2.53. W przypadku, gdy graf nie jest dwudzielny, funkcja ta zwraca fałsz. W przeciwnym razie, dla każdego wierzchołka v w grafie wyznaczana jest wartość zmiennej **int** `m` — numer wierzchołka skojarzonego z v. W przypadku wierzchołków nieskojarzonych, wyznaczona wartość to -1. Funkcja wymaga wzbogacenia struktury wierzchołków o dodatkowe pola **int** `m` oraz **int** `t`.

Metoda działania algorytmu polega na znajdowaniu kolejnych ścieżek naprzemiennych (podobnie jak w przypadku maksymalnego przepływu i ścieżki poszerzającej, skojarzenie jest maksymalne, gdy w grafie nie ma więcej ścieżek naprzemiennych). W każdym pojedynczym przebiegu algorytmu, przy użyciu przeszukiwania grafu w głąb wyznaczana jest ścieżka naprzemienna, a następnie zamieniana jest przynależność do skojarzenia wszystkich krawędzi leżących na tej ścieżce. Pojedyncze przeszukiwanie zwiększa wielkość wyznaczonego skojarzenia o 1. Ponieważ wielkość maksymalnego skojarzenia w grafie jest ograniczona z góry przez $\frac{n}{2}$, zatem złożoność algorytmu to $O(n * (n + m))$.

Listing 2.53: Implementacja funkcji **bool** `Graph<V,E>::BipMatching()`

```
// Funkcja realizująca wyszukiwanie ścieżki naprzemiennej w grafie przy użyciu
// przeszukiwania w głąb
01 bool MDfs(int x) {
// Jeśli wierzchołek nie został jeszcze odwiedzony...
02   if (!g[x].t) {
03     g[x].t = 1;
// Dla każdej krawędzi wychodzącej z wierzchołka, jeśli koniec krawędzi nie
// jest skojarzony lub istnieje możliwość wyznaczenia rekurencyjnie ścieżki
// naprzemiennej...
04     FOREACH(it, g[x]) if (g[it->v].m == -1 || MDfs(g[it->v].m)) {
// Skojarz wierzchołki wzdłuż aktualnie przetwarzanej krawędzi
05       g[g[it->v].m = x].m = it->v;
06       return true;
07     }
08   }
09   return false;
10 }
// Funkcja wyznacza maksymalne skojarzenie w grafie dwudzielnym. Umieszcza ona w
// polu m każdego wierzchołka numer wierzchołka z nim skojarzonego (lub -1
// dla wierzchołków nieskojarzonych).
11 bool BipMatching() {
```

Listing 2.53: (c.d. listingu z poprzedniej strony)

```
12    vector<char> l;
// Jeśli graf nie jest dwudzielny, zwróć fałsz
13    if (!BiPart(l)) return 0;
// Inicjalizacja zmiennych
14    int n = SIZE(g), p = 1;
15    FOREACH(it, g) it->m = -1;
// Dopóki istnieje ścieżka naprzemienna...
16    while (p) {
17       p = 0;
18       FOREACH(it, g) it->t = 0;
// Wykonaj przeszukiwanie w głąb w celu znalezienia ścieżki naprzemiennej
19       REP(i, n) if (l[i] && g[i].m == -1) p |= MDfs(i);
20    }
21    return 1;
22 }
```

Listing 2.54: Maksymalne skojarzenie wyznaczone przez funkcję **bool** `Graph<V,E>::BipMatching()` dla grafu z rysunku 2.17.a

```
Wierzcholek 0 skojarzono z 3
Wierzcholek 1 skojarzono z 2
Wierzcholek 4 skojarzono z 5
```

Listing 2.55: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.54. Pełny kod źródłowy programu znajduje się w pliku `bipmatch.cpp`

```
01 struct Ve {
02    int rev;
03 };
// Wzbogacenie wierzchołków grafu wymagane przez funkcję BipMatching
04 struct Vs {
05    int m, t;
06 };
07 int main() {
08    int n, m, s, b, e;
// Skonstruuj odpowiedni graf na podstawie danych wejściowych
09    cin >> n >> m;
10    Graph<Vs, Ve> g(n);
11    REP(x, m) {
12       cin >> b >> e;
13       g.EdgeU(b, e);
14    }
// Wyznacz maksymalne skojarzenie oraz wypisz wynik
15    if (g.BipMatching()) REP(x, SIZE(g.g)) if (g.g[x].m > x)
16       cout << "Wierzcholek " << x << " skojarzono z " << g.g[x].m << endl;
17    return 0;
18 }
```

2.13.3. Maksymalne skojarzenie w grafie dwudzielnym w czasie $O((n+m)*\sqrt{n})$

Przedstawiona w tym rozdziale implementacja algorytmu Hopcrofta-Karpa wyznacza maksymalne skojarzenie w grafie dwudzielnym, sprowadzając problem do wyznaczania maksymalnego jednostkowego przepływu w grafie. Niech zbiory V_1 oraz V_2 stanowią podział dwudzielny zbioru wierzchołków grafu $G = (V, E)$, dla którego wyznaczane jest maksymalne skojarzenie. Nasz algorytm modyfikuje graf G, dodając do niego dwa specjalne wierzchołki — źródło oraz ujście. Wierzchołek-źródło łączony jest ze wszystkimi wierzchołkami ze zbioru V_1, natomiast wszystkie wierzchołki ze zbioru V_2 łączone są z ujściem. Przykładowa konstrukcja tego typu została przedstawiona na rysunku 2.18. Po wyznaczeniu maksymalnego przepływu między źródłem a ujściem, krawędzie oryginalnego grafu G, przez które realizowany jest jednostkowy przepływ, należą do wyznaczonego maksymalnego skojarzenia. Do wyznaczenia maksymalnego przepływu można wykorzystać dowolny algorytm, jednak jeżeli zastosujemy algorytm do wyznaczania jednostkowego maksymalnego przepływu realizowany przez funkcję **int** `Graph<V,E>::UnitFlow(int,int)`, to otrzymamy algorytm o złożoności czasowej $O((n+m)*\sqrt{n})$.

Opisany algorytm realizowany jest przez funkcję `VI Graph<V,E>::Hopcroft()` z listingu 2.56. Funkcja ta zwraca jako wynik wektor liczb całkowitych o długości n. Dla każdego wierzchołka v odpowiadająca mu liczba reprezentuje numer wierzchołka, z którym v został skojarzony. W przypadku, gdy wierzchołek v nie został skojarzony, odpowiadającą liczbą jest -1.

Listing 2.56: Implementacja funkcji `VI Graph<V,E>::Hopcroft()`

```
// UWAGA: Na skutek działania algorytmu graf ulega modyfikacji
VI Hopcroft() {
// Inicjalizacja zmiennych
  int n = SIZE(g);
  VI res(n, -1);
  vector<char> l;
// Jeśli graf nie jest dwudzielny, to algorytm zwraca puste skojarzenie
  if (!BiPart(l)) return res;
// Do grafu dodawane są dwa wierzchołki, jeden z nich jest łączony ze wszystkimi
// wierzchołkami z pierwszego zbioru wyznaczonego przez funkcję BiPart,
// natomiast drugi z wierzchołkami z drugiego zbioru.
  g.resize(n + 2);
  REP(i, n) if (!l[i]) EdgeD(n, i);
  else EdgeD(i, n + 1);
// Wyznaczany jest przepływ jednostkowy w zmodyfikowanym grafie
  UnitFlow(n, n + 1);
// Skojarzenie jest rekonstruowane na podstawie wyniku wyliczonego przez
// algorytm wyznaczający przepływ jednostkowy
  REP(i, n) if (l[i] && g[i][0].v != n + 1)
    res[res[g[i][0].v] = i] = g[i][0].v;
  return res;
}
```

Listing 2.57: Wynik wygenerowany przez funkcję `VI Graph<V,E>::Hopcroft()` dla grafu z rysunku 2.18.a

```
Wierzcholek 0 skojarzono z 3
Wierzcholek 1 skojarzono z 2
Wierzcholek 4 skojarzono z 5
```

Listing 2.58: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.58. Pełny kod źródłowy programu znajduje się w pliku `hopcroft_str.cpp`

```
01 struct Ve { };
// Wzbogacenie wierzchołków grafu wymagane przez funkcję Hopcroft
02 struct Vs {
03    int t, s;
04 };
05 int main() {
06    int n, m, b, e;
// Skonstruj graf o odpowiednim rozmiarze oraz dodaj do niego wymagane krawędzie
07    cin >> n >> m;
08    Graph<Vs, Ve> g(n);
09    REP(x, m) {
10       cin >> b >> e;
11       g.EdgeD(b, e);
12    }
// Wykonaj algorytm Hopcrofta oraz wypisz wynik
13    VI res = g.Hopcroft();
14    REP(x, SIZE(res)) if (res[x] > x)
15       cout << "Wierzcholek " << x << " skojarzono z " << res[x] << endl;
16    return 0;
17 }
```

Zadanie: Skoczki

Pochodzenie: Olimpiada Informatyczna krajów Bałtyckich 2001

Rozwiązanie: `knights.cpp`

Dana jest szachownica o rozmiarach $n * n$, z której usunięto pewną liczbę pól. Zadanie polega na wyznaczeniu maksymalnej liczby skoczków, które można ustawić na planszy w ten sposób, aby żadne dwa z nich nie biły się.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis szachownicy z usuniętymi polami,
- wyznaczy maksymalną liczbę skoczków, które można ustawić na szachownicy w ten sposób, żeby żadne dwa nie biły się,
- wypisze wynik na standardowe wyjście

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n oraz m ($1 \leqslant n \leqslant 200$, $0 \leqslant m \leqslant n^2$), n jest rozmiarem szachownicy, a m to liczba usuniętych pól. Każdy z kolejnych m wierszy zawiera dwie liczby x oraz y ($1 \leqslant x, y \leqslant n$) — współrzędne usuniętego pola. Współrzędne górnego-lewego pola szachownicy to $(1, 1)$, natomiast dolnego-prawego to (n, n). Usunięte pola nie powtarzają się na liście.

Wyjście

W pierwszym i jedynym wierszu wyjścia należy wypisać liczbę — maksymalna liczba skoczków, których da się umieścić na planszy, tak, aby żadne dwa się nie biły.

Przykład

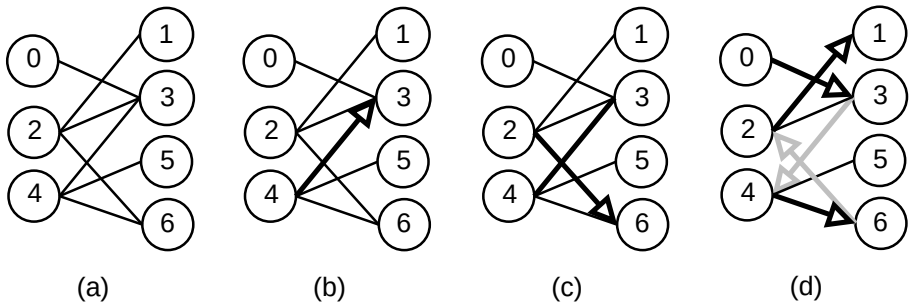
Dla następującego wejścia:

```
3 2
1 1
3 3
```

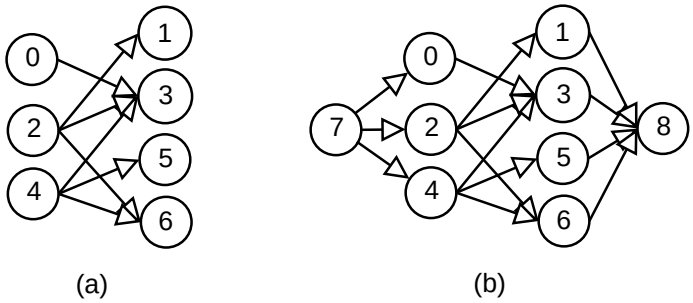
Poprawnym rozwiązaniem jest:

```
5
```

	x		x	
x				x
		S		
x				x
	x		x	



Rysunek 2.17: Kolejne fazy wyznaczania maksymalnego skojarzenia w grafie dwudzielnym. **(a)** graf wejściowy. **(b)** wyznaczono ścieżkę naprzemienną $4 \to 3$. **(c)** wyznaczono ścieżkę naprzemienną $2 \to 6$. **(d)** Wyznaczono ścieżkę naprzemienną $0 \to 3 \to 4 \to 6 \to 2 \to 1$. Krawędzie $(2, 6)$ i $(3, 4)$ usunięte zostały ze skojarzenia, a $(0, 3)$, $(2, 1)$ i $(4, 6)$ zostały dodane



Rysunek 2.18: **(a)** Graf, dla którego wyznaczane jest maksymalne skojarzenie. **(b)** sprowadzenie problemu maksymalnego skojarzenia w grafie dwudzielnym do wyznaczania maksymalnego, jednostkowego przepływu.

2.13.4. Najdroższe skojarzenie w grafie dwudzielnym

Literatura
[CON] - 5.8

W rozdziale tym przedstawiona jest funkcja `VI Hungarian(int **w, int n)`, realizująca metodę węgierską wyznaczania najdroższego maksymalnego skojarzenia w grafach dwudzielnych o równolicznych zbiorach wierzchołków V_1 oraz V_2. Działanie algorytmu zasadniczo polega na sprowadzeniu problemu wyznaczania maksymalnego najdroższego skojarzenia w grafie do zagadnienia programowania liniowego, które zostało omówione w rozdziale dotyczącym algebry liniowej. Ze względu na złożoność algorytmu, nie przedstawimy analizy jego działania, a jedynie implementację. Zainteresowanych zachęcamy do zapoznania się z literaturą.

Funkcja ta różni się istotnie od dotychczasowo omawianych. Jako parametr, przyjmuje ona bowiem macierz sąsiedztwa grafu dwudzielnego oraz jej rozmiar. Kolumny takiej macierzy reprezentują wierzchołki grafu należące do zbioru V_1, a wiersze — wierzchołki ze zbioru V_2. Wartość pola w i-tej kolumnie i j-tym wierszu reprezentuje wagę krawędzi między i-tym wierzchołkiem ze zbioru V_1 i j-tym wierzchołkiem ze zbioru V_2. Algorytm zakłada, że graf, dla którego wyznaczane jest skojarzenie, jest pełny. Takie założenie łatwo jest spełnić — wystarczy dla każdej nieistniejącej krawędzi w grafie wstawić do macierzy odpowiadającą jej wagę równą `-INF`. Na rysunku 2.19 przedstawiony jest przykładowy graf wraz z odpowiadającą mu zmodyfikowaną macierzą sąsiedztwa.

Przy użyciu metody węgierskiej można wyznaczać nie tylko najdroższe skojarzenie w grafie, ale również i najtańsze — jedyne, co należy zrobić, to zmienić na przeciwne wagi wszystkich krawędzi w grafie.

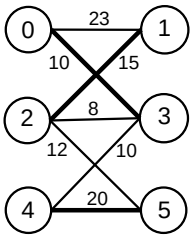
Funkcja `VI Hungarian(int **w, int n)` zwraca wektor liczb reprezentujący skojarzenie wierzchołków ze zbioru V_1 i V_2 (k-ty element tego wektora jest numerem wierzchołka ze zbioru V_2, skojarzonego z k-tym wierzchołkiem ze zbioru V_1). Implementacja tej funkcji przedstawiona jest na listingu 2.59. Jej złożoność czasowa to $O(n^3)$.

Listing 2.59: Implementacja funkcji `VI Hungarian()`

```
01 VI Hungarian(int **w, int n) {
02    int lx[n], ly[n], skojx[n], skojy[n];
03    int markx[n], marky[n], sl[n], par[n], q[n];
04    REP(i, n) {
05       skojx[i] = skojy[i] = -1;
06       ly[i] = 0;
07       lx[i] = *max_element(w[i], w[i] + n);
```

Listing 2.59: (c.d. listingu z poprzedniej strony)

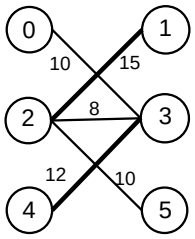
```
08    }
09    REP(k, n) {
10       int v = -1, qb = 0, qe = 0;
11       REP(i, n) {
12          marky[i] = markx[i] = 0;
13          sl[i] = -1;
14          if (skojx[i] == -1) q[qe++] = i;
15       }
16       while (v == -1) {
17          while (qb < qe) {
18             int i = q[qb++];
19             markx[i] = 1;
20             REP(j, n)
21             if (!marky[j] && (sl[j] == -1 || sl[j] > lx[i] + ly[j] - w[i][j])) {
22                if ((sl[j] = lx[par[j] = i] + ly[j] - w[i][j]) == 0) {
23                   marky[j] = 1;
24                   if (skojy[j] != -1) q[qe++] = skojy[j];
25                   else {
26                      v = j;
27                      goto end;
28                   }
29                }
30             }
31          }
32          int x = -1;
33          REP(i, n) if (!marky[i] && (x == -1 || sl[i] < x)) x = sl[i];
34          REP(i, n) {
35             if (markx[i]) lx[i] -= x;
36             if (marky[i]) ly[i] += x;
37             else if ((sl[i] -= x) == 0) {
38                marky[i] = 1;
39                if (skojy[i] != -1) q[qe++] = skojy[i];
40                else v = i;
41             }
42          }
43       }
44       end:
45       while (v != -1) {
46          int y = skojx[par[v]];
47          skojx[par[v]] = v;
48          skojy[v] = par[v];
49          v = y;
50       }
51    }
52    return VI(skojx, skojx + n);
53 }
```



(a)

	0	2	4
1	23	15	-INF
3	10	8	10
5	-INF	12	20

(b)



(c)

Rysunek 2.19: **(a)** Graf dwudzielny z wagami na krawędziach. **(b)** Zmodyfikowana macierz sąsiedztwa, wyznaczona dla grafu z rysunku **(a)**. **(c)** Graf nie zawierający doskonałego skojarzenia

W wielu zadaniach dane wejściowe są tak przedstawione, że w bardzo prosty sposób można skonstruować macierz przekazywaną do funkcji `VI Graph<V,E>::Hungarian(int **w, int n)`. Nie zawsze jednak okazuje się to prostym zadaniem. Sytuacja taka występuje przede wszystkim w zadaniach, w których najpierw należy dokonać wstępnego przetworzenia danych, a dopiero potem można przystąpić do wyznaczania najdroższego maksymalnego skojarzenia. W takich przypadkach pomocna okazać się może funkcja `VI Graph<V,E> ::HungarianG()`, która wywołuje `VI Graph<V,E>::Hungarian(int **w, int n)` dla początkowego grafu. W kolejnym kroku następuje konwersja rezultatu do postaci zgodnej z formatem wyniku wyznaczanym przez algorytm Hopcrofta. Działanie funkcji `VI Graph<V,E>:: HungarianG()` składa się z wyznaczenia podziału zbioru wierzchołków grafu na zbiory V_1 oraz V_2, konstrukcji zmodyfikowanej macierzy sąsiedztwa, wywołania właściwej funkcji wyznaczającej maksymalne skojarzenie, oraz konwersji wyznaczonego wyniku. Funkcja ta dodatkowo usuwa słabość metody węgierskiej — wymuszenia równoliczności zbiorów V_1 oraz V_2. Jeśli w grafie zbiory te nie są równe, to tworzone są specjalne wierzchołki-atrapy.

Listing 2.60: Implementacja funkcji `VI Graph<V,E>::HungarianG()`

```
01 VI HungarianG() {
02   vector<char> l;
03   VI re(SIZE(g), -1);
// Jeśli graf nie jest dwudzielny, to zwróć puste skojarzenie
04   if (!BiPart(l)) return re;
05   int gr[SIZE(g)], rel[2][SIZE(g)], n = 0, m = 0;
// Przypisz wierzchołkom z oryginalnego grafu odpowiedniki w macierzy sąsiedztwa
06   REP(x, SIZE(g)) rel[l[x]][gr[x] = (l[x] ? n++ : m++)] = x;
// Skonstruuj macierz sąsiedztwa
07   int *w[n >? = m];
08   REP(i, n) {
09     w[i] = new int[n];
10     REP(j, n) w[i][j] = -INF;
11   }
// Dodaj do macierzy wagi krawędzi z grafu
12   REP(x, SIZE(g)) FOREACH(it, g[x])
13   w[min(gr[x], gr[it->v])][max(gr[x], gr[it->v])] =
14   max(w[min(gr[x], gr[it->v])][max(gr[x], gr[it->v])], it->c);
// Wykonaj algorytm węgierski
15   VI res = Hungarian(w, n);
// Zrekonstruuj wynik, używając wyznaczonej macierzy skojarzenia
```

Listing 2.60: (c.d. listingu z poprzedniej strony)

```
16    REP(x, SIZE(res)) if (w[x][res[x]] != -INF) {
17       re[rel[0][x]] = rel[1][res[x]];
18       re[rel[1][res[x]]] = rel[0][x];
19    }
20    REP(i, n) delete[]w[i];
21    return re;
22 }
```

Listing 2.61: Wynik wygenerowany przez funkcję `VI Graph<V,E>::HungarianG()` dla grafu z rysunku 2.19.c

```
Wierzcholek 1 skojarzono z 2
Wierzcholek 3 skojarzono z 4
```

Listing 2.62: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 2.61. Pełny kod źródłowy programu znajduje się w pliku `hungarian.cpp`

```
// Wzbogacenie wierzchołków oraz krawędzi wymagane przez algorytm wyznaczania
// najcięższego maksymalnego skojarzenia
01 struct Vs {
02    int t, s;
03 };
04 struct Ve {
05    int c, rev;
06 };
07 int main() {
08    int n, m, s, b, e;
// Konstruowanie grafu zgodnie z zadanym opisem
09    cin >> n >> m;
10    Graph<Vs, Ve> g(n);
11    Ve l;
12    REP(x, m) {
13       cin >> b >> e >> l.c;
14       g.EdgeU(b, e, l);
15    }
// Wyznaczenie maksymalnego skojarzenia przy użyciu metody węgierskiej, oraz
// wypisanie wyniku
16    VI res = g.HungarianG();
17    REP(x, SIZE(res)) if (res[x] > x)
18       cout << "Wierzcholek " << x << " skojarzono z " << res[x] << endl;
19    return 0;
20 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 259	acm.uva.es - zadanie 10746	acm.uva.es - zadanie 10615
acm.uva.es - zadanie 10080	acm.uva.es - zadanie 10804	spoj.sphere.pl - zadanie 377
acm.uva.es - zadanie 10349	spoj.sphere.pl - zadanie 203	spoj.sphere.pl - zadanie 412
acm.uva.es - zadanie 10888	spoj.sphere.pl - zadanie 660	acm.sgu.ru - zadanie 218
spoj.sphere.pl - zadanie 286	acm.sgu.ru - zadanie 210	acm.sgu.ru - zadanie 234
spoj.sphere.pl - zadanie 373	acm.sgu.ru - zadanie 242	
acm.sgu.ru - zadanie 190		

Rozdział 3

Geometria obliczeniowa na płaszczyźnie

Literatura
[WDA] - 35
[ASD] - 8

W tym rozdziale przedstawione są najczęściej wykorzystywane w zadaniach algorytmy związane z geometrią obliczeniową. Poruszone zostaną między innymi takie zagadnienia, jak wyznaczanie punktów przecięcia figur, liczenie powierzchni wielokąta, wyznaczanie wypukłej otoczki, czy sortowanie kątowe zbioru punktów.

Podobnie jak w przypadku grafów, algorytmy geometryczne również wykorzystują wspólną strukturę do reprezentacji obiektów geometrycznych — elementarną strukturą jest oczywiście punkt. Wszystkie inne obiekty geometryczne są reprezentowane przy jego użyciu:

- odcinek — para punktów stanowiących odpowiednio początek i koniec odcinka,
- prosta — para różnych punktów należących do prostej,
- okrąg i koło — punkt stanowiący środek okręgu/koła oraz liczba określająca jego promień,
- wielokąt — wektor punktów będących wierzchołkami wielokąta w kolejności występowania na obwodzie.

Na wstępie musimy wprowadzić pewną konwencję oznaczeniową dla poszczególnych obiektów. Punkty będziemy oznaczać wielkimi literami alfabetu angielskiego: $A, B, \ldots$. Współrzędne punktu będą przedstawiane w postaci (x, y), gdzie x jest odciętą, natomiast y — rzędną punktu. Inne obiekty geometryczne oznaczać będziemy małymi literami alfabetu — z kontekstu będzie zawsze wynikało, o jakiego rodzaju obiekt chodzi. Definiując odcinek wyznaczony przez dwa punkty A i B będziemy pisali $A \mapsto B$. Prostą przechodzącą przez dwa różne punkty A i B reprezentujemy jako $A \rightarrow B$. Zarówno w przypadku prostej, jak i odcinka istotna jest kolejność podawania punktów je wyznaczających. W wielu miejscach powstaje potrzeba określania pozycji punktu względem prostej — może on leżeć na prostej lub po jednej z jej stron. Przez stronę lewą będziemy rozumieli tę półpłaszczyznę, która zawiera punkty leżące po lewej stronie wektora $A \mapsto B$.

Wielokąty reprezentowane będą jako listy wierzchołków podawanych w kolejności występowania na obwodzie, natomiast okręgi oraz koła — jako pary (środek, promień). Zadania geometryczne, oprócz problemów algorytmicznych związanych z ich rozwiązaniem, kryją w sobie dodatkową trudność — błędy zaokrągleń. Choćby podczas wyznaczania punktów przecięcia dwóch prostych, nie jesteśmy w stanie w ogólnym przypadku podać dokładnych

współrzędnych wyznaczanego punktu przecięcia przy użyciu reprezentacji liczb używanych przez komputery. Podczas rozwiązywania zadań, kolejno pojawiające się błędy zaokrągleń, w przypadku źle zaimplementowanego programu, mogą kumulować się i w konsekwencji silnie zaburzać wynik.

W przypadku wielu konkursowych zadań, na szczęście istnieje możliwość takiego rozwiązania, że wszystkie wykonywane obliczenia są całkowitoliczbowe (jedną ze stosowanych metod jest odpowiednie przeskalowywanie współrzędnych). W zależności od rodzaju wykonywanych obliczeń, struktura punktu do reprezentacji swoich współrzędnych powinna używać typu **int** lub **long long** — w przypadku obliczeń całkowitoliczbowych, lub **float**, **double**, a niekiedy nawet (w zadaniach wymagających wysokiej dokładności) **long double** — przy obliczeniach niecałkowitoliczbowych.

W oryginalnej biblioteczce algorytmicznej znajdują się dwa typy — `POINTT` oraz `POINTR`, które są wykorzystywane odpowiednio do reprezentacji współrzędnych punków oraz do przechowywania wyników działania niektórych algorytmów (takich jak pole wielokąta). W przypadku zadań, operujących na zmiennych całkowitoliczbowych, definicje `POINTT` oraz `POINTR` dobierane są następująco:

- **typedef** `POINTT` **int**;
- **typedef** `POINTR LL`;

w przypadku zadań wymagających obliczeń niecałkowitoliczbowych, definicje te zostają odpowiednio zmodyfikowane, przykładowo może to być:

- **typedef** `POINTT` **float**;
- **typedef** `POINTR` **long double**;

Ze względu na edukacyjny kształt książki, konwencja ta została zmieniona (co niestety wpłynęło negatywnie na długość implementacji algorytmów). Wprowadzone zostały dwa niezależne typy — `POINT` oraz `POINTD`. Pierwszy z nich służy do reprezentowania punktów o współrzędnych całkowitoliczbowych, drugi natomiast — zmiennoprzecinkowych. Implementacja struktury `POINT` przedstawiona jest na listingu 3.1, natomiast listing 3.2 prezentuje implementację struktury `POINTD`. Implementacja z listingu 3.2 wykorzystuje funkcję **bool** `IsZero(`**double**`)` znajdującą się na listingu 3.4.

Listing 3.1: Implementacja struktury `POINT`

```
// Struktura reprezentująca punkt o współrzędnych całkowitoliczbowych
01 struct POINT {
02    int x, y;
// Konstruktor punktu pozwalający na skrócenie zapisu wielu funkcji
// wykorzystujących punkty - w szczególności operacje wstawiania punktów do
// struktur danych
03    POINT(int x = 0, int y = 0) : x(x), y(y) { }
// Operator sprawdzający, czy dwa punkty są sobie równe.
04    bool operator ==(POINT & a) {
05       return a.x == x && a.y == y;
06    }
07 };
```

Listing 3.1: (c.d. listingu z poprzedniej strony)

```
// Operator używany przez przykładowe programy do wypisywania struktury punktu
08 ostream & operator<<(ostream & a, POINT & p) {
09    a << "(" << p.x << ", " << p.y << ")";
10    return a;
11 }
```

Listing 3.2: Implementacja struktury POINTD

```
// Struktura reprezentująca punkt o współrzędnych rzeczywistych
01 struct POINTD {
02    double x, y;
// Konstruktor punktu
03    POINTD(double wx = 0, double wy = 0) : x(wx), y(wy) { }
// Konstruktor POINTD z typu POINT - jest on potrzebny w celu wykonywania
// automatycznej konwersji między POINT a POINTD.
04    POINTD(const POINT & p) : x(p.x), y(p.y) { }
// Operator sprawdzający, czy dwa punkty są sobie równe. Ze względu na
// reprezentowanie współrzędnych punktów przy użyciu zmiennych double,
// operator porównuje współrzędne punktów z pewną tolerancją
05    bool operator ==(POINTD & a) {
06       return IsZero(a.x - x) && IsZero(a.y - y);
07    }
08 };
// Operator używany przez przykładowe programy do wypisywania struktury punktu
09 ostream & operator<<(ostream & a, POINTD & p) {
10    a << "(" << p.x << ", " << p.y << ")";
11    return a;
12 }
```

W zadaniach geometrycznych często pojawiają się problemy związane z zaokrąglaniem liczb. W przypadku porównywania wartości dwóch liczb typu **double**, które teoretycznie powinny być sobie równe, może okazać się, że ze względu na powstałe błędy zaokrągleń komputer uznaje je za różne. Tego typu problemy powodują, że podczas porównywania wartości liczb zmiennoprzecinkowych, należy dokonywać tego z dodatkową tolerancją. W tym celu wprowadzone zostały definicje stałej EPS oraz funkcji **bool** IsZero(**double**) — przedstawione na listingach 3.3 i 3.4.

Listing 3.3: Stała **const double** EPS

```
// Stała EPS jest używana w wielu algorytmach geometrycznych do porównywania
// wartości bliskich zera
1 const double EPS = 10e-9;
```

Listing 3.4: Implementacja funkcji **inline bool** IsZero(**double**)

```
// Funkcja sprawdza, czy podana liczba jest dostatecznie bliska 0
1 inline bool IsZero(double x) {
2    return x >= -EPS && x <= EPS;
```

Listing 3.4: (c.d. listingu z poprzedniej strony)

```
3 }
```

Algorytmy geometryczne operujące na zbiorach punktów, bardzo często muszą sprawdzać położenie poszczególnych punktów względem siebie. Ogólne pytanie, przy użyciu którego można odpowiedzieć na wiele innych, brzmi: po której stronie prostej $B \rightarrow C$ leży punkt A? Potrafiąc odpowiadać na to pytanie, możemy również odpowiedzieć na pytanie typu: „czy dwa odcinki $A \mapsto B$ i $C \mapsto D$ przecinają się". Aby to stwierdzić, wystarczy sprawdzić, czy punkty A i B leżą po przeciwnych stronach prostej $C \rightarrow D$, oraz czy punkty C i D leżą po przeciwnych stronach prostej $A \rightarrow B$. W celu udzielenia odpowiedzi na pierwotne pytanie — po której stronie prostej $B \rightarrow C$ leży punkt A — można skorzystać z własności iloczynu wektorowego. Jak wiadomo, iloczyn wektorowy $A \times B$ ma wartość 0, jeśli wektory A i B są do siebie równoległe, jego wartość jest większa od 0, jeśli kąt skierowany α między wektorami A i B ma miarę w przedziale $(0, 180)$ stopni, a mniejszy od zera, jeśli α jest większy od 180 stopni. Makro `Det` przedstawione na listingu 3.5 służyć będzie do wyznaczania wartości iloczynu wektorowego $A \mapsto B$ i $A \mapsto C$. Współrzędne punktów wyznaczających wektory $A \mapsto B$ i $A \mapsto C$ muszą być całkowitoliczbowe.

Listing 3.5: Implementacja makra `Det`

```
// Makro wyznacza wartość iloczynu wektorowego (a -> b)*(a -> c)
1 #define Det(a,b,c) (LL(b.x-a.x)*LL(c.y-a.y)-LL(b.y-a.y)*(c.x-a.x))
```

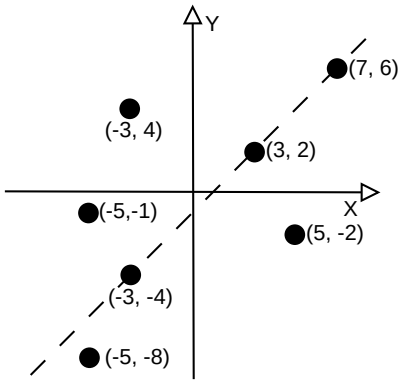
Oprócz wyznaczania wartości iloczynu wektorowego, częstą operacją podczas analizowania zbioru punktów jest ich sortowanie. Możliwe są różne sposoby określania porządku liniowego na zbiorze punktów — jednym z nich może być choćby omawiany w jednym z kolejnych podrozdziałów porządek kątowy. Najczęściej stosowanymi są jednak porządki „po współrzędnych" — (x, y), w którym punkty są sortowane w kolejności niemalejących współrzędnych x, a w przypadku takiej samej współrzędnej x — w kolejności niemalejących współrzędnych y oraz analogiczny porządek (y, x). Operatorami wykorzystywanymi przez funkcję `sort` z biblioteki STL do określenia tych dwóch porządków są **bool** `OrdXY(POINT*, POINT*)` oraz **bool** `OrdYX(POINT*, POINT*)`. Ich implementacja przedstawiona jest na listingach 3.6 i 3.7.

Listing 3.6: Implementacja operatora **bool** `OrdXY(POINT*, POINT*)`

```
// Operator określający liniowy porządek na zbiorze punktów po współrzędnych
// (x, y)
1 bool OrdXY(POINT * a, POINT * b) {
2    return a->x == b->x ? a->y < b->y : a->x < b->x;
3 }
```

Listing 3.7: Implementacja operatora **bool** `OrdYX(POINT*, POINT*)`

```
// Operator określający liniowy porządek na zbiorze punktów po współrzędnych
// (y, x)
1 bool OrdYX(POINT * a, POINT * b) {
2    return a->y == b->y ? a->x < b->x : a->y < b->y;
3 }
```



Rysunek 3.1: Zbiór siedmiu punktów na płaszczyźnie oraz prosta $A \rightarrow B$ o równaniu $y = x - 1$ wyznaczona przez punkty $A = (-3, -4)$ i $B = (7, 6)$. Punkty $(-3, 4)$ oraz $(-5, -1)$ leżą po lewej stronie prostej $A \rightarrow B$, natomiast punkty $(5, -2)$ oraz $(-5, -8)$ — po prawej, a punkt $(3, 2)$ na prostej

3.1. Odległość punktu od prostej

Często wykonywaną operacją w geometrii obliczeniowej, jest liczenie odległości punktu od prostej. Rozpatrując prostą a o równaniu $ax + by + c = 0$ oraz punkt E o współrzędnych (x_0, y_0), odległość d punktu E od prostej a można wyznaczyć na podstawie dobrze znanego wzoru:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Stosując ten wzór, funkcja **double** `PointLineDist(POINTD, POINTD, POINTD)`, której implementacja przedstawiona jest na listingu 3.8, wyznacza odległość punktu określonego przez trzeci parametr funkcji, od prostej wyznaczonej przez punkty podane jako dwa pierwsze parametry.

Listing 3.8: Implementacja funkcji **double** `PointLineDist(POINTD, POINTD, POINTD)`

```
// Funkcja wyznacza odległość punktu p od prostej (p1 -> p2)
1 double PointLineDist(POINTD p1, POINTD p2, POINTD p) {
2     double A = p2.y - p1.y, B = p2.x - p1.x;
3     return abs(A * (p1.x - p.x) + B * (p.y - p1.y)) / sqrt(A * A + B * B);
4 }
```

Listing 3.9: Odległości poszczególnych punktów z rysunku 3.1 od prostej $y = x - 1$, wyznaczone przy użyciu funkcji **double** `PointLineDist(POINTD, POINTD, POINTD)`

```
Odleglosc punktu (-5, -8) od prostej ((-3, -4),(7, 6)) wynosi 1.41421
Odleglosc punktu (5, -2) od prostej ((-3, -4),(7, 6)) wynosi 4.24264
Odleglosc punktu (3, 2) od prostej ((-3, -4),(7, 6)) wynosi 0
Odleglosc punktu (-3, 4) od prostej ((-3, -4),(7, 6)) wynosi 5.65685
Odleglosc punktu (-5, -1) od prostej ((-3, -4),(7, 6)) wynosi 3.53553
```

Listing 3.10: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.9. Pełny kod źródłowy programu znajduje się w pliku `pointdist.cpp`

```
1 int main() {
2   POINT l1, l2, p;
// Wczytaj pozycje punktów wyznaczających prostą
3   cin >> l1.x >> l1.y >> l2.x >> l2.y;
// Dla wszystkich punktów wyznacz odległość od prostej
4   while(cin >> p.x >> p.y)
5     cout << "Odleglosc punktu " << p << " od prostej (" <<
6     l1 << "," << l2 << ") wynosi " << PointLineDist(l1, l2, p) << endl;
7   return 0;
8 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10263	acm.uva.es - zadanie 10709	acm.uva.es - zadanie 10762
acm.uva.es - zadanie 10310	acm.uva.es - zadanie 10011	

3.2. Pole wielokąta

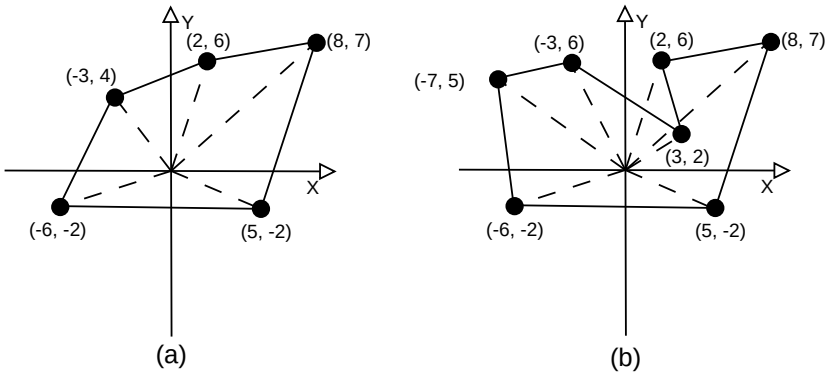
Wyznaczanie pola wielokątów wypukłych w porównaniu z dowolnymi wielokątami wydaje się istotnie prostsze. Jak przedstawiono na rysunku 3.2.a, wielokąt wypukły można podzielić na trójkąty, których sumaryczne pole jest równe polu całego wielokąta. W celu wyznaczenia pola trójkąta można skorzystać z iloczynu wektorowego — pole trójkąta wyznaczonego przez trzy punkty A, B i C jest równe $\frac{1}{2}|(A \mapsto B) \times (A \mapsto C)|$. Nie ograniczając się wyłącznie do wielokątów wypukłych, okazuje się, że suma pól trójkątów nie daje pola badanego wielokąta, zatem bezpośrednia aplikacja przedstawionej metody jest niemożliwa.

Przyglądając się rysunkowi 3.2.b widać, że obwód niektórych trójkątów jest zorientowany zgodnie z ruchem wskazówek zegara, natomiast innych — przeciwnie. Wyznaczając pole trójkąta przy użyciu iloczynu wektorowego, ale pomijając we wzorze moduł ($\frac{1}{2}(A \mapsto B) \times (A \mapsto C)$), otrzymujemy oznaczone pole trójkąta, którego wartość jest dodatnia bądź ujemna w zależności od orientacji obwodu trójkąta. Okazuje się, że suma oznaczonych pól wszystkich trójkątów jest równa oznaczonemu polu wielokąta. Własność ta wynika z faktu, iż każdy punkt nienależący do wnętrza wielokąta, należy do takiej samej liczby trójkątów o obwodach zorientowanych zgodnie z ruchem wskazówek zegara, co i przeciwnie — dzięki temu, pola wszystkich liczonych wielokrotnie obszarów znoszą się.

Postać wzoru na pole dowolnego wielokąta wyznaczonego przez punkty (X_1, Y_1), (X_2, Y_2), …, (X_n, Y_n) jest następująca:

$$\frac{1}{2}|X_1Y_2 - X_2Y_1 + X_2Y_3 - X_3Y_2 + \ldots + X_{n-1}Y_n - X_nY_{n-1} + X_nY_1 - X_1Y_n|$$

Przedstawiona na listingu 3.11 funkcja **double** `PolygonArea(vector<POINT>&)` przyjmuje jako parametr listę punktów wyznaczających wielokąt, a zwraca jego pole.



Rysunek 3.2: **(a)** Wielokąt wypukły wyznaczony przez zbiór pięciu punktów. Przerywane odcinki reprezentują podział tego wielokąta na rozłączne trójkąty. **(b)** Wielokąt wyznaczony przez siedem punktów.

Listing 3.11: Implementacja funkcji **double** `PolygonArea(vector<POINT>&)`

```
// Funkcja liczy pole wielokąta, którego wierzchołki wyznaczone są przez wektor
// p
1 double PolygonArea(vector<POINT> &p) {
2    double area = 0;
3    int s = SIZE(p);
4    REP(x, s) area += (p[x].x + p[(x + 1) % s].x) * (p[x].y - p[(x + 1) % s].y);
5    return abs(area) / 2;
6 }
```

Listing 3.12: Pole wielokąta z rysunku 3.2.b wyznaczone przez funkcję **double** `PolygonArea(vector<POINT>&)`

```
Pole wielokata: 94
```

Listing 3.13: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.12. Pełny kod źródłowy programu znajduje się w pliku `polygonarea.cpp`

```
01 int main() {
02    int n;
03    vector<POINT> l;
04    POINT p;
// Wczytaj liczbę wierzchołków wielokąta
05    cin >> n;
// Dodaj wszsytkie wierzchołki wielokąta do wektora punktów
06    REP(x, n) {
07       cin >> p.x >> p.y;
08       l.PB(p);
09    }
// Wyznacz pole wielikąta
10    cout << "Pole wielokata: " << PolygonArea(l) << endl;
11    return 0;
12 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10088 spoj.sphere.pl - zadanie 55	acm.uva.es - zadanie 10002 acm.sgu.ru - zadanie 209	acm.uva.es - zadanie 10456 acm.uva.es - zadanie 10907

3.3. Przynależność punktu do figury

Literatura
[WDA] - 35.1
[ASD] - 8.2

Rozwiązanie wielu zadań wymaga umiejętności sprawdzania, czy pewne punkty należą do różnych obiektów geometrycznych, takich jak odcinki, proste, wielokąty czy koła. W aktualnym rozdziale przedstawione zostaną metody umożliwiające dokonywanie takich testów. Analizę problemu rozpoczniemy od bardzo prostego przypadku sprawdzania przynależności punktu do prostokąta o bokach równoległych do osi układu współrzędnych.

W celu sprawdzenia czy punkt $A = (x, y)$ znajduje się we wnętrzu prostokąta wyznaczonego przez dwa jego przeciwległe wierzchołki $B = (x_1, y_1)$ oraz $C = (x_2, y_2)$, wystarczy sprawdzić, czy odcięta punktu leży pomiędzy odciętymi wierzchołków prostokąta oraz czy podobna zależność jest zachowana przez rzędne. Nierówności, które muszą zostać spełnione są następujące:

$$min(x_1, x_2) < x < max(x_1, x_2),\ min(y_1, y_2) < y < max(y_1, y_2)$$

W przypadku sprawdzania, czy punkt należy do wnętrza prostokąta lub leży na jego obwodzie, należy zamienić ostre nierówności z poprzednich warunków na nieostre:

$$min(x_1, x_2) \leqslant x \leqslant max(x_1, x_2),\ min(y_1, y_2) \leqslant y \leqslant max(y_1, y_2)$$

Do weryfikowania tych dwóch rodzajów przynależności dostępne są makra `PointInRect` oraz `PointInsideRect`. Oba makra przyjmują jako parametry dwa punkty wyznaczające prostokąt oraz punkt, dla którego badana jest przynależność. W przypadku, gdy punkt należy do prostokąta, makra przyjmują wartość prawda. Makra te zostały przedstawione na listingach 3.14 oraz 3.15.

Listing 3.14: Implementacja makra `PointInsideRect`

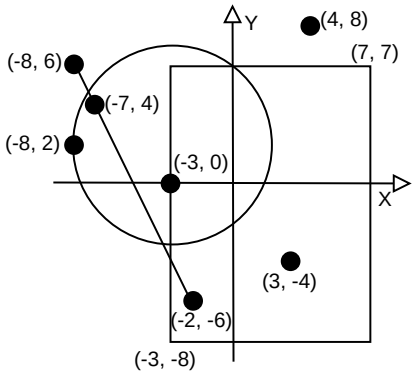
```
1 #define PointInsideRect(p1,p2,p3) (min(p1.x, p2.x) < p3.x && \
2 min(p1.y, p2.y) < p3.y && max(p1.x, p2.x) > p3.x && max(p1.y, p2.y) > p3.y)
```

Listing 3.15: Implementacja makra `PointInRect`

```
1 #define PointInRect(p1,p2,p3) (min(p1.x, p2.x) <= p3.x && \
2 min(p1.y, p2.y) <= p3.y && max(p1.x, p2.x) >= p3.x && max(p1.y, p2.y) >= p3.y)
```

Analogicznymi makrami są `PointInsideSegment` oraz `PointInSegment`. Ich zadaniem jest sprawdzanie, czy punkt podany jako trzeci parametr leży wewnątrz bądź na odcinku wyznaczonym przez punkty stanowiące pierwszy oraz drugi parametr.

Implementacja tych dwóch makr wykorzystuje zarówno poprzednie makra do sprawdzania przynależności punktu do prostokąta, jak również iloczyn wektorowy. Najpierw jest



Rysunek 3.3: Przykładowy zbiór punktów prezentujący różne przypadki przynależności punktów do obiektów geometrycznych — odcinka, prostokąta i koła

sprawdzane, czy testowany punkt leży na prostej wyznaczonej przez odcinek. Jeśli tak, to musi on jeszcze należeć do prostokąta wyznaczonego przez końce odcinka. Implementacja tych makr przedstawiona jest na listingach 3.16 i 3.17.

Listing 3.16: Implementacja makra `PointInsideSegment`

```
1 #define PointInsideSegment(p1,p2,l) (Det(p1,p2,l)==0 && PointInsideRect(p1,p2,l))
```

Listing 3.17: Implementacja makra `PointInSegment`

```
1 #define PointInSegment(p1,p2,l) (Det(p1,p2,l)==0 && PointInRect(p1,p2,l))
```

Ostatnimi z serii prostych i krótkich makr, są `PointInsideCircle` oraz `PointInCircle`. Służą one odpowiednio do sprawdzania, czy punkt określony przez trzeci parametr makra należy do wnętrza bądź wnętrza lub obwodu koła o środku w punkcie określonym przez pierwszy parametr makra oraz promieniu zdefiniowanym przez drugi parametr. Implementacje tych makr zostały umieszczone odpowiednio na listingach 3.18 i 3.19.

Listing 3.18: Implementacja makra `PointInsideCircle`

```
1 #define PointInsideCircle(c,r,p) (sqr(c.x-p.x)+sqr(c.y-p.y)+EPS < sqr(r))
```

Listing 3.19: Implementacja makra `PointInCircle`

```
1 #define PointInCircle(c,r,p) (sqr(c.x-p.x)+sqr(c.y-p.y)-EPS < sqr(r))
```

Listing 3.20: Wynik działania makr służących do sprawdzania przynależności punktów do różnych obiektów geometrycznych na przykładzie odcinka o końcach w punktach $(-2,-6)$ i $(-8,6)$, okręgu o środku w punkcie $(-3,2)$ i promieniu 5 oraz prostokąta wyznaczonego przez punkty $(-3,-8)$ i $(7,7)$ (patrz rysunek 3.3).

	Rectangle		Segment		Circle	
	Inside	In	Inside	In	Inside	In
(-8, 2)	0	0	0	0	0	1
(-7, 4)	0	0	1	1	1	1

Listing 3.20: (c.d. listingu z poprzedniej strony)

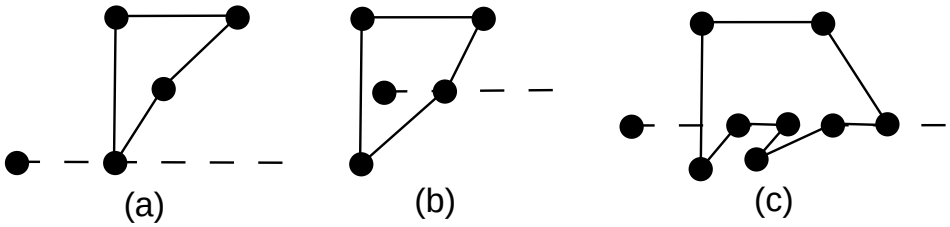
(-8, 6)	0	0	0	1	0	0
(-3, 0)	0	1	0	0	1	1
(3, -4)	1	1	0	0	0	0
(4, 8)	0	0	0	0	0	0

Listing 3.21: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.20. Pełny kod źródłowy programu znajduje się w pliku `pointin_inside.cpp`

```
01 int main() {
02    POINT r1,r2,c,s1,s2,p;
03    int r;
// Wczytaj współrzędne wierzchołków wyznaczających prostokąt
04    cin >> r1.x >> r1.y >> r2.x >> r2.y;
// Wczytaj współrzędne punktów wyznaczających odcinek
05    cin >> s1.x >> s1.y >> s2.x >> s2.y;
// Wczytaj środek oraz promień okręgu
06    cin >> c.x >> c.y >> r;
07    cout << "\t\t\t Rectangle\t\t Segment\t\t Circle" << endl;
08    cout << "\t\t\tInside\t In\t Inside\t In\t Inside\t In" << endl;
// Dla wszystkich punktów wyznacz wynik działania poszczególnych makr
09    while(cin >> p.x >> p.y) {
10       cout << p << "\t\t" << PointInsideRect(r1, r2, p) << "\t\t" <<
11          PointInRect(r1, r2, p) << "\t\t" <<
12          PointInsideSegment(s1, s2, p) << "\t\t" <<
13          PointInSegment(s1, s2, p) << "\t\t" <<
14          PointInsideCircle(c, r, p) << "\t\t" <<
15          PointInCircle(c, r, p) << endl;
16    }
17    return 0;
18 }
```

Bardziej interesującym zagadnieniem jest sprawdzanie przynależności punktu do wielokąta. Zajmiemy się tym problemem przy założeniu, że wszystkie punkty stanowiące wierzchołki wielokąta mają całkowitoliczbowe współrzędne. W celu rozwiązania tego problemu, istotną obserwacją jest to, że dowolna półprosta o początku w punkcie należącym do wnętrza wielokąta przecina jego obwód nieparzystą liczbę razy. Sytuacja odwrotna występuje w przypadku punktów nienależących do wielokąta — wtedy liczba przecięć jest parzysta. Zatem w celu sprawdzenia przynależności punktu do wielokąta można sprawdzić, z iloma bokami wielokąta przecina się dowolna półprosta o początku w tym punkcie. Wybór jakiejkolwiek półprostej niesie ze sobą jednak wiele problemów. Jak przedstawiono na rysunku 3.4, prosta taka może przecinać obwód wielokąta na różne sposoby, co wiąże się z dodatkowymi przypadkami do rozpatrzenia, znacznie komplikującymi implementację. W przypadku przedstawionym na rysunku 3.4.a, punkt przecięcia półprostej z obwodem wielokąta powinien być liczony podwójnie, podczas gdy sytuacja z rysunku 3.4.b — pojedynczo.

Widać, że odpowiedni wybór półprostej jest bardzo ważny — najlepiej byłoby wybrać ją w ten sposób, aby nie przechodziła przez żaden wierzchołek wielokąta. Okazuje się, że



Rysunek 3.4: Różne przypadki przecinania obwodu wielokąta przez półprostą.

wyboru takiego można dokonać w bardzo prosty sposób — dla testowanego punktu (x, y), półprosta przechodząca przez punkt $(INF, y+1)$, nie przecina żadnego wierzchołka wielokąta, co wynika bezpośrednio z faktu, iż wszystkie punkty mają całkowitoliczbowe współrzędne.

Implementacja algorytmu wykorzystującego ten fakt została zrealizowana w funkcjach **bool** `PointInsidePol(vector<POINT>&, POINT)` oraz **bool** `PointInPol(vector<POINT>&, POINT)` przedstawionych na listingach 3.23 i 3.24 — przyjmują one jako parametry odpowiednio wektor wierzchołków wielokąta oraz punkt, dla którego przeprowadzany jest test. Pierwsza z tych funkcji zwraca prawdę, jeśli punkt należy do wnętrza wielokąta, natomiast druga zwraca prawdę, jeśli punkt należy do wnętrza lub obwodu wielokąta. Czas ich działania jest liniowy — dla każdego boku wielokąta funkcje sprawdzają, czy przecina on półprostą o początku w testowanym punkcie. Obie przedstawione poniżej funkcje wykorzystują funkcję **bool** `SegmentCross(POINT&, POINT&, POINT&, POINT&)`, która przyjmuje jako parametry dwa odcinki reprezentowane przez pary wierzchołków oraz zwara prawdę, jeśli odcinki te przecinają się. Jej implementacja została przedstawiona na listingu 3.22.

Listing 3.22: Implementacja funkcji **bool** `SegmentCross(POINT&, POINT&, POINT&, POINT&)`

```
1 inline bool SegmentCross(POINT & p1, POINT & p2, POINT & l1, POINT & l2) {
2    return sgn(Det(p1, p2, l1)) * sgn(Det(p1, p2, l2)) == -1
3    && sgn(Det(l1, l2, p1)) * sgn(Det(l1, l2, p2)) == -1;
4 }
```

Listing 3.23: Implementacja funkcji **bool** `PointInsidePol(vector<POINT>&, POINT)`

```
1 bool PointInsidePol(vector<POINT> &l, POINT p) {
2    int v = 0, s = SIZE(l);
3    POINT d(INF, p.y + 1);
// Jeśli punkt leży na jednym z boków wielokąta, to nie należy do wnętrza
// wielokąta
4    REP(x, s) if (PointInSegment(l[x], l[(x + 1) % s], p)) return false;
// Wyznacz liczbę przecięć obwodu wielokąta z półprostą (p -> d)
5    REP(x, s) v += SegmentCross(p, d, l[x], l[(x + 1) % s]);
// Jeśli półprosta przecina obwód nieparzystą liczbę razy, to punkt należy do
// wielokąta
6    return v & 1;
7 }
```

Listing 3.24: Implementacja funkcji **bool** `PointInPol(vector<POINT>&, POINT)`

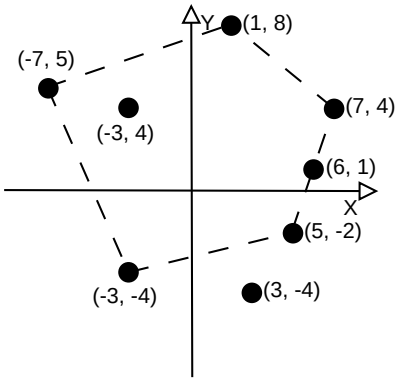
```
1 bool PointInPol(vector<POINT> &l, POINT p) {
2    int v = 0, s = SIZE(l);
3    POINT d(INF, p.y + 1);
// Jeśli punkt leży na jednym z boków wielokąta, to zwróć prawdę
4    REP(x, s) if (PointInSegment(l[x], l[(x + 1) % s], p)) return true;
// Wyznacz liczbę przecięć obwodu wielokąta z półprostą (p -> d)
5    REP(x, s) v += SegmentCross(p, d, l[x], l[(x + 1) % s]);
// Jeśli półprosta przecina obwód nieparzystą liczbę razy, to punkt należy do
// wielokąta
6    return v & 1;
7 }
```

W przypadku rozpatrywania przynależności punktu do wielokąta wypukłego, wcześniej omówione funkcje jak najbardziej działają poprawnie, jednak nie są optymalne. W takim przypadku, istnieje możliwość sprawdzenia przynależności punktu w czasie logarytmicznym ze względu na liczbę wierzchołków wielokąta.

Zasada działania takiego algorytmu opiera się na metodzie wyszukiwania binarnego — na początku algorytm wybiera sobie punkt należący do wielokąta wypukłego (może to być przykładowo jego wierzchołek). Następnie wnętrze wielokąta dzielone jest na trójkąty (tak jak przedstawiono to na rysunku 3.2.a) przy użyciu rodziny półprostych $P_1, P_2, \ldots$ Jeśli punkt należy do wielokąta, to musi znajdować się w jednym z powstałych trójkątów. Przy użyciu iloczynu wektorowego, za pomocą którego sprawdzane jest położenie punktu względem kolejno wybieranych prostych, przeprowadzane jest wyszukiwanie binarne, w celu wyznaczenia pary sąsiednich półprostych P_k i P_{k+1}, pomiędzy którymi leży testowany punkt. Wówczas jeśli znajduje się on w trójkącie wyznaczonym przez półproste P_k, P_{k+1} oraz odpowiedni bok wielokąta, to należy on również do wielokąta. Realizacja tej metody jest przedstawiona na listingach 3.25 oraz 3.26 w postaci funkcji **bool** `PointInsideConvexPol(vector <POINT>&, POINT)` oraz **bool** `PointInConvexPol(vector <POINT>&, POINT)`. Przyjmują one jako parametry listę wierzchołków wielokąta wypukłego oraz punkt, dla którego przeprowadzany jest test.

Listing 3.25: Implementacja funkcji **bool** `PointInsideConvexPol(vector<POINT>&, POINT)`

```
01 bool PointInsideConvexPol(vector<POINT> &l, POINT p) {
02    int a = 1, b = SIZE(l) - 1, c;
// Jeśli odcinek (l[0] -> l[a]) leży na prawo od odcinka (l[0] -> l[b]) to
// następuje zamiana
03    if (Det(l[0], l[a], l[b]) > 0) swap(a, b);
// Jeśli punkt p nie leży po prawej stronie prostej (l[0] -> l[a]) lub po
// lewej stronie (l[0] -> l[b]) to nie należy do wielokąta
04    if (Det(l[0], l[a], p) >= 0 || Det(l[0], l[b], p) <= 0) return false;
// Wyszukiwanie binarne wycinka płaszczyzny zawierającego punkt p
05    while (abs(a - b) > 1) {
06       c = (a + b) / 2;
07       if (Det(l[0], l[c], p) > 0) b = c;
08       else a = c;
09    }
// Jeśli punkt p leży w trójkącie (l[0],l[a],l[b]), to należy do wielokąta
```



Rysunek 3.5: Przykładowy wielokąt wypukły oraz zbiór trzech punktów $(3, -4)$, $(6, 1)$ i $(-3, 4)$, przedstawiających różne przypadki przynależności punktu do wielokąta.

Listing 3.25: (c.d. listingu z poprzedniej strony)

```
10    return Det(l[a], l[b], p) < 0;
11 }
```

Listing 3.26: Implementacja funkcji **bool** `PointInConvexPol(vector<POINT>&, POINT)`

```
01 bool PointInConvexPol(vector<POINT> &l, POINT p) {
02    int a = 1, b = SIZE(l) - 1, c;
// Jeśli odcinek (l[0] -> l[a]) leży na prawo od odcinka (l[0] -> l[b]) to
// następuje zamiana
03    if (Det(l[0], l[a], l[b]) > 0) swap(a, b);
// Jeśli punkt p leży po lewej stronie prostej (l[0] -> l[a]) lub po
// prawej stronie (l[0] -> l[b]) to nie należy do wielokąta
04    if (Det(l[0], l[a], p) > 0 || Det(l[0], l[b], p) < 0) return false;
// Wyszukiwanie binarne wycinka płaszczyzny zawierającego punkt p
05    while (abs(a - b) > 1) {
06       c = (a + b) / 2;
07       if (Det(l[0], l[c], p) > 0) b = c;
08       else a = c;
09    }
// Jeśli punkt p leży w trójkącie (l[0],l[a],l[b]), to należy do wielokąta
10    return Det(l[a], l[b], p) <= 0;
11 }
```

Listing 3.27: Przykład wykorzystania funkcji sprawdzających przynależności punktów do wielokąta na podstawie zbioru punktów z rysunku 3.5

	Polygon		Convex Polygon	
	Inside	In	Inside	In
(-3, 4)	1	1	1	1
(1, 8)	0	1	0	1
(6, 1)	0	1	0	1
(3, -4)	0	0	0	0

Listing 3.28: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.27. Pełny kod źródłowy programu znajduje się w pliku `pointincpol.cpp`

```
01 int main() {
02    vector<POINT> pol;
03    POINT p;
04    int n;
// Wczytaj liczbę wierzchołków wielokąta
05    cin >> n;
// Wczytaj kolejne wierzchołki wielokąta oraz dodaj je do wektora
06    REP(x, n) {
07       cin >> p.x >> p.y;
08       pol.PB(p);
09    }
10    cout << "\t\t\t\t Polygon\t\t\t\t Convex Polygon" << endl;
11    cout << "\t\t Inside\t\t In\t\t Inside\t\t In" << endl;
// Dla kolejnych punktów wyznacz ich przynależność do wielokąta przy użyciu
// poszczególnych funkcji
12    while(cin >> p.x >> p.y) cout << p << "\t\t" <<
13       PointInsidePol(pol, p) << "\t\t\t" <<
14       PointInPol(pol, p) << "\t\t\t" <<
15       PointInsideConvexPol(pol, p) << "\t\t\t" <<
16       PointInConvexPol(pol, p) << endl;
17    return 0;
18 }
```

3.4. Punkty przecięcia

Literatura
[WDA] - 35.1

W niektórych sytuacjach stwierdzenie samego faktu przecinania się obiektów geometrycznych jest niewystarczające — potrzebne okazuje się również wyliczenie współrzędnych punktów przecięcia. W niniejszym rozdziale zajmiemy się tego typu problemami. Przedstawimy sposób wyznaczania punktów przecięcia prostych, odcinków, prostej z okręgiem oraz dwóch okręgów. Rozpoczniemy od przedstawienia problemu wyznaczania punktu przecięcia dwóch prostych.

Jeżeli przeanalizujemy zbiór punktów przecięcia prostych $P_1 \rightarrow P_2$ oraz $L_1 \rightarrow L_2$ to zauważymy, że może zachodzić jeden z trzech przypadków: proste mogą być do siebie równoległe i się w ogóle nie przecinać, mogą się pokrywać czyli mieć nieskończenie wiele punktów przecięcia, albo mieć tylko jeden punkt wspólny. Pierwsze dwa przypadki w prosty sposób można zbadać, sprawdzając wartość iloczynu wektorowego $(P_1 \mapsto P_2) \times (L_1 \mapsto L_2)$ — jeśli jest on równy 0, to znaczy, że proste są równoległe. W takim przypadku trzeba jeszcze sprawdzić, czy się pokrywają, co można w łatwy sposób zweryfikować choćby poprzez zbadanie przynależności punktu L_1 do prostej $P_1 \rightarrow P_2$.

Jeśli proste nie są równoległe, to istnieje dokładnie jeden punkt przecięcia, który można wyznaczyć, rozwiązując odpowiedni układ równań. Zauważmy, że dowolny punkt (x, y) należący do prostej $P_1 \rightarrow P_2$ można wyrazić w postaci $(b * P_1.x + (1 - b) * P_2.x, b * P_1.y + (1 - b) * P_2.y)$ dla pewnej rzeczywistej liczby b. Ponieważ poszukiwany punkt należy do obu prostych $P_1 \rightarrow P_2$ oraz $L_1 \rightarrow L_2$, to uzyskujemy w ten sposób układ równań z czterema

niewiadomymi x, y, b i c:

$$\begin{cases} x = b * x_1 + (1 - b) * x_2 \\ y = b * y_1 + (1 - b) * y_2 \\ x = c * x_3 + (1 - c) * x_4 \\ y = c * y_3 + (1 - c) * y_4 \end{cases}$$

Wyznaczając z niego niewiadome x oraz y, otrzymujemy:

$$x = \frac{x_2 * (x_3 * y_4 - x_4 * y_3 + (x_4 - x_3) * y_1) + x_1 * (x_4 * y_3 - x_3 * y_4 + (x_3 - x_4) * y_2)}{(x_1 - x_2) * (y_3 - y_4) - (x_3 - x_4) * (y_1 - y_2)}$$

$$y = \frac{y_2 * (x_3 * y_4 - x_4 * y_3) + y_1 * (x_4 * y_3 - x_3 * y_4) + x_2 * y_1 * (y_4 - y_3) + x_1 * y_2 * (y_3 - y_4)}{(x_1 - x_2) * (y_3 - y_4) - (x_3 - x_4) * (y_1 - y_2)}$$

Funkcją realizującą powyższą metodę jest **int** `LineCrossPoint(POINTD, POINTD, POINTD, POINTD, POINTD&)`, która jako parametry przyjmuje dwie proste wyznaczone przez pary punktów oraz dodatkowy punkt przekazywany przez referencję. Zwracana przez funkcję wartość może być następująca:

- 0 — proste nie przecinają się
- 1 — proste przecinają się w jednym punkcie
- 2 — proste pokrywają się (istnieje nieskończenie wiele punktów przecięcia)

W przypadku, gdy funkcja zwraca wartość 1, przekazany przez referencję punkt jest ustawiany na punkt przecięcia prostych. Implementacja tej funkcji przedstawiona jest na listingu 3.29.

Listing 3.29: Implementacja funkcji **int** `LineCrossPoint(POINTD, POINTD, POINTD, POINTD, POINTD&)`

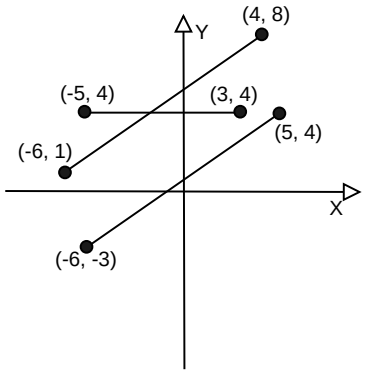
```
1 int LineCrossPoint(POINTD p1, POINTD p2, POINTD l1, POINTD l2, POINTD & prz) {
// Iloczyn wektorowy (p1 -> p2) i (l1 -> l2)
2   double s, t = (p1.x - p2.x) * (l1.y - l2.y) - (p1.y - p2.y) * (l1.x - l2.x);
// Iloczyn wektorowy (l2 -> p2) i (l1 -> l2)
3   s = (l2.x - p2.x) * (l1.y - l2.y) - (l2.y - p2.y) * (l1.x - l2.x);
// Jeśli proste są równoległe (t == 0), to istnieje nieskończenie wiele
// punktów wspólnych wtw gdy proste się pokrywają (iloczyn wektorowy s == 0)
4   if (IsZero(t)) return IsZero(s) ? 2 : 0;
5   s = s / t;
// Istnieje jeden punkt wspólny - wyznacz jego współrzędne
6   prz.x = s * p1.x + (1 - s) * p2.x;
7   prz.y = s * p1.y + (1 - s) * p2.y;
8   return 1;
9 }
```

Problem wyznaczania punktów przecięcia dwóch odcinków z teoretycznego punktu widzenia jest stosunkowo podobny, jednak istnieją tu pewne różnice związane ze sposobem reprezentacji wspólnego obszaru. Gdy odcinki częściowo się pokrywają, ich część wspólna nie jest

całą prostą je zawierającą. W takim przypadku, obszar stanowiący ich przecięcie można reprezentować również przy użyciu odcinka. Jeśli odcinki się przecinają i nie są równoległe, to sposób wyznaczania punktu przecięcia jest podobny do tego, wykorzystanego w przypadku dwóch prostych. Funkcja **bool** `SegmentCrossPoint(POINT, POINT, POINT, POINT, vector<POINTD>&)` przyjmuje jako parametry dwa odcinki reprezentowane przez pary punktów oraz referencję na wektor punktów v. Zwracanym wynikiem jest fałsz, jeżeli odcinki się nie przecinają, a prawda wpp. Jeśli część wspólna odcinków składa się tylko z jednego punktu, to jest on umieszczany w wektorze v. W przypadku, gdy odcinki pokrywają się, wektor v wypełniany jest dwoma punktami — końcami odcinka reprezentującego wspólny obszar. Implementacja omawianej funkcji znajduje się na listingu 3.30.

Listing 3.30: Implementacja funkcji **bool** `SegmentCrossPoint(POINT, POINT, POINT, POINT, vector<POINTD>&)`

```
01 inline bool SegmentCrossPoint(POINT p1, POINT p2, POINT l1, POINT l2,
02    vector<POINTD> &r) {
03    r.clear();
04    int w1 = sgn(Det(p1, p2, l1)), w2 = sgn(Det(p1, p2, l2)),
05       v1 = sgn(Det(l1, l2, p1)), v2 = sgn(Det(l1, l2, p2));
// Jeśli punkty l1 i l2 znajdują się po tej samej stronie prostej p1 -> p2
// lub p1 i p2 znajdują się po tej samej stronie prostej l1 -> l2,
// to odcinki nie przecinają się
06    if (w1*w2 > 0 || v1*v2 > 0) return false;
// Jeśli punkty l1 i l2 leżą na prostej p1 -> p2, to odcinki
// l1 -> l2 i p1 -> p2 są współliniowe
07    if (!w1 && !w2) {
// Zamiana par punktów reprezentujących odcinki, tak aby pierwsze punkty
// w parach były mniejsze w porządku po współrzędnych (x,y)
08       if (OrdXY(&p2, &p1)) swap(p1, p2);
09       if (OrdXY(&l2, &l1)) swap(l1, l2);
// Jeśli odcinki są rozłączne, to nie ma punktów przecięcia
10       if (OrdXY(&p2, &l1) || OrdXY(&l2,&p1)) return false;
// Wyznacz krańcowe punkty wspólne
11       if (p2 == l1) r.PB(POINTD(p2.x, p2.y));
12       else if (p1 == l2) r.PB(POINTD(l2.x, l2.y));
13       else {
14          r.PB(OrdXY(&p1, &l1) ? l1 : p1);
15          r.PB(OrdXY(&p2, &l2) ? p2 : l2);
16       }
17    }
// Jeśli jeden z odcinków jest zdegenerowany, to jest on punktem przecięcia
18    else if (l1 == l2) r.PB(l1);
19    else if (p1 == p2) r.PB(p2);
20    else {
// Wyznacz punkt przecięcia
21       double t = double(LL(l2.x - p2.x) * LL(l1.y - l2.y) - LL(l2.y - p2.y) *
22          LL(l1.x - l2.x)) / double(LL(p1.x - p2.x) * LL(l1.y - l2.y) -
23          LL(p1.y - p2.y) * LL(l1.x - l2.x));
24       r.PB(POINTD(t * p1.x + (1.0 - t) * p2.x, t * p1.y + (1.0 - t) * p2.y));
25    }
```



Rysunek 3.6: Możliwe przypadki przecinania się zbioru odcinków oraz prostych je zawierających

Listing 3.30: (c.d. listingu z poprzedniej strony)

```
26    return true;
27 }
```

Listing 3.31: Wynik działania funkcji **bool** `SegmentCrossPoint(POINT, POINT, POINT, POINT, vector<POINTD>&)` oraz **int** `LineCrossPoint(POINTD, POINTD, POINTD, POINTD, POINTD&)` dla zbioru odcinków i prostych z rysunku 3.6

```
(4, 8) - (-6, 1) oraz (-5, 4) - (3, 4)
Punkt przeciecia prostych: 1 (-1.71429, 4)
Punkt przeciecia odcinkow: 1 (-1.71429, 4)
(-6, -3) - (5, 4) oraz (-5, 4) - (3, 4)
Punkt przeciecia prostych: 1 (5, 4)
Punkt przeciecia odcinkow: 0
(-6, -3) - (5, 4) oraz (4, 8) - (-6, 1)
Punkt przeciecia prostych: 1 (-68.8571, -43)
Punkt przeciecia odcinkow: 0
```

Listing 3.32: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.31. Pełny kod źródłowy programu znajduje się w pliku `linesegcross.cpp`

```
01 int main() {
02    vector<POINT> b, e;
03    vector<POINTD> l;
04    int res;
05    POINT p1, p2;
06    POINTD p;
// Wczytaj wszystkie pary punktów wyznaczające proste i odcinki
07    while(cin >> p1.x >> p1.y >> p2.x >> p2.y) {
08       b.PB(p1); e.PB(p2);
09    }
// Dla każdej pary punktów wykonaj funkcje LineCrossPoint oraz SegmentCrossPoint
10    REP(x, SIZE(b)) REP(y, x) {
11
```

Listing 3.32: (c.d. listingu z poprzedniej strony)

```
12      cout << b[x] << " - " << e[x] <<
13         " oraz " << b[y] << " - " << e[y] << endl;
14      cout << "Punkt przeciecia prostych: " <<
15         (res = LineCrossPoint(b[x], e[x], b[y], e[y], p));
16      if (res == 1) cout << " " << p;
17      cout << endl;
18      cout << "Punkt przeciecia odcinkow: " <<
19         SegmentCrossPoint(b[x], e[x], b[y], e[y], l);
20      FOREACH(it, l) cout << " " << (*it);
21      cout << endl;
22   }
23   return 0;
24 }
```

Kolejną funkcją, służącą do wyznaczania punktów przecięcia obiektów geometrycznych, jest `vector<POINTD> LineCircleCross (POINTD, double, POINTD, POINTD)`. Wyznacza ona punkty przecięcia prostej z okręgiem. Przyjmowane przez tę funkcję parametry, to odpowiednio punkt stanowiący środek okręgu, jego promień oraz dwa punkty wyznaczające prostą. Wynikiem działania funkcji jest wektor punktów przecięcia. W zależności od wzajemnego położenia okręgu i prostej, wektor ten zawiera zero, jeden lub dwa elementy. W celu wyznaczenia poszukiwanych punktów przecięcia między okręgiem o środku w punkcie P i promieniu r oraz prostej wyznaczonej przez dwa punkty L_1 oraz L_2, należy rozwiązać następujący układ równań:

$$\begin{cases} x = b * L_1.x + (1-b) * L_2.x \\ y = b * L_1.y + (1-b) * L_2.y \\ (x - P.x)^2 + (y - P.y)^2 = r^2 \end{cases}$$

Działanie omawianej funkcji sprowadza się w zasadzie do wyznaczenia rozwiązań tego układu równań.

Listing 3.33: Implementacja funkcji `vector<POINTD> LineCircleCross(POINTD, double, POINTD, POINTD)`

```
// Funkcja wyznacza punkty przecięcia okręgu i prostej
01 vector<POINTD> LineCircleCross(POINTD p, double cr, POINTD p1, POINTD p2){
02    double a = sqr(p1.x) + sqr(p1.y) + sqr(p2.x) + sqr(p2.y) -
03       2.0 * (p1.x * p2.x + p1.y * p2.y);
04    double b = 2.0 * (p.x * (p2.x - p1.x) + p.y * (p2.y - p1.y) +
05       p1.x * p2.x + p1.y * p2.y - sqr(p2.x)-sqr(p2.y));
06    double c = -sqr(cr) + sqr(p2.x) + sqr(p2.y) + sqr(p.x) +
07       sqr(p.y) - 2.0 * (p.x * p2.x + p.y * p2.y);
08    double d = b * b - 4.0 * a * c;
09    vector<POINTD> r;
// Jeśli nie istnieje rozwiązanie równania kwadratowego,
// to brak punktów przecięcia
10    if (d < -EPS) return r;
11    double t = -b / (2.0 * a), e = sqrt(abs(d)) / (2.0 * a);
12    if (IsZero(d))
```

Listing 3.33: (c.d. listingu z poprzedniej strony)

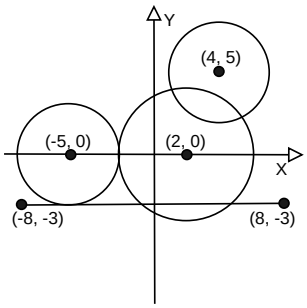
```
// Istnieje tylko jeden punkt przecięcia...
13      r.PB(POINTD(t * p1.x + (1.0 - t) * p2.x, t * p1.y + (1.0 - t) * p2.y));
14 else {
// Istnieją dwa punkty przecięcia
15      r.PB(POINTD((t + e) * p1.x + (1.0 - t - e) * p2.x,
16         (t + e) * p1.y + (1.0 - t - e) * p2.y));
17      r.PB(POINTD((t - e) * p1.x + (1.0 - t + e) * p2.x,
18         (t - e) * p1.y + (1.0 - t + e) * p2.y));
19    }
20    return r;
21 }
```

Ostatnią prezentowaną funkcją, służącą do wyznaczania punktów przecięcia, będzie `vector<POINTD> CirclesCross(POINTD, double, POINTD, double)`. Jako parametry funkcja ta przyjmuje środki oraz promienie dwóch okręgów (P_1, r_1, P_2, r_2), natomiast jako wynik zwraca wektor punktów przecięcia. Wyznaczenie punktów przecięcia dwóch okręgów sprowadza się jak i poprzednio do rozwiązania odpowiedniego układu równań.

$$\begin{cases} (x - P_1.x)^2 + (y - P_1.y)^2 = r_1^2 \\ (x - P_2.x)^2 + (y - P_2.y)^2 = r_2^2 \end{cases}$$

Listing 3.34: Implementacja funkcji `vector<POINTD> CirclesCross(POINTD, double, POINTD, double)`

```
01 vector<POINTD> CirclesCross(POINTD c1, double c1r, POINTD c2, double c2r) {
02    vector<POINTD> r;
03    c2.x -= c1.x;
04    c2.y -= c1.y;
// Jeśli okręgi są współśrodkowe, to nie wyznaczamy punktów przecięcia
05    if (IsZero(c2.x) && IsZero(c2.y)) return r;
06    double A = (-sqr(c2r) + sqr(c1r) + sqr(c2.x) + sqr(c2.y)) / 2.0;
// Jeśli środki okręgów mają tą samą współrzędną y...
07    if (IsZero(c2.y)) {
08       double x = A / c2.x;
09       double y2 = sqr(c1r) - sqr(x);
10       if (y2 < -EPS) return r;
// Jeśli okręgi są styczne...
11       if (IsZero(y2)) r.PB(POINTD(c1.x + x, c1.y));
12       else {
// Okręgi przecinają się
13          r.PB(POINTD(c1.x + x, c1.y + sqrt(y2)));
14          r.PB(POINTD(c1.x + x, c1.y - sqrt(y2)));
15       }
16       return r;
17    }
18    double a = sqr(c2.x) + sqr(c2.y);
19    double b = -2.0 * A * c2.x;
20    double c = A * A - sqr(c1r) * sqr(c2.y);
```



Rysunek 3.7: Różne możliwości przecinania się okręgów oraz prostych

Listing 3.34: (c.d. listingu z poprzedniej strony)

```
21    double d = b * b - 4.0 * a * c;
22    if (d < -EPS) return r;
23    double x = -b / (2.0 * a);
// Jeśli okręgi są styczne...
24    if (IsZero(d)) r.PB(POINTD(c1.x + x, c1.y + (A - c2.x * x) / c2.y));
25    else {
// Okręgi przecinają się
26       double e = sqrt(d) / (2.0 * a);
27       r.PB(POINTD(c1.x + x + e, c1.y + (A - c2.x * (x + e)) / c2.y));
28       r.PB(POINTD(c1.x + x - e, c1.y + (A - c2.x * (x - e)) / c2.y));
29    }
30    return r;
31 }
```

Listing 3.35: Wynik działania funkcji `vector<POINTD> CirclesCross(POINTD, double, POINTD, double)` oraz `vector<POINTD> LineCircleCross(POINTD, double, POINTD, POINTD)` na przykładzie rysunku 3.7

```
Przeciecie prostej (-8, -3) - (8, -3) i okregu:
      (-5, 0, 3) -  (-5, -3)
      (2, 0, 4) -  (-0.645751, -3) (4.64575, -3)
      (4, 5, 3) - brak punktow przeciecia
Przeciecie okregow:
      (2, 0, 4) oraz (-5, 0, 3) -  (-2, 0)
      (4, 5, 3) oraz (-5, 0, 3) - brak punktow przeciecia
      (4, 5, 3) oraz (2, 0, 4) -  (5.28141, 2.28744) (1.20135, 3.91946)
```

Listing 3.36: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.35. Pełny kod źródłowy programu znajduje się w pliku `linecirclecross.cpp`

```
01 int main() {
02    vector<POINT> cCen;
03    VI cR;
04    vector<POINTD> res;
05    POINT l1, l2, p; int r;
06
```

Listing 3.36: (c.d. listingu z poprzedniej strony)

```
// Wczytaj współrzędne punktów wyznaczających prostą
07    cin >> l1.x >> l1.y >> l2.x >> l2.y;
// Wczytaj opisy kolejnych okręgów
08    while(cin >> p.x >> p.y >> r) {
09       cR.PB(r); cCen.PB(p);
10    }
// Wyznacz punkty przecięcia prostej i okręgu
11    cout << "Przeciecie prostej " << l1 << " - " << l2 <<
12       " i okregu:" << endl;
13    REP(x, SIZE(cCen)) {
14       res = LineCircleCross(cCen[x], cR[x], l1, l2);
15       cout << "\t(" << cCen[x].x << ", " << cCen[x].y <<
16          ", " << cR[x] << ") - ";
17       if (!SIZE(res)) cout << "brak punktow przeciecia";
18       FOREACH(it, res) cout << " " << (*it);
19       cout << endl;
20    }
// Wyznacz punkty przecięcia dwóch okręgów
21    cout << "Przeciecie okregow:" << endl;
22    REP(x, SIZE(cCen)) REP(y, x) {
23       res = CirclesCross(cCen[x], cR[x], cCen[y], cR[y]);
24       cout << "\t(" << cCen[x].x << ", " << cCen[x].y << ", " << cR[x] <<
25          ") oraz (" <<
26          cCen[y].x << ", " << cCen[y].y << ", " << cR[y] << ") - ";
27       if (!SIZE(res)) cout << "brak punktow przeciecia";
28       FOREACH(it, res) cout << " " << *(it);
29       cout << endl;
30    }
31    return 0;
32 }
```

Zadanie: Akcja komandosów

Pochodzenie: XII Olimpiada Informatyczna

Rozwiązanie: `commando.cpp`

Na Pustyni Błędowskiej odbywa się w tym roku Bardzo Interesująca i Widowiskowa Akcja Komandosów (BIWAK). Podstawowym elementem BIWAK-u ma być neutralizacja bomby, która znajduje się gdzieś na pustyni, jednak nie wiadomo dokładnie gdzie.

Pierwsza część akcji to desant z powietrza. Z helikoptera krążącego nad pustynią, wyskakują pojedynczo, w ustalonej kolejności komandosi. Gdy któryś z komandosów wyląduje w jakimś miejscu, okopuje się i już się nie rusza z miejsca. Dopiero potem może wyskoczyć kolejny komandos.

Dla każdego komandosa określona jest pewna odległość rażenia. Jeśli komandos przebywa w tej odległości (lub mniejszej) od bomby, to w przypadku jej ewentualnej eksplozji zginie. Dowództwo chce zminimalizować liczbę komandosów biorących udział w akcji, ale chce mieć pewność, że w przypadku wybuchu bomby, przynajmniej jeden z komandosów przeżyje.

Na potrzeby zadania przyjmujemy, że Pustynia Błędowska jest płaszczyzną, a komandosów, którzy się okopali utożsamiamy z punktami. Mamy dany ciąg kolejno mogących wyskoczyć komandosów. Żaden z nich nie może opuścić swojej kolejki, tzn. jeśli i-ty komandos wyskakuje z samolotu, to wszyscy poprzedni wyskoczyli już wcześniej. Dla każdego z komandosów znamy jego odległość zagrożenia rażenia oraz współrzędne punktu, w którym wyląduje, o ile w ogóle wyskoczy.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opisy komandosów,
- wyznaczy minimalną liczbę komandosów, którzy muszą wyskoczyć,
- wypisze wynik na standardowe wyjście.

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita n ($2 \leqslant n \leqslant 2\,000$) — liczba komandosów. W kolejnych n wierszach opisani są komandosi — po jednym w wierszu. Opis każdego komandosa składa się z trzech liczb całkowitych: x, y i r ($-1\,000 \leqslant x, y \leqslant 1\,000, 1 \leqslant r \leqslant 5\,000$). Punkt (x, y) to miejsce, gdzie wyląduje komandos, a r to jego odległość „rażenia". Jeśli komandos znajdzie się w odległości r lub mniejszej od bomby, to w przypadku jej wybuchu zginie.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia Twój program powinien zapisać jedną liczbę całkowitą - minimalną liczbę komandosów, którzy muszą wyskoczyć, aby zapewnić, że co najmniej jeden z nich przeżyje, lub jedno słowo NIE jeśli nie jest możliwe, aby mieć pewność, że któryś z komandosów przeżyje.

Przykład

Dla następującego wejścia:

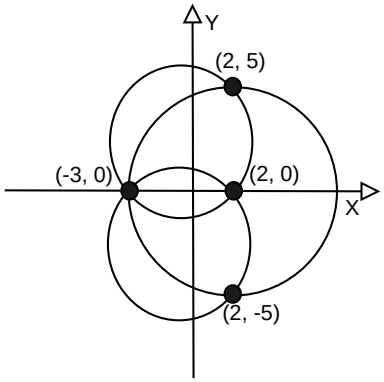
```
5
2  2  4
7  2  3
4  3  1
5  7  1
8  7  1
```

Poprawnym rozwiązaniem jest:

```
4
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 191	acm.sgu.ru - zadanie 253	spoj.sphere.pl - zadanie 182
acm.sgu.ru - zadanie 124	spoj.sphere.pl - zadanie 102	spoj.sphere.pl - zadanie 332
acm.sgu.ru - zadanie 129	acm.sgu.ru - zadanie 198	spoj.sphere.pl - zadanie 272
acm.uva.es - zadanie 10084	acm.sgu.ru - zadanie 267	



Rysunek 3.8: Rysunek prezentujący okręgi wyznaczone przez zbiór czterech punktów. Są tylko trzy, a nie cztery okręgi opisane na tych punktach, ponieważ punkty $(2,-5)$, $(2,0)$ oraz $(2,5)$ są współliniowe i nie wyznaczają okręgu

3.5. Trzy punkty — okrąg

W poprzednich rozdziałach przedstawiliśmy wiele różnych funkcji operujących na okręgach. Przyjmowaliśmy wtedy, że okrąg reprezentowany jest w postaci pary (środek okręgu, promień). Nie jest to jednak jedyny sposób — w wielu przypadkach okręgi reprezentuje się jako trzy różne punkty należące do ich obwodu. W sytuacjach, w których konieczne jest korzystanie z różnych reprezentacji okręgu (co może być wymuszone formatem danych wejściowych w zadaniu), pojawia się potrzeba konwertowania jednej reprezentacji do innej. Konwersja reprezentacji okręgu $((C.x, C.y), r)$ używanej w funkcjach z tej książki do reprezentacji wykorzystującej trzy punkty jest prosta. Jako reprezentantów okręgu można użyć przykładowo następujący zbiór punktów: $\{(C.x, C.y+r), (C.x, C.y-r), (C.x+r, C.y)\}$. Konwersja w drugą stronę wydaje się być znacznie bardziej skomplikowana — nad nią właśnie skupimy się w aktualnym rozdziale.

Załóżmy zatem, że mamy dane trzy punkty: P_1, P_2 oraz P_3, które nie leżą na jednej prostej. W celu wyznaczenia środka oraz promienia okręgu opisanego na tych trzech punktach, należy rozwiązać układ równań z trzema niewiadomymi r, $C.x$ oraz $C.y$ postaci:

$$\begin{cases} r^2 = (C.x - P_1.x)^2 + (C.y - P_1.y)^2 \\ r^2 = (C.x - P_2.x)^2 + (C.y - P_2.y)^2 \\ r^2 = (C.x - P_3.x)^2 + (C.y - P_3.y)^2 \end{cases}$$

Prezentowana na listingu 3.37 funkcja służąca do konwertowania reprezentacji okręgu nie implementuje rozwiązywania powyższego układu równań wprost, lecz wykorzystuje przedstawioną wcześniej funkcję do wyznaczania punktu przecięcia dwóch prostych. Przyjmuje ona jako parametry trzy punkty oraz referencję na punkt C i zmienną r typu **double**. Jeśli na trzech określonych przez parametry punktach można opisać okrąg, funkcja zwraca prawdę, punktowi C przypisywany jest środek, a zmiennej r promień wyznaczonego okręgu.

Listing 3.37: Implementacja funkcji **bool** `ThreePointCircle(POINTD, POINTD, POINTD, POINTD&, double&)`

```
// Funkcja znajduje okrąg wyznaczony przez trzy punkty lub zwraca fałsz,
// jeśli taki okrąg nie istnieje
01 bool ThreePointCircle(POINTD p1, POINTD p2, POINTD p3, POINTD &c, double &r){
// Wyznacz punkt przecięcia symetralnych trójkąta (p1, p2, p3)
```

Listing 3.37: (c.d. listingu z poprzedniej strony)

```
02    if (LineCrossPoint(POINTD((p1.x + p2.x) / 2.0, (p1.y + p2.y) / 2.0),
03       POINTD((p1.x + p2.x) / 2.0 + p2.y - p1.y, (p1.y + p2.y) / 2.0 + p1.x
04       - p2.x), POINTD((p1.x + p3.x) / 2.0, (p1.y + p3.y) / 2.0),
05       POINTD((p1.x + p3.x) / 2.0 + p3.y - p1.y,
06          (p1.y + p3.y) / 2.0 + p1.x - p3.x) ,c) != 1)
07    return false;
// Wylicz promień okręgu o środku w punkcie c
08    r = sqrt(sqr(p1.x - c.x) + sqr(p1.y - c.y));
09    return true;
10 }
```

Listing 3.38: Przykład użycia funkcji **bool** `ThreePointCircle(POINTD, POINTD, POINTD, POINTD&, double&)` na przykładzie zbioru punktów przedstawionych na rysunku 3.8

```
Punkty: (-3, 0), (2, 0), (2, 5)
Srodek okregu = (-0.5, 2.5), promien = 3.53553
Punkty: (2, -5), (2, 0), (2, 5)
Punkty sa wspolliniowe
Punkty: (2, -5), (-3, 0), (2, 5)
Srodek okregu = (2, 0), promien = 5
Punkty: (2, -5), (-3, 0), (2, 0)
Srodek okregu = (-0.5, -2.5), promien = 3.53553
```

Listing 3.39: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.38. Pełny kod źródłowy programu znajduje się w pliku `threepointcircle.cpp`

```
01 int main() {
02    vector<POINT> l;
03    POINT p;
04    POINTD po;
05    double r;
06    bool res;
// Wczytaj listę punktów
07    while (cin >> p.x >> p.y) l.PB(p);
// Dla każdej trójki punktów, wyznacz okrąg na nich opisany
08    REP(x, SIZE(l)) REP(y, x) REP(z, y) {
09       cout << "Punkty: " << l[x] << ", " << l[y] << ", " << l[z] << endl;
10       if (!ThreePointCircle(l[x], l[y], l[z], po, r))
11          cout << "Punkty sa wspolliniowe" << endl;
12       else cout << "Srodek okregu = " << po << ", promien = " << r << endl;
13    }
14    return 0;
15 }
```

3.6. Sortowanie kątowe

Sortowanie kątowe zbioru punktów S dookoła wektora $C \mapsto D$ polega na takim uporządkowaniu punktów, że dla dwóch dowolnych punktów P_1 oraz P_2 ze zbioru S, P_1 znajduje się przed punktem P_2, jeśli kąty skierowane $\angle DCP_1$ oraz $\angle DCP_2$ zachowują własność $\angle DCP_1 < \angle DCP_2$. W przypadku, gdy $\angle DCP_1 = \angle DCP_2$, umawiamy się, że jako wcześniejszy wybierany jest punkt znajdujący się bliżej punktu C.

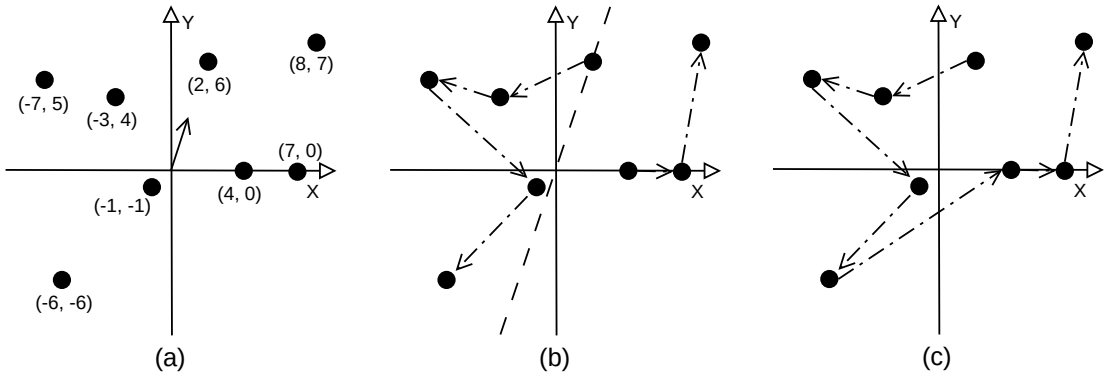
Sortowanie kątowe można zrealizować przy użyciu funkcji `sort` pochodzącej z biblioteki STL. Jedyne co trzeba zrobić, to dostarczyć operator, który dla dwóch danych punktów będzie w stanie stwierdzić, który z nich jest mniejszy. Ze względu jednak na konieczność rozpatrzenia wielu przypadków, implementacja takiego operatora jest stosunkowo skomplikowana. Poza tym, funkcja `sort` nie przekazuje żadnej informacji na temat wektora, względem którego wykonywane jest sortowanie i trzeba by było operatorowi porównującemu punkty dostarczyć informację na temat tego wektora.

Prezentowana w tym rozdziale funkcja również wykorzystuje funkcję `sort`, jednak wcześniej dokonuje podziału zbioru punktów na dwie części — punkty znajdujące się po lewej stronie prostej $C \rightarrow D$ oraz resztę. Dzięki wprowadzeniu takiego podziału, znika wiele przypadków skrajnych, które normalnie należałoby rozpatrzyć, a w rezultacie implementacja operatora wykorzystywanego przez funkcję `sort` jest prosta. Po posortowaniu obu zbiorów punktów wystarczy je z powrotem scalić. Złożoność czasowa sortowania, ze względu na wykorzystanie funkcji `sort` to $O(n*log(n))$. Rysunek 3.9 prezentuje proces wyznaczania porządku kątowego dla przykładowego zbioru punktów.

Prezentowana na listingu 3.40 funkcja `vector<POINT*> AngleSort(vector<POINT>& p, POINT s, POINT k)` przyjmuje jako parametry listę punktów do posortowania oraz dwa punkty C i D. Jako wynik działania zwracany jest wektor wskaźników na punkty w kolejności zgodnej z porządkiem kątowym.

Listing 3.40: Implementacja funkcji `vector<POINT*> AngleSort(vector<POINT>&, POINT, POINT)`

```
// Wskaźnik na punkt centralny (używane przez funkcję porównującą)
01 POINT* RSK;
// Funkcja porównująca punkty
02 bool Rcmp(POINT *a, POINT *b) {
03    LL w = Det((*RSK), (*a), (*b));
04    if (w == 0) return abs(RSK->x - a->x) + abs(RSK->y - a->y) <
05       abs(RSK->x - b->x) + abs(RSK->y - b->y);
06    return w > 0;
07 }
// Funkcja sortuje po kącie odchylenia zbiór punktów względem punktu centralnego
// s zaczynając od wektora s -> k
08 vector<POINT*> AngleSort(vector<POINT>& p, POINT s, POINT k) {
09    RSK = &s;
10    vector<POINT*> l, r;
// Każdy punkt, który podlega sortowaniu, zostaje wstawiony do jednego
// z wektorów l lub r, w zależności od tego,
// czy znajduje się po lewej czy po prawej stronie prostej s -> k
11    FOREACH(it, p) {
12       LL d = Det(s, k, (*it));
13       (d > 0 || (d==0 && (s.x == it->x ? s.y < it->y : s.x < it->x)))
14          ? l.PB(&*it) : r.PB(&*it);
```



Rysunek 3.9: **(a)** Zbiór punktów do posortowania kątowego względem wektora $(0,0) \mapsto (1,3)$. **(b)** Rozdzielenie zbioru punktów na dwa podzbiory i posortowanie punktów w ich obrębie. **(c)** Scalenie obu zbiorów

Listing 3.40: (c.d. listingu z poprzedniej strony)

```
15    }
// Posortuj niezależnie punkty w obu wyznaczonych wektorach
16    sort(ALL(l), Rcmp);
17    sort(ALL(r), Rcmp);
// Wstaw wszystkie punkty z wektora r na koniec wektora l
18    FOREACH(it, r) l.PB(*it);
19    return l;
20 }
```

Listing 3.41: Przykład działania funkcji `vector<POINT*> AngleSort(vector<POINT>&, POINT, POINT)` dla przykładowego zbioru punktów z rysunku 3.9

```
(2, 6) (-3, 4) (-7, 5) (-1, -1) (-6, -6) (4, 0) (7, 0) (8, 7)
```

Listing 3.42: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.41. Pełny kod źródłowy programu znajduje się w pliku `anglesort.cpp`

```
01 int main() {
02    POINT c, k;
03    int n;
// Wczytaj liczbę punktów oraz współrzędne wektora, względem którego wykonywane
// będzie sortowanie
04    cin >> n >> c.x >> c.y >> k.x >> k.y;
05    vector<POINT> l(n);
// Wczytaj wszystkie punkty
06    REP(x, n) cin >> l[x].x >> l[x].y;
// Posortuj punkty
07    vector<POINT *> res = AngleSort(l, c, k);
08    FOREACH(it, res) cout << " " << *(*it);
09    return 0;
10 }
```

Zadanie: Ołtarze

Pochodzenie: VI Olimpiada Informatyczna

Rozwiązanie: `altar.cpp`

Według chińskich wierzeń ludowych złe duchy mogą poruszać się tylko po linii prostej. Ma to istotne znaczenie przy budowie świątyń. Świątynie są budowane na planach prostokątów o bokach równoległych do kierunków północ-południe oraz wschód-zachód. Żadne dwa z tych prostokątów nie mają punktów wspólnych. Po środku jednej z czterech ścian jest wejście, którego szerokość jest równa połowie długości tej ściany. W centrum świątyni (na przecięciu przekątnych prostokąta) znajduje się ołtarz. Jeśli znajdzie się tam zły duch, świątynia zostanie zhańbiona. Tak może się zdarzyć, jeśli istnieje półprosta (w płaszczyźnie równoległej do powierzchni terenu), która biegnie od ołtarza w centrum świątyni przez otwór wejściowy aż do nieskończoności, nie przecinając i nie dotykając po drodze żadnej ściany, tej lub innej świątyni.

Zadanie

Napisz program, który:

- wczyta opisy świątyń,
- sprawdzi, które świątynie mogą zostać zhańbione,
- wypisze ich numery na standardowe wyjście

Wejście

W pierwszym wierszu wejścia zapisana jest jedna liczba naturalna n, $1 \leqslant n \leqslant 1\,000$, będąca liczbą świątyń. W każdym z kolejnych n wierszy znajduje się opis jednej świątyni (w i-tym z tych wierszy opis świątyni numer i). Opis świątyni składa się z czterech nieujemnych liczb całkowitych, nie większych niż 8 000 oraz jednej litery E, W, S lub N. Pierwsze dwie liczby, to współrzędne północno-zachodniego narożnika świątyni, a dwie następne, to współrzędne przeciwległego, południowo-wschodniego narożnika. Określając współrzędne punktu podajemy najpierw jego długość geograficzną (która rośnie z zachodu na wschód), a następnie — szerokość geograficzną, która rośnie z południa na północ. Piąty element opisu wskazuje ścianę, na której znajduje się wejście do świątyni (E — wschodnią, W — zachodnią, S — południową, N — północną). Kolejne elementy opisu świątyni są pooddzielane pojedynczymi odstępami.

Wyjście

W kolejnych wierszach wyjścia, Twój program powinien zapisać w porządku rosnącym numery świątyń, które mogą zostać zhańbione przez złego ducha, każdy numer w osobnym wierszu.

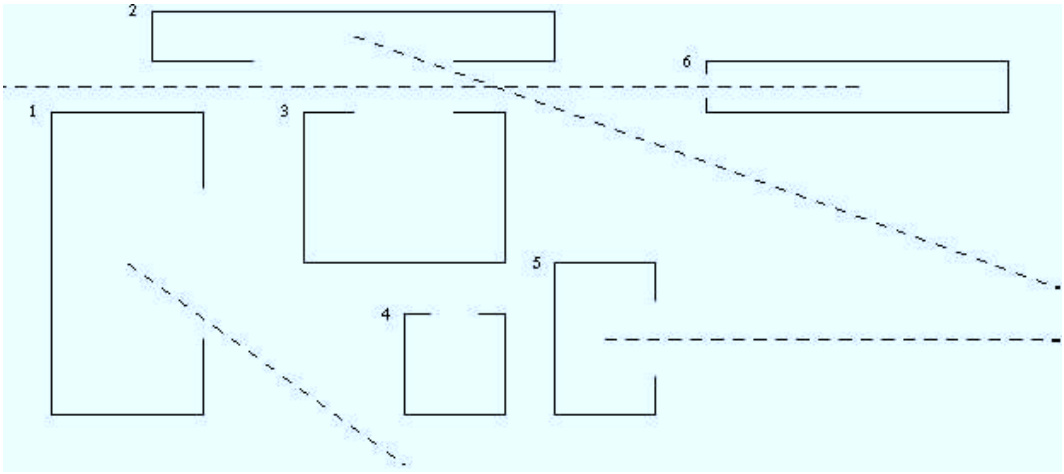
Przykład

Dla następującego wejścia:

```
6
1  7  4  1  E
3  9  11  8  S
6  7  10  4  N
8  3  10  1  N
11  4  13  1  E
14  8  20  7 W
```

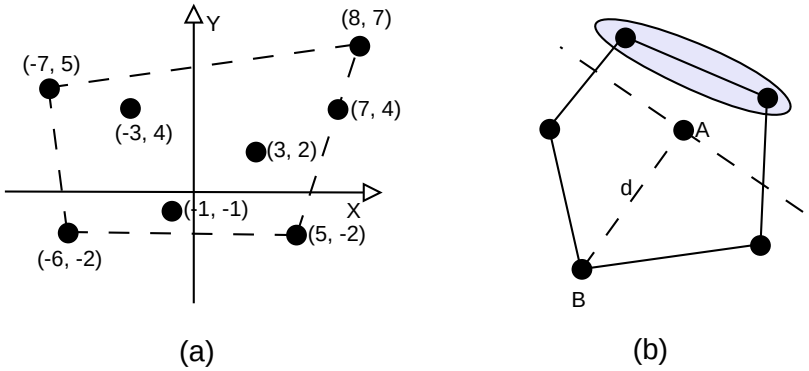
Poprawnym rozwiązaniem jest:

```
1
2
5
6
```



Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10002	acm.uva.es - zadanie 10927	spoj.sphere.pl - zadanie 202



Rysunek 3.10: **(a)** Zbiór ośmiu punktów na płaszczyźnie oraz najmniejsza wypukła otoczka dla tego zbioru punków. **(b)** Dla każdej pary punktów oddalonych o d, z których co najmniej jeden nie leży na wypukłej otoczce, istnieje para punktów oddalona o $l > d$.

3.7. Wypukła otoczka

Literatura
[WDA] - 35.3
[ASD] - 8.3
[RII] - 4

Dla danego zbioru punktów S, wypukła otoczka jest to wielokąt wypukły, który zawiera wszystkie punkty ze zbioru S. Najmniejsza wypukła otoczka cechuje się dodatkowo najmniejszym możliwym polem. Ważną własnością najmniejszej wypukłej otoczki pomagającą w jej wyznaczaniu jest to, że każdy jej wierzchołek to również punkt ze zbioru S (patrz rysunek 3.10.a)

Istnieje wiele algorytmów służących do wyznaczania najmniejszej wypukłej otoczki. Jednym z nich jest algorytm Grahama, który dla n-elementowego zbioru punktów działa w czasie $O(n * log(n))$. Algorytm ten najpierw wyznacza punkt zbioru o najmniejszej odciętej (współrzędnej x), a następnie sortuje wszystkie inne w kolejności rosnących kątów odchylenia (patrz rysunek 3.11.a). Wszystkie punkty są następnie przetwarzane w takiej kolejności — dodaje się je do konstruowanej wypukłej otoczki, usuwając jednocześnie punkty dodane poprzednio, o ile kąt utworzony pomiędzy trzema ostatnio dodanymi do otoczki punktami wyznacza skręt w prawo (rysunek 3.11.d). Proces konstrukcji wypukłej otoczki dla przykładowego zbioru punktów został przedstawiony na rysunku 3.11.

Przedstawiona na listingu 3.43 funkcja `vector<POINT*> ConvexHull(vector<POINT>&)` przyjmuje jako parametr wektor punktów, a zwraca wektor wskaźników na punkty należące do wypukłej otoczki. Wskaźniki te są uporządkowane w kolejności odwrotnej do ruchu wskazówek zegara. Przedstawiona funkcja jest modyfikacją opisanego algorytmu Grahama, mającą na celu skrócenie implementacji oraz zwiększenie wydajności algorytmu. Punkty nie są sortowane po kącie odchylenia, lecz w porządku „po współrzędnych", natomiast konstrukcja otoczki realizowana jest w dwóch fazach — dolnej oraz górnej połówki.

Listing 3.43: Implementacja funkcji `vector<POINT*> ConvexHull(vector<POINT>&)`

```
01 #define XCAL {while(SIZE(w) > m && Det((*w[SIZE(w) - 2]), (*w[SIZE(w) - 1]), \
02    (*s[x])) <= 0) w.pop_back(); w.PB(s[x]);}
// Funkcja wyznaczająca wypukłą otoczkę dla zbioru punktów
03 vector<POINT*> ConvexHull(vector<POINT>& p) {
04    vector<POINT*> s, w;
// Wypełnij wektor s wskaźnikami do punktów,
// dla których konstruowana jest wypukła otoczka
05    FOREACH(it, p) s.PB(&*it);
```

Listing 3.43: (c.d. listingu z poprzedniej strony)

```
// Posortuj wskaźniki punktów w kolejności
// (niemalejące współrzędne x, niemalejące współrzędne y)
06    sort(ALL(s), OrdXY);
07    int m = 1;
// Wyznacz dolną część wypukłej otoczki - łączącą najbardziej lewy - dolny
// punkt z najbardziej prawym - górnym punktem
08    REP(x, SIZE(s)) XCAL
09    m = SIZE(w);
// Wyznacz górną część otoczki
10    FORD(x, SIZE(s) - 2, 0) XCAL
// Usuń ostatni punkt (został on wstawiony do otoczki dwa razy)
11    w.pop_back();
12    return w;
13 }
```

Wiele problemów dla zbioru punktów można rozwiązać, wyznaczając najpierw wypukłą otoczkę. Przykładem może być problem znajdowania pary najbardziej oddalonych punktów. Oczywistym jest, że punkty takie muszą znajdować się na wypukłej otoczce rozpatrywanego zbioru. Wybierając bowiem parę punktów A i B oddalonych od siebie o d, z których przynajmniej jeden nie leży na wypukłej otoczce (załóżmy bez straty ogólności, że jest nim punkt A) oraz prowadząc przez punkt A prostą l prostopadłą do odcinka $A \mapsto B$, okazuje się, że dla wszystkich punktów C, leżących po przeciwnej stronie prostej l niż punkt B, zachodzi $|B \mapsto C| > d$ (patrz rysunek 3.10.b). W celu wyznaczenia najdalszej pary punktów, należy dla każdego punktu leżącego na wypukłej otoczce wyznaczyć punkt dla niego najdalszy oraz wybrać spośród wszystkich znalezionych par tą najbardziej oddaloną. Można tego dokonać w czasie liniowym poprzez wyznaczanie kolejnych par antypodycznych punktów, co daje sumaryczną złożoność czasową algorytmu $O(n * log(n))$.

Listing 3.44: Przykład działania funkcji `vector<POINT*> ConvexHull(vector<POINT>&)` dla zbioru punktów z rysunku 3.10. Do wyznaczanej wypukłej otoczki należą tylko skrajne punkty krawędzi — punkt $(7, 4)$ nie należy do wyznaczonej otoczki.

```
(-7, 5) (-6, -2) (5, -2) (8, 7)
```

Listing 3.45: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.44. Pełny kod źródłowy programu znajduje się w pliku `convexhull_str.cpp`

```
01 int main() {
02    int n;
03    vector<POINT> l;
04    POINT p;
// Wczytaj liczbę punktów
05    cin >> n;
// Wczytaj wszystkie punkty
06    REP(x, n) {
07       cin >> p.x >> p.y;
08       l.PB(p);
09    }
```

Listing 3.45: (c.d. listingu z poprzedniej strony)

```
// Wyznacz wypukłą otoczkę
10   vector<POINT *> res = ConvexHull(l);
11   FOREACH(it, res)
12   cout << " " << (**it);
13   return 0;
14 }
```

Zadanie: Pomniki

Pochodzenie: Zadanie przygotowane na potrzeby książki

Rozwiązanie: `monuments.cpp`

W Bajtocji od jakiegoś czasu trwa akcja, mająca na celu zwiększenie atrakcyjności turystycznej miasta. Sadzone są drzewa, trawa malowana jest na zielono... Służby miejskie postanowiły wybudować również wiele ozdobnych pomników z Bito-Kryptu. Pojawił się jednak pewien problem — w Bajtocji znajduje się potężna sieć Bito-serwerów, komunikujących się przy użyciu fal radiowych. Budowa pomników z Bito-Kryptu na obszarze aktywnej komunikacji radiowej może zakłócać transmisję danych. Transmisja może zostać zakłócona, jeśli pomnik znajduje się we wnętrzu pewnego trójkąta, którego wierzchołkami są lokalizacje Bito-serwerów. Służby miejskie zwróciły się do Ciebie z prośbą o stwierdzenie, dla każdej sugerowanej lokalizacji pomnika, czy nie będzie ona zakłócać komunikacji.

Zadanie

Napisz program, który:

- wczyta lokalizacje Bito-serwerów oraz sugerowane lokalizacje pomników,
- dla każdego pomnika stwierdzi, czy może on zakłócać łączność,
- wypisze wynik

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite n i m ($1 \leqslant n, m \leqslant 100\,000$) — liczbę Bito-serwerów oraz liczbę sugerowanych lokalizacji pomników. Następne n wierszy zawiera po dwie liczby całkowite x, y ($1\,000\,000 \leqslant x, y \leqslant 1\,000\,000$), oznaczające współrzędne kolejnych Bito-serwerów. Kolejnych m wierszy zawiera po dwie liczby całkowite x, y ($1\,000\,000 \leqslant x, y \leqslant 1\,000\,000$), oznaczające współrzędne sugerowanych lokalizacji pomników.

Wyjście

Twój program powinien wypisać dokładnie m wierszy — w i-tym wierszu wyjścia powinno się znaleźć słowo TAK, jeśli lokalizacja pomnika jest bezpieczna (nie będzie powodowała zakłóceń), lub słowo NIE w przeciwnym przypadku.

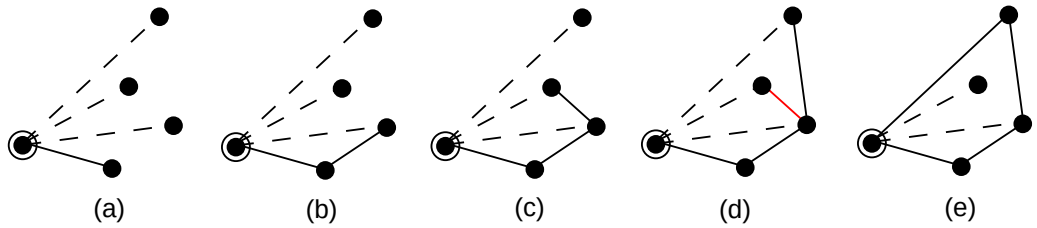
Przykład

Dla następującego wejścia:

```
4  5
0  0
10  0
10  10
0  10
5  5
12  1
9  9
10  5
3  10
```

Poprawnym rozwiązaniem jest:

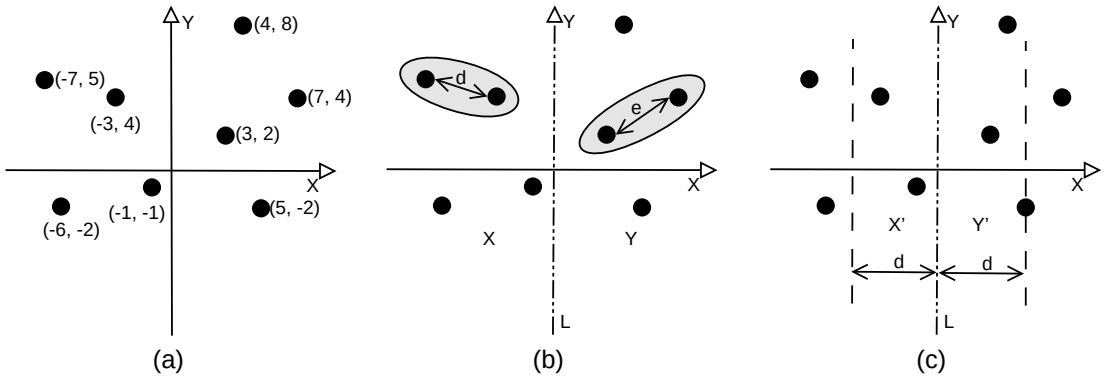
```
NIE
TAK
NIE
TAK
TAK
```



Rysunek 3.11: Proces konstruowania wypukłej otoczki. **(a)** Wyznaczenie skrajnego punktu, uporządkowanie punktów po kątach odchylenia oraz dodanie pierwszej krawędzi. **(b-c)** Przetworzenie kolejnych wierzchołków oraz dodanie krawędzi do konstruowanej wypukłej otoczki. **(d)** Usunięcie poprzedniej krawędzi leżącej na lewo od nowo dodanej. **(e)** Konstrukcja wypukłej otoczki zakończona.

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10078	acm.uva.es - zadanie 218	spoj.sphere.pl - zadanie 228
acm.uva.es - zadanie 109	spoj.sphere.pl - zadanie 26	acm.sgu.ru - zadanie 277
acm.uva.es - zadanie 10065	acm.sgu.ru - zadanie 227	acm.uva.es - zadanie 10135
		acm.sgu.ru - zadanie 290



Rysunek 3.12: **(a)** Zbiór punktów, dla których wyznaczana jest para najbliższych punktów. **(b)** Rozdzielenie zbioru punktów na dwa podzbiory X oraz Y. Wyznaczenie w obrębie tych zbiorów par najbliższych punktów oddalonych odpowiednio o d i e. **(c)** Scalenie podrozwiązań — następuje sprawdzenie par punktów ze zbiorów X' oraz Y' oddalonych co najwyżej o $min(d, e) = d$ od prostej l.

3.8. Para najbliższych punktów

Literatura
[WDA] - 35.4
[ASD] - 8.4.1

W rozdziale dotyczącym wyznaczania wypukłej otoczki zbioru punktów wspomnieliśmy o algorytmie wyznaczania pary najdalszych punktów. W niniejszym rozdziale zajmiemy się podobnym problemem — dla danego zbioru punktów S na płaszczyźnie należy wyznaczyć parę najbliższych punktów.

Nasuwającym się rozwiązaniem tego problemu jest sprawdzenie każdej pary punktów oraz wybranie tych dwóch, które są najbliżej siebie. Rozwiązanie takie jest bardzo proste w implementacji, niestety jego złożoność czasowa nie jest zachwycająca i wynosi $O(n^2)$.

Istnieje algorytm pozwalający na znalezienie pary najbliższych punktów w czasie $O(n * log(n))$ i działający w oparciu o technikę dziel i zwyciężaj. Na samym początku tworzone są dwie kopie zbioru punktów. Jedna z nich sortowana jest w kolejności niemalejących odciętych (zbiór ten nazwiemy S_x), druga w kolejności niemalejących rzędnych (zbiór S_y). Po wykonaniu tego kroku następuje przejście do fazy dzielenia zbioru punktów. Wyznaczana jest prosta l, równoległa do osi y, taka że po obu jej stronach znajduje się tyle samo punktów ze zbioru S (patrz rysunek 3.12.b). W ten sposób powstają dwa zbiory punktów X oraz Y. X zawiera punkty położone na lewo od prostej l, natomiast Y — punkty położone na prawo od prostej l.

Rekurencyjnie rozwiązane zostają podproblemy dla obu części X i Y, na skutek czego wyznaczone zostają dwie odległości d oraz e, oznaczające odpowiednio odległość najbliższej pary punktów w zbiorze X oraz Y. Załóżmy, bez straty ogólności, że $d < e$.

Kolejnym krokiem algorytmu jest scalanie podrozwiązań. Oczywistym jest, że para najbliższych punktów dla całego zbioru S nie może być oddalona o więcej niż d. Możliwe są dwa przypadki — albo najbliższa para należy do zbioru X (wtedy jest to wyznaczona para punktów oddalonych o d), albo jeden z punktów należy do zbioru X, a drugi do zbioru Y. W przypadku tej drugiej sytuacji, punkty są oddalone od siebie o odległość mniejszą niż d, więc każdy z nich musi leżeć nie dalej niż o d od prostej l — podzbiory X i Y punktów znajdujących się nie dalej niż o d oznaczamy odpowiednio przez X' i Y' (rysunek 3.12.c).

To, co pozostaje do zrobienia, to wyznaczenie wszystkich par punktów $u \in X'$ oraz $v \in Y'$ takich, że $|u \mapsto v| < d$. Ponieważ odległość dowolnej pary punktów, zarówno w zbiorze X', jak

i Y' wynosi co najmniej d, zatem liczba punktów ze zbiorów X' oraz Y' w dowolnym kwadracie o boku długości d jest ograniczona przez liczbę 5. Aby wyznaczyć wszystkie pożądane pary punktów u i v, wystarczy dla każdego punktu ze zbioru X' sprawdzić 5 najbliższych pod względem współrzędnej y punktów ze zbioru Y'. Można tego dokonać w czasie liniowym ze względu na sumaryczną moc zbiorów X oraz Y, dzięki wcześniejszemu posortowaniu punktów względem współrzędnej y.

Implementacja opisanej wyżej metody jest realizowana przez strukturę `NearestPoints`. Do jej konstruktora przekazywany jest wektor punktów, dla których chcemy znaleźć parę punktów najbliższych. Wynik — najmniejszą odległość — można odczytać z pola **double** `dist`, natomiast para najbliższych punktów jest wskazywana przez wskaźniki — `POINT *p1, *p2`. Implementacja przedstawiona jest na listingu 3.46.

Listing 3.46: Implementacja struktury `NearestPoints`

```
01 struct NearestPoints {
02   vector<POINT*> l;
// Wskaźniki na dwa punkty, stanowiące znalezioną parę najbliższych punktów
03   POINT *p1, *p2;
// Odległość między punktami p1 i p2
04   double dist;
// Funkcja usuwa z listy l wszystkie punkty, których odległość
// od prostej x=p jest większa od odległości między parą aktualnie
// znalezionych najbliższych punktów
05   void Filter(vector<POINT*> &l, double p) {
06     int s = 0;
07     REP(x, SIZE(l))
08       if (sqr(l[x]->x - p) <= dist) l[s++] = l[x];
09     l.resize(s);
10   }
// Funkcja realizuje fazę dziel i zwyciężaj dla zbioru punktów z wektora l
// od pozycji p do k. Wektor ys zawiera punkty
// z przetwarzanego zbioru posortowane w kolejności niemalejących współrzędnych y
11   void Calc(int p, int k, vector<POINT*> &ys) {
// Jeśli zbiór zawiera więcej niż jeden punkt, to następuje faza podziału
12     if (k - p > 1) {
13       vector<POINT*> lp, rp;
// Wyznacz punkt podziału zbioru
14       int c = (k + p - 1) / 2;
// Podziel wektor ys na dwa zawierające odpowiednio punkty
// na lewo oraz na prawo od punktu l[c]
15       FOREACH(it, ys)
16         if (OrdXY(l[c], *it)) rp.PB(*it); else lp.PB(*it);
// Wykonaj fazę podziałów
17       Calc(p, c + 1, lp);
18       Calc(c + 1, k, rp);
// Następuje faza scalania. Najpierw z wektorów l i r usuwane
// są punkty położone zbyt daleko od prostej wyznaczającej podział zbiorów
19       Filter(lp, l[c]->x);
20       Filter(rp, l[c]->x);
21       int p = 0; double k;
```

Listing 3.46: (c.d. listingu z poprzedniej strony)

```
// Następuje faza wyznaczania odległości pomiędzy kolejnymi parami punktów,
// które mogą polepszyć aktualny wynik
22          FOREACH(it, lp) {
23           while (p < SIZE(rp) - 1 && rp[p + 1]->y < (*it)->y) p++;
24             FOR(x, max(0, p - 2), min(SIZE(rp) - 1, p + 1))
// Jeśli odległość między parą przetwarzanych punktów jest mniejsza od aktualnego
wyniku,
// to zaktualizuj wynik
25                if (dist > (k = sqr((*it)->x - rp[x]->x) +
26                   sqr((*it)->y - rp[x]->y))) {
27                   dist = k;
28                   p1 = *it;
29                   p2 = rp[x];
30                }
31          }
32       }
33    }
// Konstruktor struktury NearestPoints wyznaczający parę najbliższych punktów
34    NearestPoints(vector<POINT> &p) {
// Wypełnij wektor l wskaźnikami do punktów z wektora p
// oraz posortuj te wskaźniki w kolejności niemalejących współrzędnych x
35       FOREACH(it, p) l.PB(&(*it));
36       sort(ALL(l), OrdXY);
// Jeśli w zbiorze istnieją dwa punkty o tych samych współrzędnych,
// to punkty te są poszukiwanym wynikiem
37       FOR(x, 1, SIZE(l) - 1)
38        if (l[x - 1]->x == l[x]->x && l[x - 1]->y == l[x]->y) {
39             dist = 0;
40             p1 = l[x - 1]; p2 = l[x]; return;
41        }
42       dist = double(INF) * double(INF);
// Skonstruuj kopię wektora wskaźników do punktów i posortuj go w kolejności
// niemalejących współrzędnych y
43       vector<POINT*> v = l;
44       sort(ALL(v), OrdYX);
// Wykonaj fazę dziel i rządź dla wszystkich punktów ze zbioru
45       Calc(0, SIZE(l), v);
46       dist = sqrt(dist);
47    }
48 };
```

Listing 3.47: Przykład wykorzystania struktury `NearestPoints` na zbiorze punktów z rysunku 3.12

```
Wyznaczona odleglosc: 4.12311
Znaleziona para najblizszych punktow:
(-7, 5) (-3, 4)
```

Listing 3.48: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 3.47. Pełny kod źródłowy programu znajduje się w pliku `nearestpoints_str.cpp`

```
01 int main() {
02    int n;
03    vector<POINT> l;
04    POINT p;
// Wczytaj liczbę punktów
05    cin >> n;
// Wczytaj kolejne punkty
06    REP(x, n) {
07       cin >> p.x >> p.y;
08       l.PB(p);
09    }
// Wyznacz parę najbliższych punktów oraz wypisz wynik
10    NearestPoints str(l);
11    cout << "Wyznaczona odleglosc: " << str.dist << endl;
12    cout << "Znaleziona para najblizszych punktow:" << endl;
13    cout << *(str.p1) << " " << *(str.p2) << endl;
14    return 0;
15 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10245	acm.sgu.ru - zadanie 120 acm.uva.es - zadanie 10750	

Rozdział 4

Kombinatoryka

Literatura
[KDP] - 1
[SZP] - 1.2.5
[ASP] - 3

Wiele zadań, które zawodnik musi rozwiązać podczas zawodów, należy do klasy problemów NP-trudnych. Stosunkowo dużo tego typu zadań pojawia się na konkursach ACM czy TopCoder. Rozpoznanie zadania tego typu jest zazwyczaj stosunkowo łatwe — wszystkie one charakteryzują się bardzo ścisłymi limitami na wielkość danych wejściowych (umiejętność szacowania złożoności wymaganego algorytmu na podstawie ograniczeń na wielkość danych wejściowych jest bardzo istotna). Jak wiadomo, nie są znane algorytmy pozwalające na rozwiązywanie zadań NP-trudnych w czasie wielomianowym — często stosowane techniki bazują na programowaniu dynamicznym, które zasadniczo polega na wyznaczaniu wyników dla pewnej klasy podproblemów, na podstawie których można obliczyć wynik dla postawionego problemu. Liczba podproblemów, które należy rozpatrzeć w przypadku zadań NP-trudnych jest wykładnicza, co nie tylko wiąże się z długim czasem ich generowania, ale również wymaga dużej ilości pamięci potrzebnej na spamiętywanie wyników częściowych. Programowanie dynamiczne w przypadku wielu zadań nie jest również proste z punktu widzenia implementacyjnego.

Czasem rozsądniejszym podejściem okazuje się wygenerowanie wszystkich możliwych rozwiązań do postawionego problemu, a następnie wybranie spośród nich najlepszego. Rozwiązaniem może być przykładowo odpowiednie uporządkowanie pewnych obiektów (wyznaczany porządek można reprezentować przy użyciu permutacji), czy też wybranie pewnego ich podzbioru.

W przypadku wielu zadań NP-trudnych okazuje się, że umiejętność efektywnego generowania rozmaitych obiektów kombinatorycznych jest podstawą do szybkiego rozwiązania zadania. W aktualnym rozdziale przedstawimy kilka algorytmów pozwalających na generowanie wszystkich permutacji oraz podzbiorów danego zbioru, jak również podziałów liczby.

Świetnym źródłem informacji, na bazie którego powstała zawartość tego rozdziału jest książka Witolda Lipskiego „Kombinatoryka dla programistów", do której lektury gorąco zachęcamy.

4.1. Permutacje w kolejności antyleksykograficznej

Pierwszą prezentowaną funkcją jest **void** `PermAntyLex::Gen (VI&, void (*)(VI&))`, której kod źródłowy przedstawiony jest na listingu 4.1. Funkcja ta przyjmuje jako argumenty wektor liczb v oraz wskaźnik na funkcję f. Jej zadaniem jest generowanie wszystkich permutacji zbioru v, oraz dla każdej wygenerowanej permutacji wywołanie funkcji f. Sposób użycia tej funkcji w praktyce, jak i przykładowa kolejność wyznaczanych permutacji przedstawione

są na listingu 4.2. Specyficzny porządek generowania permutacji okazuje się pożyteczny w niektórych zadaniach, jednak główną zaletą tej funkcji jest jej prostota.

Czas wyznaczenia wszystkich permutacji zbioru n-elementowego to $O(n!)$, co daje optymalny algorytm, gdyż średni czas przypadający na wygenerowanie pojedynczej permutacji jest stały.

Literatura
[KDP] - 1.4
[SZP] - 1.2.5

Listing 4.1: Implementacja funkcji **void** `PermAntyLex:Gen(VI&, void (*)(VI&))`

```
01 namespace PermAntyLex {
// Wskaźnik na wektor liczb reprezentujących generowaną permutację
02   VI *V;
// Wskaźnik na funkcję, która jest wywoływana dla każdej wygenerowanej
// permutacji
03   void (*fun) (VI &);
// Funkcja rekurencyjna, wyznaczająca wszystkie m-elementowe permutacje
04   void Perm(int m) {
05     if (!m) fun(*V);
06     else FOR(i, 0, m) {
07       Perm(m - 1);
08       if (i < m) {
// Zamiana miejscami elementu i-tego oraz m-1-szego
09         swap((*V)[i], (*V)[m]);
// Odwrócenie kolejności wszystkich elementów na przedziale [0..m-1]. W
// miejscu tym wykorzystywana jest niejawna informacja na temat budowy
// wektorów, gdyby elementy nie znajdowały się w pamięci w postaci
// ciągłej, nie wolno byłoby skorzystać z funkcji reverse
10         reverse(&(*V)[0], &(*V)[m]);
11       }
12     }
13   }
// Właściwa funkcja wywoływana z zewnątrz przestrzeni nazw PermAntyLex
14   void Gen(VI & v, void (*f) (VI &)) {
15     V = &v;
16     fun = f;
17     Perm(SIZE(v) - 1);
18   }
19 };
```

Listing 4.2: Ciąg permutacji wygenerowany przez funkcję **void** `PermAntyLex:Gen(VI&, void (*)(VI&))` dla zbioru liczb $\{1, 2, 3, 4\}$

```
1 2 3 4     2 1 3 4     1 3 2 4     3 1 2 4     2 3 1 4     3 2 1 4
1 2 4 3     2 1 4 3     1 4 2 3     4 1 2 3     2 4 1 3     4 2 1 3
1 3 4 2     3 1 4 2     1 4 3 2     4 1 3 2     3 4 1 2     4 3 1 2
2 3 4 1     3 2 4 1     2 4 3 1     4 2 3 1     3 4 2 1     4 3 2 1
```

Listing 4.3: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.2. Pełny kod źródłowy programu znajduje się w pliku `permantylex.cpp`

```
// Funkcja wywoływana dla każdej wygenerowanej permutacji
01 void Display(VI & v) {
02    static int calc = 0;
03    if (!(calc++ % 6)) cout << endl;
// Wypisz elementy permutacji
04    FOREACH(it, v) cout << *it << " ";
05    cout << "\t";
06 }
07 int main() {
08    VI p;
// Stwórz wektor liczb {1,2,3,4}
09    FOR(x, 1, 4) p.PB(x);
// Wygeneruj permutacje zbioru {1,2,3,4} w kolejności antyleksykograficznej
10    PermAntyLex::Gen(p, &Display);
11    return 0;
12 }
```

4.2. Permutacje — minimalna liczba transpozycji

Definicja 4.2.1 Transpozycja *jest to zamiana miejscami dwóch elementów permutacji — przykładowo, permutacjami uzyskanymi poprzez zastosowanie pojedynczej transpozycji to* $\{1, 2, 3, 4\}$ *są* $\{2, 1, 3, 4\}$ *oraz* $\{1, 4, 3, 2\}$*, ale nie jest nią* $\{2, 3, 1, 4\}$.

Jak już zauważyliśmy wcześniej, nie da się szybciej wygenerować wszystkich permutacji zbioru n-elementowego niż w czasie $O(n!)$. Nie oznacza to jednak, że rozwiązanie zadania bazującego na przeanalizowaniu wszystkich permutacji w celu znalezienia optymalnego rozwiązania działa w takim czasie. Dla każdej permutacji bowiem muszą zostać wykonane pewne obliczenia, zależne od rozwiązywanego problemu. Jedną z możliwości wykonywania tych obliczeń jest dokonywanie ich niezależnie dla każdej wyznaczonej permutacji. Podejście takie jest najprostsze z możliwych, jednak nie uwzględnia zależności, które mogą występować pomiędzy różnymi rozwiązaniami, a co za tym idzie — podczas sprawdzania kolejnych permutacji, algorytm może wykonywać pewną nadmierną pracę. Uwzględnianie zależności między rozwiązaniami wymaga jednak pewnej wiedzy na temat kolejności generowania permutacji. W zależności od tego może okazać się, że wykorzystanie informacji z poprzedniego rozwiązania, w celu wyznaczenia wyniku dla aktualnego, nie jest proste, o ile w ogóle możliwe. Przedstawiona w aktualnym rozdziale funkcja **void** `PermMinTr::Gen(VI&, void (*)(VI&))` służy do generowania wszystkich permutacji danego zbioru w kolejności minimalnych transpozycji. Przykład wygenerowanej listy permutacji przedstawiony jest na listingu 4.5.

Kolejne dwie permutacje powstają poprzez zamianę miejscami tylko dwóch elementów — podejście takie w wielu zadaniach bardzo ułatwia wyznaczanie wyników dla kolejnych rozwiązań z wykorzystaniem poprzednich. Parametrami funkcji są odpowiednio wektor liczb reprezentujący permutowany zbiór oraz wskaźnik na funkcję wywoływaną dla każdej wyznaczonej permutacji. Czas działania algorytmu to $O(n!)$. Implementacja przedstawiona jest na listingu 4.4.

Listing 4.4: Implementacja funkcji **void** `PermMinTr::Gen(VI&, void (*)(VI&))`

```
01 namespace PermMinTr {
// Wskaźnik na wektor liczb reprezentujących generowaną permutację
02   VI* V;
// Wskaźnik na funkcję, która jest wywoływana dla każdej wygenerowanej permutacji
03   void (*fun)(VI&);
// Funkcja rekurencyjna, wyznaczająca wszystkie m-elementowe permutacje
04   void Perm(int m){
05     if(m == 1) fun(*V); else
06     REP(i, m) {
07       Perm(m - 1);
08       if(i < m - 1) swap((*V)[(!(m & 1) && m > 2) ?
09         (i < m - 1) ? i : m - 3 : m - 2], (*V)[m - 1]);
10     }
11   }
// Właściwa funkcja wywoływana z zewnątrz przestrzeni nazw PermAntyLex
12   void Gen(VI& v, void (*f)(VI&)) {
13     V = &v;
14     fun = f;
15     Perm(SIZE(v));
16   }
17 };
```

Listing 4.5: Przykład działania funkcji **void** `PermMinTr::Gen(VI&, void (*)(VI&))` dla zbioru $\{1, 2, 3, 4\}$

```
1 2 3 4     2 1 3 4     2 3 1 4     3 2 1 4     3 1 2 4     1 3 2 4
4 3 2 1     3 4 2 1     3 2 4 1     2 3 4 1     2 4 3 1     4 2 3 1
4 1 3 2     1 4 3 2     1 3 4 2     3 1 4 2     3 4 1 2     4 3 1 2
4 3 2 1     3 4 2 1     3 2 4 1     2 3 4 1     2 4 3 1     4 2 3 1
```

Listing 4.6: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.5. Pełny kod źródłowy programu znajduje się w pliku `permmintr.cpp`

```
// Funkcja wywoływana dla każdej wygenerowanej permutacji
01 void Display(VI & v) {
02   static int calc = 0;
03   if (!(calc++ % 6)) cout << "\n";
// Wypisz elementy permutacji
04   FOREACH(it, v) cout << *it << " ";
05   cout << "\t";
06 }
07 int main() {
08   VI p;
// Stwórz wektor liczb {1,2,3,4}
09   FOR(x, 1, 4) p.PB(x);
// Wygeneruj permutacje zbioru {1,2,3,4} z użyciem minimalnej liczby
```

Listing 4.6: (c.d. listingu z poprzedniej strony)

```
// transpozycji
10    PermMinTr::Gen(p, &Display);
11    return 0;
12 }
```

4.3. Permutacje — minimalna liczba transpozycji sąsiednich

Definicja 4.3.1 Transpozycja sąsiednia *jest to taka transpozycja, która zamienia miejscami dwa sąsiednie elementy — przykładowo, permutacjami uzyskanymi poprzez zastosowanie pojedynczej transpozycji sąsiedniej do permutacji* $\{1,2,3,4\}$ *są* $\{2,1,3,4\}$ *oraz* $\{1,2,4,3\}$*, ale nie są nimi* $\{2,3,1,4\}$ *oraz* $\{4,2,3,1\}$.

W niektórych zadaniach generowanie permutacji w kolejności minimalnych transpozycji może okazać się niewystarczające — zależności między poszczególnymi rozwiązaniami są na tyle skomplikowane, że proces aktualizacji wyniku dla konkretnego rozwiązania nie może zostać zrealizowany w zadowalającym czasie na podstawie wyniku dla innego rozwiązania, różniącego się pojedynczą transpozycją. W takich sytuacjach pożądanym sposobem generowania kolejnych permutacji może okazać się metoda wyznaczania permutacji z użyciem minimalnej liczby transpozycji sąsiednich. Funkcja **void** `PermMinTrAdj(VI&, void (*)(VI&, int))`, której kod źródłowy przedstawiony jest na listingu 4.7, realizuje wyżej omówioną metodę wyznaczania permutacji. Jej parametrami są odpowiednio wektor liczb oraz funkcja f wywoływana dla każdej wyznaczonej permutacji. Funkcja f otrzymuje dwa parametry — pierwszym jest wektor liczb reprezentujący permutację, drugi natomiast to liczba k wyznaczająca parę elementów (k oraz $k+1$), które zostały zamienione miejscami w celu uzyskania nowej permutacji ($k=-1$ w przypadku pierwszej permutacji). Czas działania funkcji to $O(n!)$. Przykład jej użycia przedstawiony jest na listingu 4.8.

Listing 4.7: Implementacja funkcji **void** `PermMinTrAdj(VI&, void (*)(VI&, int))`

```
// Funkcja generuje wszystkie permutacje zbioru p z użyciem minimalnej liczby
// transpozycji sąsiednich
01 void PermMinTrAdj(VI & p, void (*fun) (VI &, int)) {
02    int x, k, i = 0, s = SIZE(p);
03    VI c(s, 1);
04    vector<bool> pr(s, 1);
05    c[s - 1] = 0;
06    fun(p, -1);
07    while (i < s - 1) {
08       i = x = 0;
09       while (c[i] == s - i) {
10          if (pr[i] = !pr[i]) x++;
11          c[i++] = 1;
12       }
13       if (i < s - 1) {
14          k = pr[i] ? c[i] + x : s - i - c[i] + x;
15          swap(p[k - 1], p[k]);
16          fun(p, k - 1);
```

Listing 4.7: (c.d. listingu z poprzedniej strony)

```
17          c[i]++;
18        }
19    }
20 }
```

Listing 4.8: Wynik wygenerowany przez funkcję **void** `PermMinTrAdj(VI&, void (*)(VI&, int))` dla zbioru liczb $\{1, 2, 3\}$.

```
Permutacja: 1 2 3
Zamiana elementow 0 i 1
Permutacja: 2 1 3
Zamiana elementow 1 i 2
Permutacja: 2 3 1
Zamiana elementow 0 i 1
Permutacja: 3 2 1
Zamiana elementow 1 i 2
Permutacja: 3 1 2
Zamiana elementow 0 i 1
Permutacja: 1 3 2
```

Listing 4.9: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.8. Pełny kod źródłowy programu znajduje się w pliku `permmintradj.cpp`

```
// Funkcja wywoływana dla każdej wygenerowanej permutacji
01 void Display(VI & v, int k) {
02    if (k != -1) cout << "Zamiana elementow " << k << " i " << k + 1 << endl;
// Wypisz elementy permutacji
03    cout << "Permutacja: ";
04    FOREACH(it, v) cout << *it << " ";
05    cout << endl;
06 }
07 int main() {
08    VI p;
// Stwórz wektor liczb {1,2,3}
09    FOR(x, 1, 3) p.PB(x);
// Wygeneruj permutacje zbioru {1,2,3} z użyciem minimalnej liczby transpozycji
// sąsiednich
10    PermMinTrAdj(p, &Display);
11    return 0;
12 }
```

Zadanie: Liczby permutacyjnie-pierwsze

Pochodzenie: Eliminacje do konkursu ACM ICPC — Dhaka Regional 2004

Rozwiązanie: `permprime.cpp`

Permutacje ciągu cyfr $0-9$ posiadają bardzo ciekawe własności — różnica dowolnych dwóch permutacji tej samej sekwencji cyfr jest zawsze podzielna przez 9. Jest to dość interesujące

zjawisko. Przykładowo:

$$|458967 - 456879| = 2088 = 9 * 232$$

Zadaniem nie będzie jednak udowodnienie tego faktu (gdyż jest to zbyt proste) lecz przeanalizowanie nieco innej własności. Istnieją pewne liczby, które pomniejszone o pewną permutację swoich cyfr są postaci $9p$, gdzie p to liczba pierwsza mniejsza od 1111111. Liczby takie nazywane są permutacyjnie-pierwszymi. Dla przykładu: $92-29 = 63 = 9*7$, a ponieważ 7 jest liczbą pierwszą, zatem 92 jest liczbą permutacyjnie-pierwszą.

Zadanie

Napisz program, który:

- wczyta listę przedziałów liczb naturalnych,
- dla każdego z przedziałów wyznaczy ilość liczb permutacyjnie-pierwszych.
- wypisze wynik.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą T $(0 < T < 51)$ oznaczającą liczbę testów. Następnych T wierszy zawiera po dwie liczby p oraz q $(0 < p \leqslant q \leqslant 99\,999\,999$, $q - p \leqslant 1\,000)$.

Wyjście

Dla każdego testu należy wypisać jeden wiersz zawierający dokładnie jedną liczbę całkowitą, oznaczającą liczbę liczb w przedziale $[a, b]$, które są permutacyjnie-pierwsze.

Przykład

Dla następującego wejścia:

```
2
1  10
2  20
```

Poprawnym rozwiązaniem jest:

```
0
5
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10911	acm.uva.es - zadanie 216	acm.sgu.ru - zadanie 224
acm.uva.es - zadanie 10063	spoj.sphere.pl - zadanie 399	acm.uva.es - zadanie 10012
acm.uva.es - zadanie 146	acm.sgu.ru - zadanie 222	acm.uva.es - zadanie 10252
acm.uva.es - zadanie 10098	acm.uva.es - zadanie 10460	acm.uva.es - zadanie 195
spoj.sphere.pl - zadanie 379	acm.uva.es - zadanie 10475	

4.4. Wszystkie podzbiory zbioru

Literatura
[KDP] - 1.5

Rozwiązania niektórych problemów nie wyrażają się w postaci permutacji elementów pewnego zbioru, co wymusza potrzebę poszukiwania innych sposobów reprezentacji. Jedną z nich jest użycie podzbiorów danego zbioru — w takich sytuacjach przydatna może się okazać, podobnie jak to było w przypadku permutacji, efektywna metoda generująca wszystkie 2^n podzbiorów zbioru n-elementowego.

Prezentowana w tym rozdziale funkcja **void** `SubsetMin(`**int, void** `(*)(vector<`**bool**`>&,` **int**`))`, której implementacja przedstawiona jest na listingu 4.10, przyjmuje jako parametry liczbę naturalną n, określającą liczbę elementów w rozpatrywanym zbiorze oraz wskaźnik na funkcję f, która wywoływana jest dla każdego wyznaczonego podzbioru. Funkcji f przekazywane są dwa parametry. Pierwszym z nich jest wektor n zmiennych logicznych reprezentujących podzbiór zbioru — k-ty element przyjmuje wartość 1, jeśli k-ty element zbioru należy do wyznaczonego podzbioru. Drugi parametr reprezentuje numer elementu, który został dodany lub usunięty w procesie generowania kolejnego podzbioru. Każde dwa kolejne podzbiory wyznaczane przez funkcję różnią się dokładnie jednym elementem (w każdym kroku jeden element zostaje dodany lub usunięty z generowanego podzbioru). Czas działania funkcji jest liniowy ze względu na liczbę wyznaczanych podzbiorów i wynosi $O(2^n)$.

Listing 4.10: Implementacja funkcji **void** `SubsetMin(VI&, void (*)(vector<`**bool**`>&,`**int**`))`

```
// Funkcja generująca wszystkie podzbiory zbioru n-elementowego
01 void SubsetMin(int n, void (*fun)(vector<bool>&, int)) {
02    vector<bool> B(n, 0);
03    fun(B, -1);
04    for(int i = 0, p = 0, j; p < n;) {
05       for(p = 0, j = ++i; j & 1; p++) j >>= 1;
06       if(p < n) {
// Zmień przynależność do podzbioru elementu p
07          B[p] =! B[p];
08          fun(B, p);
09       }
10 }
11 }
```

Listing 4.11: Przykład działania funkcji **void** `SubsetMin(VI&, void (*)(vector<`**bool**`>&,`**int**`))` na przykładzie 3-elementowego zbioru

```
Podzbior: 0 0 0
Do zbioru zostal dodany element 1
Podzbior: 0 1 0
Do zbioru zostal dodany element 0
Podzbior: 1 1 0
Do zbioru zostal dodany element 2
Podzbior: 1 1 1
Ze zbioru zostal usuniety element 0
Podzbior: 0 1 1
Ze zbioru zostal usuniety element 1
Podzbior: 0 0 1
```

Listing 4.11: (c.d. listingu z poprzedniej strony)

```
Do zbioru zostal dodany element 0
Podzbior: 1 0 1
```

Listing 4.12: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.12. Pełny kod źródłowy programu znajduje się w pliku `subsetmin.cpp`

```
// Funkcja wywoływana dla każdego wygenerowanego podzbioru
01 void Display(vector<bool> &v, int k) {
02    if (k != -1) {
03       if (v[k]) cout << "Do zbioru zostal dodany element " << k << endl;
04       else cout << "Ze zbioru zostal usuniety element " << k << endl;
05    }
06    cout << "Podzbior: ";
07    FOREACH(it, v) cout << *it << " ";
08    cout << endl;
09 }
10 int main() {
// Wyznacz wszystkie 3-elementowe podzbiory
11    SubsetMin(3, &Display);
12    return 0;
13 }
```

Ćwiczenia

Proste	Średnie	Trudne
spoj.sphere.pl - zadanie 147	acm.sgu.ru - zadanie 249	acm.uva.es - zadanie 811

4.5. Podzbiory k-elementowe w kolejności leksykograficznej

Literatura
[KDP] - 1.7

Istnieje grupa zadań, w których rozwiązanie można reprezentować przy użyciu podzbioru, ale do rozwiązania nie są potrzebne wszystkie podzbiory, lecz tylko te o odpowiedniej liczności. W takich sytuacjach przydatny jest mechanizm pozwalający na wyznaczanie wszystkich podzbiorów zbioru n-elementowego zawierających dokładnie k elementów. Zadanie takie okazuje się nie tylko możliwe do zrealizowania w czasie liniowym ze względu na liczbę wyznaczanych podzbiorów, ale również proste w implementacji.

Funkcja **void** `SubsetKLex(`**int, int, void** `(*)(VI&))` przedstawiona na listingu 4.13 pozwala na wyznaczanie wszystkich podzbiorów k-elementowych w kolejności leksykograficznej. Jako parametry funkcja ta przyjmuje liczbę elementów w rozpatrywanym zbiorze n, wielkość generowanych podzbiorów k oraz wskaźnik na funkcję, której jako parametr będą przekazywane kolejno generowane podzbiory. Podzbiór reprezentowany jest w postaci listy numerów elementów do niego należących. Elementy numerowane są od 0 do $n-1$. Przykład użycia tej funkcji został przedstawiony na listingu 4.15.

Listing 4.13: Implementacja funkcji **void** `SubsetKLex(`**int, int, void** `(*)(VI&))`

```
// Funkcja wyznacza wszystkie podzbiory k-elementowe n-elementowego zbioru
01 void SubsetKLex(int k, int n, void (*fun) (VI &)) {
02    int i, p = k;
03    VI A(k);
04    REP(x, k) A[x] = x;
05    while (p) {
06       fun(A);
07       A[k - 1] == n - 1 ? p-- : p = k;
08       if (p) FORD(i, k, p) A[i - 1] = A[p - 1] + i - p + 1;
09    }
10 }
```

Listing 4.14: Przykład wykorzystania funkcji **void** `SubsetKLex(`**int, int, void** `(*)(VI&))` do wyznaczenia wszystkich podzbiorów 4-elementowych 7-elementowego zbioru

```
0 1 2 3    0 1 2 4    0 1 2 5    0 1 2 6    0 1 3 4
0 1 3 5    0 1 3 6    0 1 4 5    0 1 4 6    0 1 5 6    0 2 3 4
0 2 3 5    0 2 3 6    0 2 4 5    0 2 4 6    0 2 5 6    0 3 4 5
0 3 4 6    0 3 5 6    0 4 5 6    1 2 3 4    1 2 3 5    1 2 3 6
1 2 4 5    1 2 4 6    1 2 5 6    1 3 4 5    1 3 4 6    1 3 5 6
1 4 5 6    2 3 4 5    2 3 4 6    2 3 5 6    2 4 5 6    3 4 5 6
```

Listing 4.15: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.15. Pełny kod źródłowy programu znajduje się w pliku `subsetklex.cpp`

```
// Funkcja wywoływana dla każdego wyznaczonego podzbioru
01 void Display(VI & v) {
02    static int calc = 0;
03    if (!(++calc % 6)) cout << "\n";
// Wypisz elementy podzbioru
04    FOREACH(it, v) cout << *it << " ";
05    cout << "\t";
06 }
07 int main() {
// Wygeneruj wszystkie 4-elementowe podzbiory 7-elementowego zbioru
08    SubsetKLex(4, 7, &Display);
09    return 0;
10 }
```

4.6. Podziały zbioru z użyciem minimalnej liczby zmian

Podział zbioru polega na przyporządkowaniu każdego elementu tego zbioru do dokładnie jednego podzbioru. Przykładami podziałów zbioru 5-elementowego $\{1,2,3,4,5\}$ są $\{1,5\}$ $\{2,3,4\}$ oraz $\{1\}$ $\{2\}$ $\{3,4\}$ $\{5\}$. Tak jak i w przypadku permutacji, wyznaczanie wyniku dla konkretnego podziału można dokonać na podstawie wyniku wyznaczonego dla innego —

podobnego podziału, zatem metoda wyznaczania podziałów jest dość istotna. Przedstawiony tu algorytm, realizowany przez funkcję **void** `SetPartition(`**int, void** `(*)(VI&))` z listingu 4.16, wyznacza podziały zbioru w sekwencji minimalnych zmian — kolejne dwa podziały różnią się przynależnością do różnych podzbiorów tylko jednego elementu. Funkcja jako parametry przyjmuje liczbę n, określającą moc dzielonego zbioru oraz wskaźnik na funkcję f, której przekazywane są wszystkie wyznaczane podziały. Podział zbioru reprezentowany jest przez wektor n liczb, gdzie k-ta liczba reprezentuje numer podzbioru, do którego został przydzielony k-ty element. Przykład działania funkcji został przedstawiony na listingu 4.17.

Listing 4.16: Implementacja funkcji **void** `SetPartition(`**int, void** `(*)(VI&))`

```
// Funkcja wyznacza wszystkie podziały n-elementowego zbioru
01 void SetPartition(int n, void (*fun)(VI&)) {
02    VI B(n, 1), N(n + 1), P(n + 1);
03    vector<bool> Pr(n + 1, 1);
04    N[1] = 0;
05    fun(B);
06    for(int i, k, j = n; j > 1;) {
07       k = B[j - 1];
08       if(Pr[j]) {
09          if(!N[k]) P[N[k] = j + (N[j] = 0)] = k;
10          if(N[k] > j) N[P[j] = k] = P[N[j] = N[k]] = j;
// W tym miejscu następuje przydzielenie elementu j do zbioru o numerze N[k]
11          B[j - 1] = N[k];
12       } else {
// W tym miejscu następuje przydzielenie elementu j do zbioru o numerze P[k]
13          B[j - 1] = P[k];
14          if(k == j) N[k] ? P[N[P[k]] = N[k]] = P[k] : N[P[k]] = 0;
15       }
16       fun(B);
17       j = n;
18       while(j > 1 && ((Pr[j] && (B[j - 1] == j))
19          || (!Pr[j] && (B[j - 1] == 1)))) Pr[j--] = !Pr[j];
20    }
21 }
```

Listing 4.17: Przykład działania funkcji **void** `SetPartition(`**int, void** `(*)(VI&))` dla 4-elementowego zbioru

```
1 1 1 1      1 1 1 4      1 1 3 4      1 1 3 3      1 1 3 1      1 2 3 1
1 2 3 2      1 2 3 3      1 2 3 4      1 2 2 4      1 2 2 2      1 2 2 1
1 2 1 1      1 2 1 2      1 2 1 4
```

Listing 4.18: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.17. Pełny kod źródłowy programu znajduje się w pliku `setpartition.cpp`

```
// Funkcja wywoływana dla każdego wyznaczonego podziału zbioru
01 void Display(VI & v) {
02    static int calc = 0;
```

Listing 4.18: (c.d. listingu z poprzedniej strony)

```
03    if (!(calc++ % 6)) cout << endl;
// Wypisz wyznaczony podział zbioru
04    FOREACH(it, v) cout << *it << " ";
05    cout << "\t";
06 }
07 int main() {
// Wyznacz wszystkie podziały 4-elementowego zbioru
08    SetPartition(4, &Display);
09    return 0;
10 }
```

4.7. Podziały liczby w kolejności antyleksykograficznej

Literatura
[KDP] - 1.11

Ostatnim z serii algorytmów kombinatorycznych jest funkcja, służąca do generowania wszystkich podziałów zadanej liczby w kolejności antyleksykograficznej. Podziałem liczby naturalnej nazywamy zaprezentowanie jej w postaci sumy liczb naturalnych dodatnich. Dla przykładu — liczbę 10 można przedstawić na wiele sposobów, między innymi jako $5 + 3 + 2$, czy jako $5 + 5$. Dwa podziały liczby są takie same, jeśli w ich skład wchodzą te same liczby (nie jest istotna kolejność). Generowanie podziałów liczb okazuje się potrzebne choćby podczas rozwiązywania problemów związanych z wydawaniem reszty.

Prezentowana na listingu 4.19 funkcja **void** `NumPart(`**int, void** `(*)(VI&, VI&))` przyjmuje jako parametry liczbę naturalną n, która ma zostać poddana podziałom oraz wskaźnik do funkcji f, wywoływanej dla każdego wyznaczonego podziału. Funkcja f jako parametry otrzymuje parę równolicznych wektorów s oraz r — k-ta para elementów z tych wektorów (s_k, r_k) reprezentuje k-ty składnik sumy — s_k liczb r_k (szczegółowy sposób użycia został przedstawiony na listingu 4.21).

Listing 4.19: Implementacja funkcji **void** `NumPart(`**int, void** `(*)(VI&, VI&))`

```
// Funkcja generuje wszystkie podziały liczby n
01 void NumPart(int n, void (*fun) (VI &, VI &, int)) {
02    VI S(n + 1), R(n + 1);
03    int d, sum, l;
04    S[0] = n;
05    R[0] = d = 1;
06    while (1) {
07       int summ = 0, x, s;
08       fun(R, S, d);
09       if (S[0] == 1) break;
10       sum = 0;
11       if (S[d - 1] == 1) sum += R[--d];
12       sum += S[d - 1];
13       R[d - 1]--;
14       l = S[d - 1] - 1;
15       if (R[d - 1]) d++;
16       S[d - 1] = l;
```

Listing 4.19: (c.d. listingu z poprzedniej strony)

```
17        R[d - 1] = sum / l;
18        l = sum % l;
19        if (l) {
20           S[d] = l;
21           R[d++] = 1;
22        }
23     }
24 }
```

Listing 4.20: Przykład zastosowania funkcji **void** `NumPart(`**int, void** `(*)(VI&, VI&))` do wyznaczenia podziałów liczby 5

```
5
4 + 1
3 + 2
3 + 1 + 1
2 + 2 + 1
2 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1
```

Listing 4.21: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 4.21. Pełny kod źródłowy programu znajduje się w pliku `numpart.cpp`

```
// Funkcja wywoływana dla każdego wyznaczonego podziału liczby
01 void Display(VI & s, VI & r, int n) {
02    bool ch = 0;
// Wypisz kolejne elementy podziału
03    REP(x, n) REP(y, s[x]) {
04       if (ch) cout << " + ";
05       cout << r[x];
06       ch = 1;
07    }
08    cout << endl;
09 }
10 int main() {
// Wygeneruj wszystkie podziały liczby 5
11    NumPart(5, &Display);
12    return 0;
13 }
```

Rozdział 5

Teoria liczb

Literatura
[SZP] - 4

W tym rozdziale poruszymy problematykę związaną z teorią liczb. Przedstawimy implementacje takich algorytmów, jak wyznaczanie największego wspólnego dzielnika, metodę szybkiego potęgowania, czy wyznaczanie odwrotności modularnej. Zaprezentowane zostaną również metody testowania pierwszości liczb oraz implementacja arytmetyki wielkich liczb, pozwalająca na wykonywanie operacji zarówno na liczbach całkowitych niemieszczących się w ograniczeniach standardowych typów zmiennych, jak również liczb wymiernych, reprezentowanych w postaci ułamka nieskracalnego $\frac{p}{q}$.

5.1. Współczynnik dwumianowy Newtona

Literatura
[MK] - 5

Każdy zapewne zna definicję współczynniku dwumianowego Newtona — $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Istnieje wiele zadań, w których należy wyznaczyć liczbę sposobów wyboru podzbioru jakiegoś zbioru, spełniającego pewne dodatkowe założenia. W zadaniach tego typu zawsze pojawia się w którymś momencie współczynnik dwumianowy Newtona. Wyznaczenie wartości $\binom{n}{k}$ jest proste — wystarczy policzyć wartość liczby $n!$, a następnie podzielić ją przez $k!(n-k)!$. W tym miejscu jednak pojawia się pewien problem. W zadaniach często powiedziane jest, że obliczany wynik mieści się w pewnym standardowym typie zmiennych. Nie gwarantuje to jednak, że wszystkie wykonywane pośrednie obliczenia nie spowodują przepełnienia wartości zmiennej. Taka sytuacja może mieć miejsce podczas wyznaczania wartości współczynnika dwumianowego Newtona — wartość n! może być istotnie większa od wyznaczanej wartości $\binom{n}{k}$. Jednym z rozwiązań tego problemu jest zastosowanie arytmetyki wielkich liczb, ale nie jest to w wielu przypadkach najlepsze rozwiązanie, ze względu na efektywność oraz złożoność implementacji. Zmiana sposobu wyliczania wyniku może okazać się znacznie lepszym pomysłem — tak jest w przypadku współczynnika dwumianowego Newtona.

Sposób wyliczania współczynnika dwumianowego Newtona, który nie powoduje przepełnienia zmiennych, o ile sama wartość $\binom{n}{k}$ mieści się w arytmetyce, polega na wyznaczeniu rozkładu na liczby pierwsze poszczególnych liczb $n!$, $k!$ oraz $(n-k)!$, a następnie wykonaniu dzielenia poprzez skracanie czynników uzyskanych rozkładów. Implementacja takiego algorytmu została przedstawiona na listingu 5.1. Funkcja `LL Binom(`**int, int**`)` przyjmuje jako parametry liczby n oraz k, a zwraca jako wynik wartość $\binom{n}{k}$.

Listing 5.1: Implementacja funkcji `LL Binom(int, int)`

```
// Funkcja wyznacza wartość dwumianu Newtona
LL Binom(int n, int k) {
#define Mark(x, y) for(int w = x, t = 2; w > 1; t++) \
while(!(w % t)) {w /= t; p[t] += y;}
   if (n < k || n < 0) return 0;
   int p[n + 1];
   REP(x, n + 1) p[x] = 0;
// Wyznacz wartość liczby n!/(n-k)!=(n-k+1)*...*n w postaci rozkładu
// na liczby pierwsze
   FOR(x, n - k + 1, n) Mark(x, 1);
// Podziel liczbę, której rozkład znajduje się w tablicy
// p przez k!
   FOR(x, 1, k) Mark(x, -1);
// Wylicz wartość współczynnika dwumianowego na podstawie
// jej rozkładu na liczby pierwsze i zwróć wynik
   LL r = 1;
   FOR(x, 1, n) while(p[x]--) r *= x;
   return r;
}
```

Listing 5.2: Przykład działania funkcji `LL Binom(int, int)`

```
Binom(40,20) = 137846528820
Binom(100000,3) = 166661666700000
Binom(10,23) = 0
```

Listing 5.3: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.2. Pełny kod źródłowy programu znajduje się w pliku `binom.cpp`

```
int main() {
   int n, k;
// Dla wszystkich par liczb wyznacz wartość dwumianu Newtona
   while (cin >> n >> k)
      cout << "Binom(" << n << "," << k << ") = " << Binom(n, k) << endl;
   return 0;
}
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 369	spoj.sphere.pl - zadanie 78 acm.uva.es - zadanie 10338	acm.uva.es - zadanie 10219

5.2. Największy wspólny dzielnik

Literatura
[WDA] - 33.2
[SZP] - 4.5.3
[MD] - 4.6
[MK] - 4.1
[TLK] - I.2

Największy wspólny dzielnik (ang. greatest common divisor) dwóch liczb a i b ($NWD(a,b)$), jest to największa liczba naturalna, która dzieli bez reszty obie liczby a i b. Przykładowo, $NWD(10,15) = 5$, $NWD(7,2) = 1$. Dwoma interesującymi własnościami największego wspólnego dzielnika, pozwalającymi na jego efektywne wyliczanie są:

$$NWD(a,b) = a, \text{ dla } b = 0$$
$$NWD(a,b) = NWD(a \bmod b, b), \text{ dla } b \neq 0$$

Mając dane dwie liczby naturalne a oraz b, dla których należy wyznaczyć wartość największego wspólnego dzielnika, wystarczy powtarzać proces zamiany ich wartości, przypisując liczbie b wartość $a \bmod b$, a liczbie a wartość liczby b, dopóki $b \neq 0$. Po zakończeniu, wartość wspólnego największego dzielnika jest równa liczbie a. Łatwo można wykazać, że dwukrotne zastosowanie kroku zamiany powoduje co najmniej dwukrotne zmniejszenie sumy liczb a i b, a co za tym idzie, liczba wszystkich wykonywanych kroków jest logarytmiczna ze względu na sumę $a + b$.

Opisany algorytm znany jest jako wyznaczanie największego wspólnego dzielnika metodą Euklidesa. Realizuje go funkcja `LL GCD(LL, LL)`, której implementacja przedstawiona jest na listingu 5.4.

Listing 5.4: Implementacja funkcji **int** `GCD(`**int, int**`)`

```
// Funkcja służąca do wyznaczania największego wspólnego dzielnika dwóch liczb
1 LL GCD(LL a, LL b) {
2     while (b) swap(a %= b, b);
3     return a;
4 }
```

Rozpatrując różne ważne własności największego wspólnego dzielnika, należy wspomnieć o tym, że dla każdej pary dwóch liczb naturalnych a oraz b, istnieją liczby całkowite l oraz k, takie że

$$NWD(a,b) = a * l + b * k$$

Fakt istnienia (a dokładniej możliwości obliczenia) tych dwóch liczb jest użyteczny podczas rozwiązywania wielu problemów, na przykład wyznaczania odwrotności modularnej, co stanowi temat kolejnego rozdziału. Wyznaczenie liczb l oraz k jest możliwe poprzez modyfikację algorytmu Euklidesa. Załóżmy, że znamy wartości liczb l' oraz k', występujące w równaniu postaci:

$$n = (b \bmod a) * l' + a * k'$$

Rozpatrując równanie postaci:

$$n = a * l + b * k$$

możemy uzależnić wartości zmiennych l i k od l' i k'. Poszukiwane podstawienie ma postać:

$$\begin{cases} l = k' - \lfloor \frac{b}{a} \rfloor * l' \\ k = l' \end{cases}$$

Stosując algorytm Euklidesa, dochodzimy pod koniec jego działania do równania postaci:

$$a = l * a + k * 0$$

zatem wartościami zmiennych l oraz k, spełniającymi to równanie, są $l = 1$, $k = 0$. Cofając wszystkie zamiany wartości zmiennych a oraz b wykonane przez algorytm Euklidesa oraz wykonując za każdym razem odpowiednie podstawienia zmiennych l i k, otrzymamy w końcu poszukiwane współczynniki początkowego równania:

$$NWD(a, b) = a * l + b * k$$

Algorytm realizujący tę metodę został zaimplementowany jako funkcja **int** GCDW(**int, int,** LL&, LL&), której implementacja została przedstawiona na listingu 5.5. Funkcja ta przyjmuje jako parametry dwie liczby a i b, dla których wyznaczany jest największy wspólny dzielnik oraz referencje na dwie dodatkowe zmienne, którym przypisywane są wyznaczane wartości współczynników l oraz k. Złożoność czasowa algorytmu nie uległa zmianie w stosunku do oryginalnej wersji algorytmu Euklidesa i wynosi $O(log(a+b))$. Funkcja **int** GCDW(**int, int,** LL&, LL&) jest rekurencyjna (w odróżnieniu od **int** GCD(**int, int**)), a co za tym idzie, zużycie pamięci jest również logarytmiczne.

Listing 5.5: Implementacja funkcji **int** GCDW(**int, int,** LL&, LL&)

```
// Funkcja wyznacza największy wspólny dzielnik dwóch liczb oraz współczynniki
// l i k
01 int GCDW(int a, int b, LL & l, LL & k) {
02    if (!a) {
// gcd(0, b) = 0 * 0 + 1 * b
03       l = 0;
04       k = 1;
05       return b;
06    }
// Wyznacz rekurencyjnie wartość największego wspólnego dzielnika oraz
// współczynniki l oraz k
07    int d = GCDW(b % a, a, k, l);
// Zaktualizuj wartości współczynników oraz zwróć wynik
08    l -= (b / a) * k;
09    return d;
10 }
```

Listing 5.6: Przykład działania funkcji **int** GCDW(**int, int,** LL&, LL&)

```
gcd(10, 15) = 5 = -1*10 + 1*15
gcd(123, 291) = 3 = -26*123 + 11*291
```

Listing 5.7: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.6. Pełny kod źródłowy programu znajduje się w pliku `gcdw.cpp`

```
// Funkcja Pokaz wypisuje wynik wyznaczony przez funkcję GCDW
// dla pary liczb a i b
01 void Pokaz(int a, int b) {
```

Listing 5.7: (c.d. listingu z poprzedniej strony)

```
02    LL l,k;
03    int gcd=GCDW(a,b,l,k);
04    cout << "gcd(" << a << ", " << b << ") = " << GCDW(a,b,l,k);
05    cout << " = " << l << "*" << a << " + " << k << "*" << b << endl;
06 }
07 int main() {
08    Pokaz(10,15);
09    Pokaz(123,291);
10    return 0;
11 }
```

Zadanie: Bilard

Pochodzenie: Potyczki Algorytmiczne 2005

Rozwiązanie: `bil.cpp`

Bajtazar i przyjaciele w piątkowy wieczór udali się do klubu na partyjkę bilarda. Jak zazwyczaj, podczas tego typu spotkań, wywiązała się sprzeczka między Bajtazarem, a Bitolem. Bajtazar zarzucił Bitolowi, że jego strategia gry jest bezsensowna, gdyż kula uderzana przez niego nie ma najmniejszych szans wpaść do łuzy. Bitol natomiast twierdził, że gdyby uderzył kulę dostatecznie mocno, to w końcu wpadłaby ona do jakiejś łuzy. Pomóż rozstrzygnąć spór między kolegami.

Napisz program, który stwierdzi, czy faktycznie kula wpadłaby do łuzy, a jeśli tak, to do której.

Zadanie

Napisz program, który:

- wczyta wymiary stołu bilardowego, początkową pozycję uderzanej kuli oraz wektor wyznaczający ruch kuli po uderzeniu,
- wyznaczy łuzę, do której wpadnie kula lub stwierdzi, że kula nigdy nie wpadnie do żadnej łuzy,
- wypisze wynik.

Wejście

Pierwszy i jedyny wiersz zawiera sześć liczb całkowitych $s_x, s_y, p_x, p_y, w_x, w_y$ oddzielonych pojedynczymi znakami odstępu, gdzie s_x, s_y — wymiary stołu bilardowego, $1 \leqslant s_x, s_y \leqslant 1\,000\,000$, s_x jest parzyste; p_x, p_y — współrzędne początkowego położenia kuli, $0 \leqslant p_x \leqslant s_x, 0 \leqslant p_y \leqslant s_y$; w_x, w_y — współrzędne wektora wyznaczającego ruch kuli, $-1\,000 \leqslant w_x, w_y \leqslant 1\,000$.

Stół bilardowy ma s_x metrów długości i s_y metrów szerokości. Łuzy znajdują się w rogach stołu oraz na środkach boków o długości s_x. Przykładowo, stół o wymiarach $(8, 3)$ ma łuzy w punktach $(0, 0), (4, 0), (8, 0), (0, 3), (4, 3), (8, 3)$. Kule nie wypadają poza obręb stołu, poruszają się bez tarcia, a wszystkie odbicia od band podlegają zasadzie, że kąt padania równa się kątowi odbicia. Kula wpada do łuzy, gdy znajdzie się dokładnie w punkcie, w którym znajduje się dana łuza.

Wyjście

Twój program powinien wypisać jeden wiersz zawierający nazwę łuzy, do której wpadnie kula, bądź słowo NIE, jeśli to się nigdy nie zdarzy. Nazwy kolejnych łuz są następujące:

- GL — dla łuzy o współrzędnych $(0, s_y)$
- GP — dla łuzy o współrzędnych (s_x, s_y)
- GS — dla łuzy o współrzędnych $(\frac{s_x}{2}, s_y)$
- DL — dla łuzy o współrzędnych $(0, 0)$
- DP — dla łuzy o współrzędnych $(s_x, 0)$
- DS — dla łuzy o współrzędnych $(\frac{s_x}{2}, 0)$

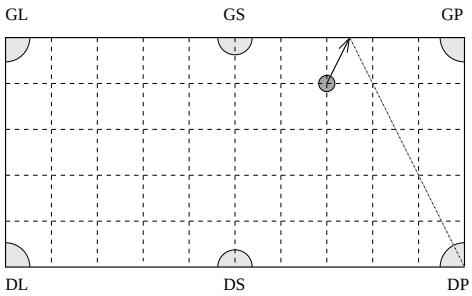
Przykład

Dla następującego wejścia:

```
10 5 7 4 1 2
```

Poprawnym rozwiązaniem jest:

```
DP
```



Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10104	spoj.sphere.pl - zadanie 62	acm.uva.es - zadanie 10294
acm.uva.es - zadanie 10179	acm.uva.es - zadanie 10090	acm.sgu.ru - zadanie 292

5.3. Odwrotność modularna

Literatura
[MD] - 3.6

Załóżmy, że mamy dane równanie modularne z jedną niewiadomą x:

$$a * x \equiv 1 \pmod{m}$$

Chcielibyśmy znaleźć liczbę $x \in \{0, 1 \ldots m-1\}$, dla której powyższe równanie jest spełnione. Problem ten znany jest jako wyznaczanie odwrotności modularnej.

Nasuwającym się na samym początku pomysłem jest sprawdzenie wszystkich m możliwości. Takie rozwiązanie sprawdza się jednak tylko w przypadku małych wartości liczby m. Należy

zatem zastanowić się nad szybszym rozwiązaniem. Kongruencję, którą chcemy rozwiązać można przedstawić w innej postaci poprzez wprowadzenie dodatkowej niewiadomej y, która może przyjmować tylko wartości całkowitoliczbowe:

$$a * x + m * y = 1$$

Z tej postaci od razu widać, że jeśli liczby a i m nie są względnie pierwsze, to równanie nie ma rozwiązania. Gdy jednak liczby te są względnie pierwsze, to istnieje nieskończenie wiele rozwiązań. Załóżmy, że x_0 i y_0 są pewnym rozwiązaniem tego równania. Wtedy $x_1 = x_0 + m*k$, $y_1 = y_0 - a * k$, $k \in N$ również są rozwiązaniem tego równania, dokonując podstawienia, otrzymujemy bowiem:

$$a*x_1+m*y_1 = a*(x_0+m*k)+m*(y_0-a*k) = a*x_0+m*y_0+a*m*k-a*m*k = a*x_0+m*y_0 = 1$$

Wyznaczenia wartości x_0 oraz y_0 można dokonać przy użyciu rozszerzonego algorytmu Euklidesa. Nie ma gwarancji, że wyznaczona w ten sposób liczba x_0 będzie należała do przedziału $\{0, 1 \ldots m-1\}$ (a takiego właśnie rozwiązania poszukujemy). Można to jednak poprawić, wybierając zamiast wyznaczonej wartości x_0, jej odpowiednik różniący się o wielokrotność liczby m, który, jak pokazaliśmy, również jest rozwiązaniem naszego równania.

Listing 5.8 przedstawia implementację funkcji **int** `RevMod(`**int, int**`)`, realizującą omówiony algorytm. Funkcja ta przyjmuje jako parametry dwie liczby — a oraz m, zwraca natomiast wartość zmiennej x lub -1, jeśli nie istnieje odwrotność liczby a (modulo m).

Listing 5.8: Implementacja funkcji **int** `RevMod(`**int, int**`)`.

```
// Funkcja wyznacza odwrotność modularną liczby a (mod m)
1 int RevMod(int a, int m){
2    LL x, y;
3    if (GCDW(a, m, x, y) != 1) return -1;
// Dokonaj przesunięcia zmiennej x, tak aby znajdowała się w przedziale [0..m-1]
4    x %= m;
5    if (x < 0) x += m;
6    return x;
7 }
```

Listing 5.9: Przykładowe wyniki wyliczone przez funkcję **int** `RevMod(`**int, int**`)`

```
Rownanie: 3*x = 1 (mod 7)
x = 5
Rownanie: 5*x = 1 (mod 11)
x = 9
Rownanie: 11*x = 1 (mod 143)
Brak rozwiazan
```

Listing 5.10: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.9. Pełny kod źródłowy programu znajduje się w pliku `revmod.cpp`

```
01 int main() {
02    int a, m;
// Dla wszystkich par liczb wyznacz rozwiązanie równania modularnego
```

Listing 5.10: (c.d. listingu z poprzedniej strony)

```
03    while (cin >> a >> m) {
04       cout << "Rownanie: " << a << "*x = 1 (mod " << m << ")" << endl;
05       int sol = RevMod(a, m);
06       if (sol == -1) cout << "Brak rozwiazan" << endl;
07       else cout << "x = " << sol << endl;
08    }
09    return 0;
10 }
```

5.4. Kongruencje

Literatura
[MK] - 4.6, 4.7
[TLK] - I.3

Dla danej liczby naturalnej n w prosty sposób można wyznaczyć jej reszty z dzielenia przez różne liczby naturalne. Po wykonaniu serii takich operacji można zadać sobie pytanie, czy na podstawie sekwencji reszt uzyskanych w procesie dzielenia można odtworzyć wartość liczby n. Pomocne może się tu okazać chińskie twierdzenie o resztach, zgodnie z którym jeśli mamy dany zbiór liczb parami względnie pierwszych $K = \{k_1, k_2, \ldots k_m\}$, to dla każdej sekwencji reszt $r_1, r_2, \ldots, r_m$ z dzielenia przez liczby ze zbioru K, istnieje dokładnie jedna liczba całkowita w przedziale $[0..k_1 * k_2 * \ldots * k_m - 1]$ dająca takie reszty. Proces odtwarzania wartości n można wykonać krokami, za każdym razem rozwiązując prosty układ kongruencji postaci:

$$\begin{cases} x \equiv a \pmod{p} \\ x \equiv b \pmod{q} \end{cases}$$

i uzyskując na skutek jego rozwiązania nową kongruencję postaci
$x \equiv c \pmod{r}$.

Wyjściowy problem sprowadza się zatem do rozwiązywania układu dwóch kongruencji. Zgodnie z chińskim twierdzeniem o resztach, rozpatrywana sekwencja dzielników jest parami względnie pierwsza. Jednak nie jest to warunek wymagany istnienia rozwiązania. Okazuje się, że rozwiązanie układu dwóch kongruencji istnieje wtedy i tylko wtedy, gdy $a \equiv b \pmod{NWD(p,q)}$ i można wyrazić je wzorem

$$x \equiv a * \beta * q + b * \alpha * p \pmod{\tfrac{p*q}{NWD(p,q)}}$$

gdzie α i β to liczby całkowite spełniające równanie $\alpha * p + \beta * q = NWD(p,q)$.

Funkcja **bool congr(int, int, int, int, int&, int&)** przedstawiona na listingu 5.11 przyjmuje jako parametry liczby a, b, p i q (w takiej właśnie kolejności), a zwraca wartość logiczną, oznaczającą istnienie rozwiązania układu kongruencji. W przypadku istnienia rozwiązania, wartości dwóch ostatnich parametrów funkcji zostają ustawione na odpowiednio liczby c oraz r spełniające kongruencję postaci $x \equiv c \pmod{r}$.

Listing 5.11: Implementacja funkcji **bool congr(int, int, int, int, int&, int&)**

```
// Funkcja wyznacza rozwiązanie układu dwóch kongruencji
01 bool congr(int a, int b, int p, int q, int &c, int &r) {
02    LL x, y;
```

Listing 5.11: (c.d. listingu z poprzedniej strony)

```
03    r = GCDW(p, q, x, y);
// Jeśli liczba a nie przystaje do b (mod nwd(p,q)), to nie ma
// rozwiązania
04    if ((a - b) % r) return 0;
// Wyznacz wartości c oraz r zgodnie ze wzorami
05    x = LL(a) + LL(p) * LL(b - a) / r * x;
06    r = LL(p) * LL(q) / r;
07    c = x % r;
08    if (c < 0) c += r;
09    return 1;
10 }
```

Listing 5.12: Przykład działanie funkcji **bool** `congr(`**int, int, int, int, int&, int&**`)`

```
5 (mod 7), 9 (mod 11), rozwiazanie: 75 (mod 77)
2 (mod 4), 4 (mod 8), rozwiazanie: Brak rozwiazan
```

Listing 5.13: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.12. Pełny kod źródłowy programu znajduje się w pliku `congr.cpp`

```
01 int main() {
02    int a, b, p, q, c, v;
// Dla wszystkich zestawów danych, wyznacz rozwiązania układu kongruencji
03    while(cin >> a >> p >> b >> q) {
04       cout << a << " (mod " << p << "), " << b <<
05          " (mod " << q << "), rozwiazanie: ";
06       if (congr(a, b, p, q, c, v))
07          cout << c << " (mod " << v << ")" << endl;
08       else cout << "Brak rozwiazan" << endl;
09    }
10    return 0;
11 }
```

Zadanie: Wyliczanka

Pochodzenie: IX Olimpiada Informatyczna

Rozwiązanie: `enum.cpp`

Dzieci ustawiły się w kółko i bawią się w wyliczankę. Dzieci są ponumerowane od 1 do n w ten sposób, że (dla $i = 1, 2, ..., n-1$) na lewo od dziecka nr i stoi dziecko nr $i+1$, a na lewo od dziecka nr n stoi dziecko nr 1. Dziecko, „na które wypadnie" w wyliczance, wypada z kółka. Wyliczanka jest powtarzana, aż nikt nie zostanie w kółku. Zasady wyliczanki są następujące:

- pierwszą wyliczankę zaczyna dziecko nr 1. Każdą kolejną wyliczankę rozpoczyna dziecko stojące na lewo od dziecka, które ostatnio wypadło z kółka;
- wyliczanka za każdym razem składa się z k sylab. Dziecko, które zaczyna wyliczankę, mówi pierwszą jej sylabę; dziecko stojące na lewo od niego mówi drugą sylabę, kolejne dziecko trzecią itd. Dziecko, które mówi ostatnią sylabę wyliczanki, odpada z kółka.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis kolejności, w jakiej dzieci wypadały z kółka,
- wyznaczy najmniejszą dodatnią liczbę k, dla której dzieci bawiąc się w k-sylabową wyliczankę będą wypadać z kółka w zadanej kolejności lub stwierdzi, że takie k nie istnieje,
- wypisze wynik na standardowe wyjście wyznaczoną liczbę k lub słowo NIE w przypadku, gdy takie k nie istnieje.

Wejście

W pierwszym wierszu znajduje się jedna dodatnia liczba całkowita n, $2 \leqslant n \leqslant 20$. W drugim wierszu znajduje się n liczb całkowitych pooddzielanych pojedynczymi odstępami — i-ta liczba określa, jako które z kolei dziecko nr i wypadło z kółka.

Wyjście

Twój program powinien wypisać w pierwszym i jedynym wierszu jedną liczbę całkowitą: najmniejszą liczbę k sylab, jakie może mieć wyliczanka lub jedno słowo NIE, jeśli taka liczba nie istnieje.

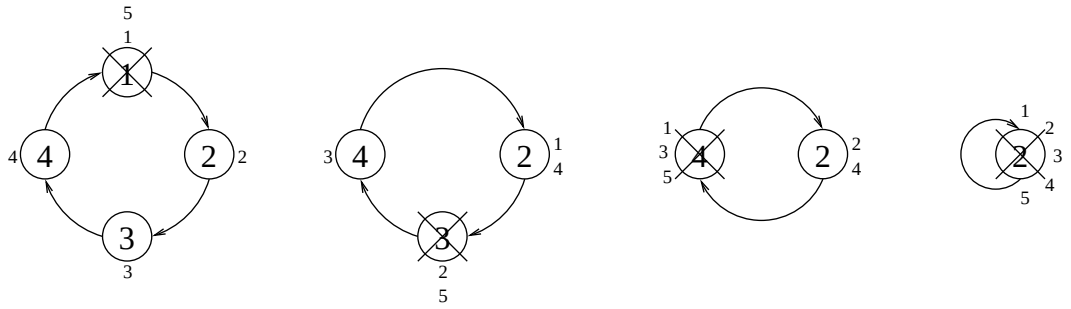
Przykład

Dla następującego wejścia:

```
4
1  4  2  3
```

Poprawnym rozwiązaniem jest:

```
5
```



5.5. Szybkie potęgowanie modularne

W prezentowanym rozdziale zajmiemy się problemem potęgowania modularnego, czyli dla danych liczb a, b oraz q, wyliczania wartości wyrażenia:

$$a^b \pmod q$$

Literatura
[WDA] - 33.6

Jeżeli rozpatrywać zagadnienie polegające na liczeniu wartości a^b, możemy mieć dwa różne przypadki — albo wyznaczany wynik mieści się w standardowym typie zmiennych, albo nie.

W tym pierwszym przypadku, jeżeli zastosujemy prostą metodę potęgowania polegającą na b-krotnym pomnożeniu przez siebie liczby a, to liczba wykonywanych mnożeń będzie ograniczona przez liczbę bitów wykorzystywanych do reprezentacji wykorzystywanych zmiennych. Ponieważ jest to stała wartość (zazwyczaj 32 lub 64), więc algorytm uzyskany w ten sposób działa w czasie stałym (dla $a > 1$, b nie może być większe niż 32 / 64). Inna sytuacja zachodzi, gdy wyznaczany wynik nie mieści się w arytmetyce. Wtedy pojawiają się dodatkowe problemy, związane z implementacją własnej arytmetyki (przykład takiej arytmetyki przedstawiony jest w rozdziale „Arytmetyka wielkich liczb"). Podczas zawodów, w przypadku pojawiającej się potrzeby implementacji własnej arytmetyki, nie ma czasu na tworzenie bardzo efektywnych operacji. Chociaż istnieje algorytm pozwalający na mnożenie dwóch liczb w czasie $O(n * log(n) * log(log(n)))$, to zazwyczaj implementowany algorytm ma złożoność $O(n * m)$, gdzie n i m to liczby cyfr występujących w mnożonych liczbach. W takich sytuacjach, każda operacja mnożenia jest bardzo kosztowna i dowolne „oszczędności" w liczbie wykonywanych mnożeń podczas potęgowania dają duże różnice. Zmiana sposobu wykonywania potęgowania jest w stanie znacznie zredukować złożoność czasową algorytmu.

Podobna sytuacja zachodzi w przypadku wykonywania potęgowania modularnego — w takiej sytuacji mamy gwarancję, że wynik jest mniejszy od wartości liczby q (czyli mieści się w arytmetyce komputera), nie mając jednocześnie żadnych ograniczeń dotyczących wartości liczb a oraz b (jak było w pierwszym rozpatrywanym przez nas przypadku). W takiej sytuacji, dobrym pomysłem jest wykorzystanie algorytmu szybkiego potęgowania modularnego. Jego zasada działania polega na analizie reprezentacji binarnej liczby b oraz odpowiedniego mnożenia liczby a przez kolejne potęgi a^{2^k}:

$$a^b = a^{(b_m b_{m-1} \ldots b_0)_2} = a^{b_m * 2^m + b_{m-1} * 2^{m-1} + \ldots + b_0 * 2^0} = a^{b_m * 2^m} * a^{b_{m-1} * 2^{m-1}} * \ldots * a^{b_0 * 2^0}$$

Algorytm szybkiego potęgowania modularnego wylicza wartości kolejnych potęg a^{2^m} oraz w przypadku, gdy b_m jest równe 1, mnoży wynik przez a^{2^m}. Implementacja tego algorytmu została zrealizowana w funkcji **int** `ExpMod(`**int, int, int**`)`, której kod źródłowy przedstawiony jest na listingu 5.14. Złożoność całego algorytmu to $O(log(b))$.

Listing 5.14: Implementacja funkcji **int** `ExpMod(`**int, int, int**`)`

```
// Funkcja realizująca szybkie potęgowanie modularne
1 int ExpMod(int a, int b, int q) {
2    LL p = 1;
3    while (b > 0) {
4       if (b & 1) p = (LL(a) * p) % q;
5       a = (LL(a) * LL(a)) % q;
6       b /= 2;
7    }
8    return p;
9 }
```

Listing 5.15: Przykład działania funkcji **int** `ExpMod(`**int, int, int**`)`

```
2 do potegi 100 (mod 10) = 6
3 do potegi 17 (mod 123) = 3
```

Listing 5.16: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.15. Pełny kod źródłowy programu znajduje się w pliku `expmod.cpp`

```
1 int main() {
2    int a, b, q;
// Dla wszystkich danych wejściowych, wyznacz wartość liczby a^b (mod q)
3    while(cin >> a >> b >> q) cout << a << " do potegi " << b <<
4       " (mod " << q << ") = " << ExpMod(a,b,q) << endl;
5    return 0;
6 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.sgu.ru - zadanie 117	acm.uva.es - zadanie 10006	acm.uva.es - zadanie 10710

5.6. Sito Eratostenesa

Załóżmy, że mamy do czynienia z zadaniem, którego rozwiązanie wymaga sprawdzania dla różnych liczb k z przedziału $[1..n]$, czy k jest liczbą pierwszą. W takiej sytuacji widać, że nawet jeżeli pytania o tę samą liczbę nie będą się powtarzać, to i tak może okazać się opłacalne na początku działania programu sprawdzenie, które liczby z tego zakresu są pierwsze a które nie, a następnie zapamiętanie tych wyników w celu późniejszego użycia (o ile tylko n jest na tyle małe, że jesteśmy w stanie pomieścić wszystkie wyniki w pamięci). Doskonała do tego celu jest metoda sita Eratostenesa, której zasada działania jest bardzo prosta: buduje się tablicę liczb od 2 do n, a następnie wykreśla się z niej kolejne wielokrotności jeszcze nie wykreślonych liczb, zaczynając od liczby 2. W ten sposób dla liczby 2 zostaną wykreślone liczby $4, 6, 8, \ldots$, dla liczby 3 — $6, 9, 12, \ldots$, dla liczby 5 — $10, 15, 20, \ldots$. Po zakończeniu tego procesu, jedynymi niewykreślonymi elementami będą liczby pierwsze (co wynika bezpośrednio z ich definicji — liczba pierwsza nie ma dzielników różnych od 1 i jej samej).

Funkcja realizująca ten algorytm to `bit_vector Sieve(int)`, której kod źródłowy został przedstawiony na listingu 5.17. Funkcja ta jako parametr przyjmuje wielkość zakresu, natomiast zwraca jako wynik wektor binarny, w którym na k-tej pozycji znajduje się wartość 1, jeśli liczba k jest pierwsza, a 0 wpp. Złożoność czasowa algorytmu to $O(n * log(log(n)))$, co zawdzięczane jest temu, że wykreślanie wielokrotności liczby pierwszej p następuje dopiero od jej kwadratu (p^2). Wszystkie mniejsze wielokrotności liczby p zostały bowiem już wcześniej wykreślone podczas przetwarzania mniejszych liczb pierwszych.

Listing 5.17: Implementacja funkcji `bit_vector Sieve(int)`

```
// Funkcja realizuje algorytm sita Eratostenesa
1 bit_vector Sieve(int s) {
2    bit_vector V(s+1,1);
3    V[0] = V[1] = 0;
// Dla kolejnej liczby, jeśli nie została ona wykreślona, to wykreśl wszystkie jej
wielokrotności
4    FOR(x, 2, s) if(V[x] && LL(s) >= LL(x) * LL(x))
5       for(int y = x * x; y <= s; y += x) V[y] = 0;
```

Listing 5.17: (c.d. listingu z poprzedniej strony)

```
6     return V;
7 }
```

Listing 5.18: Przykład działania funkcji `bit_vector Sieve(int)`

```
0 nie jest liczba pierwsza
1 nie jest liczba pierwsza
2 jest liczba pierwsza
3 jest liczba pierwsza
4 nie jest liczba pierwsza
5 jest liczba pierwsza
6 nie jest liczba pierwsza
7 jest liczba pierwsza
```

Listing 5.19: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.18. Pełny kod źródłowy programu znajduje się w pliku `sieve.cpp`

```
1 int main() {
2     int n;
// Wczytaj wielkość zakresu
3     cin >> n;
4     bit_vector V = Sieve(n);
// Dla każdej liczby z zakresu wypisz, czy jest ona liczbą pierwszą
5     REP(x, n) cout << x << (V[x] ? " jest liczba pierwsza"
6        : " nie jest liczba pierwsza") << endl;
7     return 0;
8 }
```

5.7. Lista liczb pierwszych

Innym sposobem przedstawienia problemu z poprzedniego rozdziału, może być chęć wyznaczenia wszystkich liczb pierwszych w danym przedziale $[1 \dots n-1]$. Podejście takie może okazać się przydatne w zadaniach wymagających wyznaczania k-tej liczby pierwszej. Sposób wyznaczania wszystkich liczb pierwszych mniejszych od zadanej wartości, można zrealizować sprawdzając dla kolejnych liczb naturalnych, czy są one pierwsze. Aby tego dokonać, należy upewnić się, że liczba nie ma żadnych dzielników różnych od 1 oraz jej samej. Istotne z punktu widzenia złożoności czasowej algorytmu są dwie obserwacje:

- jeśli liczba m posiada jakikolwiek dzielnik różny od 1 i m, to posiada również dzielnik mniejszy bądź równy $\sqrt{m}$ (dla każdego dzielnika k liczby m, $\frac{m}{k}$ jest też dzielnikiem m) — dzięki temu wystarczy szukać dzielników liczby m w przedziale $[2 \dots \sqrt{m}]$.
- jeśli liczba m posiada dzielnik k niebędący liczbą pierwszą, to posiada również dzielnik $k' < k$, będący liczbą pierwszą.

Dzięki tym dwóm spostrzeżeniom, wystarczy poszukiwać dzielników liczby m wśród liczb pierwszych mniejszych bądź równych $\sqrt{m}$.

Funkcja `VI PrimeList(int)` przedstawiona na listingu 5.20 realizuje opisany powyżej algorytm, przyjmując jako parametr liczbę n, natomiast zwracając jako wynik wektor liczb pierwszych mniejszych od n.

Listing 5.20: Implementacja funkcji `VI PrimeList(int)`

```
// Funkcja wyznacza listę liczb pierwszych mniejszych od n
VI PrimeList(int n) {
// Stwórz listę liczb pierwszych zawierającą liczbę 2
  VI w(1, 2);
  int s = 0, i = 2;
// Dla kolejnej liczby sprawdź, czy jest pierwsza i jeśli tak, to dodaj ją do
// listy
  FOR(l, 3, n - 1) {
    i = 0;
    while (w[s] * w[s] <= l) s++;
    while (i < s && l % w[i]) i++;
// Nie znaleziono dzielnika liczby l - jest ona pierwsza
    if (i == s) w.PB(l);
  }
  return w;
}
```

Opisana metoda wyznaczania listy liczb pierwszych nie jest jedyną możliwą. Innym sposobem jest np. wykorzystanie wyniku wyznaczonego przez algorytm sita Eratostenesa. Na jego podstawie w prosty sposób można wyznaczyć wszystkie liczby pierwsze mniejsze od zadanego ograniczenia. Metoda taka w praktyce okazuje się około 5-krotnie szybsza od algorytmu realizowanego przez funkcję `VI PrimeList(int)`, ze względu na wykonywaną przez nią stosunkowo wolną operację arytmetyczną liczenia reszty z dzielenia.

Przedstawiona na listingu 5.21 funkcja `VI PrimeListS(int)` wyznacza liczby pierwsze w oparciu o algorytm sita. Jest ona szybsza od `VI PrimeList(int)`, jednak zużywa również więcej pamięci. Jeśli wyznaczanie liczb pierwszych wykonywane jest w zakresie nie większym niż $[1 \ldots 2\,000\,000\,000]$ oraz mamy wystarczająco dużo pamięci, to nowo opisana metoda działa istotnie lepiej od poprzedniego rozwiązania.

Listing 5.21: Implementacja funkcji `VI PrimeListS(int)`

```
// Funkcja wyznacza listę liczb pierwszych mniejszych od n, wykorzystując do
// tego celu sito Eratostenesa
VI PrimeListS(int s) {
  bit_vector l = Sieve(++s);
  VI V;
  REP(x, s) if (l[x]) V.PB(x);
  return V;
}
```

Listing 5.22: Przykład działania funkcji `VI PrimeListS(int)`

```
Lista liczb pierwszych mniejszych od 100:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
```

Listing 5.22: (c.d. listingu z poprzedniej strony)

```
71, 73, 79, 83, 89, 97
```

Listing 5.23: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.22. Pełny kod źródłowy programu znajduje się w pliku `primelist.cpp`

```
01 int main() {
02    int n;
// Wczytaj analizowany zakres
03    cin >> n;
// Wyznacz listę liczb pierwszych oraz wypisz wynik
04    VI l = PrimeListS(n);
05    int count = 0;
06    cout << "Lista liczb pierwszych mniejszych od " << n << ":" << endl;
07    FOREACH(it, l) {
08       if (count++) cout << ", ";
09       if (!(count % 20)) cout << endl;
10       cout << *it;
11    }
12    return 0;
13 }
```

5.8. Test pierwszości

Literatura
[WDA] - 33.8
[RII] - 11
[TLK] - V

Rozważaliśmy już problem wyznaczania wszystkich liczb pierwszych mniejszych od pewnej liczby n. W tym podrozdziale postawimy sobie inny problem — dla danej liczby n należy sprawdzić, czy jest ona pierwsza. Teoretycznie, do rozwiązania tego problemu można zastosować jeden z algorytmów z poprzednich dwóch rozdziałów, jednak podejście takie zaaplikowane wprost byłoby wyjątkowo nieefektywne. Istnieje jednak możliwość zaadoptowania poprzedniego pomysłu. Korzystając z własności dzielników liczb, które wykorzystaliśmy do konstrukcji algorytmu realizowanego przez funkcję `VI PrimeList(int)`, można dla danej liczby n sprawdzić, czy nie ma ona żadnych dzielników mniejszych bądź równych $\sqrt{n}$. Należy na wstępie zauważyć jednak, że taki algorytm ma złożoność $O(\sqrt{n})$, co jest funkcją wykładniczą w stosunku do liczby cyfr występujących w zapisie liczby n. Algorytm ten został zrealizowany w funkcji **bool** `IsPrime(int)`, której kod źródłowy został przedstawiony na listingu 5.24. Jako parametr, funkcja przyjmuje liczbę n, a zwraca prawdę, jeśli n jest liczbą pierwszą, a fałsz wpp.

Listing 5.24: Implementacja funkcji **bool** `IsPrime(int)`

```
// Funkcja sprawdza, czy liczba n jest pierwsza
1 bool IsPrime(int n) {
2    if (x < 2) return 0;
3    for(int x = 2; x * x <= n; x++)
4       if (!(n % x)) return 0;
5    return 1;
6 }
```

Jak już zauważyliśmy wcześniej, algorytm realizowany przez funkcję **bool** `IsPrime(`**int**`)` ma złożoność wykładniczą. Przez długi czas nie był znany żaden deterministyczny algorytm wielomianowy, pozwalający na stwierdzenie czy dana liczba jest pierwsza. Wprawdzie istniały różne randomizacyjne algorytmy, które w praktyce działały bardzo dobrze (takie jak test Millera-Rabina), jednak z teoretycznego punktu widzenia bardzo ważnym było pokazanie istnienia algorytmu deterministycznego. Dopiero w 2002 roku trzech naukowców — Manindra Agrawal, Neeraj Kayal oraz Nitin Saxena przedstawili algorytm wielomianowy o nazwie AKS (jak można się domyśleć, nazwa pochodzi od nazwisk jego autorów). Ogromna złożoność czasowa tego algorytmu — $O(log^{12}(n))$ oraz bardzo skomplikowana implementacja powodują jego małą przydatność praktyczną. O algorytmie tym można przeczytać w oryginalnej pracy jego autorów — „Primes is in P" [AKS].

Wielomianową metodą sprawdzania pierwszości liczb, którą przedstawimy, jest test Millera-Rabina. Zasada działania tego algorytmu bazuje na małym twierdzeniu Fermata, które mówi, że jeżeli p jest liczbą pierwszą, to

$$a^p \equiv a \pmod{p}$$

Dobierając za liczbę $a = 2$, można wykazać, że liczba 875 jest złożona:

$$2^{875} \pmod{875} = 193 \pmod{875} \neq 2 \pmod{875}$$

Wybierając do testu liczbę 17 otrzymujemy:

$$2^{17} \pmod{17} = 2 \pmod{17}$$

zatem mamy prawo podejrzewać, że 17 jest liczbą pierwszą, co jest w istocie prawdą. Jednak sprawdzając w ten sposób liczbę 341, otrzymujemy:

$$2^{341} \pmod{341} = 2 \pmod{341}$$

pomimo tego, że 341 jest złożona — $341 = 11 * 31$. Dobierając jednak inną wartość liczby $a = 3$, otrzymamy:

$$3^{341} \pmod{341} = 168 \pmod{341}$$

co jest świadectwem tego, że liczba 341 jest złożona. Można wykazać, że prawdopodobieństwo uzyskania takiej sytuacji, jaka miała miejsce w przypadku pary liczb $n = 341$, $a = 2$ nie przekracza $\frac{1}{4}$, zatem wykonując k testów z różnymi wartościami liczby a, prawdopodobieństwo otrzymania za każdym razem złej odpowiedzi wynosi $\frac{1}{4^k}$.

Niestety istnieją liczby złożone n, dla których własność $a^n \equiv a \pmod{n}$ zachodzi dla każdego a. Liczby takie nazywane są liczbami Carmichaela. Najmniejszą z nich jest 561. Algorytm do testowania pierwszości liczb bazujący na małym twierdzeniu Fermata, zawsze będzie dawał złą odpowiedź dla tych liczb. Na szczęście istnieje możliwość poprawienia tego algorytmu poprzez dodanie dodatkowego testu, zwanego testem Millera. Podstawą tego testu jest własność liczb pierwszych postaci:

$$x^2 \equiv 1 \pmod{p} \Rightarrow x \equiv \pm 1 \pmod{p}$$

gdzie p jest liczbą pierwszą. Przy użyciu tej własności można wykazać, że liczba 561 jest złożona. Mianowicie:

$$5^{560} = (5^{280})^2 \equiv 1 \pmod{561} \text{ oraz } 5^{280} \equiv 67 \pmod{561}$$

Algorytm Millera-Rabina przeprowadza serię testów bazujących na małym twierdzeniu Fermata, wykonując jednocześnie testy Millera dla kolejnych potęg liczby a wyznaczanych podczas szybkiego potęgowania. Implementacja tego algorytmu została przedstawiona na listingu 5.25 jako funkcja **bool** `IsPrime(int)`. Funkcja ta wykonuje testy dla liczb $a \in \{2, 3, 5, 7\}$, co gwarantuje jej prawidłowe działanie dla wszystkich liczb należących do przedziału $[1 \ldots 2\,000\,000\,000]$ (fakt ten został zweryfikowany empirycznie).

Listing 5.25: Implementacja funkcji **bool** `IsPrime(int)`

```
// Funkcja przeprowadza test Millera-Rabina dla liczby x przy podstawie n
bool MaR(LL x, LL n) {
   if (x >= n) return 0;
   LL d = 1, y;
   int t = 0, l = n - 1;
   while (!(l & 1)) {
      ++t;
      l >>= 1;
   }
   for (; l > 0 || t--; l >>= 1) {
      if (l & 1) d = (d * x) % n;
      if (!l) {
         x = d;
         l = -1;
      }
      y = (x * x) % n;
// Jeśli test Millera wykrył, że liczba nie jest pierwsza, to zwróć prawdę
      if (y == 1 && x != 1 && x != n - 1) return 1;
      x = y;
   }
// Jeśli nie jest spełnione założenie twierdzenia Fermata, to liczba jest
// złożona
   return x != 1;
}
// Funkcja sprawdza, czy dana liczba x jest pierwsza. W tym celu wykonuje test
// Millera-Rabina przy podstawie 2, 3, 5, 7
bool IsPrime(int x) {
   return !(x < 2 || MaR(2, x) || MaR(3, x) || MaR(5, x) || MaR(7, x));
}
```

Listing 5.26: Wyznaczanie liczb pierwszych na przedziale $[2\,000\,000\,000 \ldots 2\,000\,000\,100]$ przy użyciu funkcji **bool** `IsPrime(int)`

```
2000000011    2000000033    2000000063    2000000087    2000000089
2000000099
```

Listing 5.27: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.26. Pełny kod źródłowy programu znajduje się w pliku `isprime.cpp`

```
int main() {
   int b, e, count = 0;
```

Listing 5.27: (c.d. listingu z poprzedniej strony)

```
// Wczytaj przedział testowanych liczb
3   cin >> b >> e;
// Dla każdej liczby z przedziału sprawdź, czy jest pierwsza
4   while (b < e) {
5      if (IsPrime(b)) cout << b << (!(++count % 5) ? "\n" : "\t");
6      b++;
7   }
8   return 0;
9 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.sgu.ru - zadanie 113	acm.sgu.ru - zadanie 116	acm.uva.es - zadanie 10139
acm.uva.es - zadanie 382	acm.uva.es - zadanie 294	acm.sgu.ru - zadanie 200
acm.uva.es - zadanie 160	acm.sgu.ru - zadanie 231	spoj.sphere.pl - zadanie 2
acm.uva.es - zadanie 406	acm.uva.es - zadanie 10168	spoj.sphere.pl - zadanie 134
acm.uva.es - zadanie 543	acm.uva.es - zadanie 583	spoj.sphere.pl - zadanie 288
acm.uva.es - zadanie 136	acm.uva.es - zadanie 10235	

5.9. Arytmetyka wielkich liczb

Literatura
[SZP] - 4.4, 4.5.1

Zdarzają się zadania, w których zakres standardowych typów zmiennych jest niewystarczający. W takich przypadkach pojawia się konieczność implementowania własnej arytmetyki wielkich liczb. W zależności od rozwiązywanego zadania, implementacja ta powinna umożliwiać wykonywanie pewnych operacji arytmetycznych. Wielkie liczby zazwyczaj przechowuje się w postaci tablicy cyfr. Ze względów wydajnościowych, nie reprezentuje się ich przy podstawie 10, lecz istotnie większej, co ma na celu zmniejszenie liczby cyfr, a tym samym zwiększenie wydajności wykonywanych operacji.

Przedstawiona w tym rozdziale implementacja wielkich liczb — zrealizowana jako struktura `BigNum` z listingu 5.28 — ma dość długą implementację, co wynika z dużej liczby wspomaganych operacji. Implementacja została zrealizowana w taki sposób, aby umożliwić zminimalizowanie ilości kodu, który trzeba przepisać, aby zapewnić sobie wymaganą funkcjonalność struktury. Implementacja została podzielona na sekcje: część podstawowa, która jest zawsze wymagana; operatory porównawcze pozwalające na dokonywanie porównywania wartości liczb; operacje na parach (wielka liczba, zmienna typu **int**); operatory działające na parach (wielka liczba, wielka liczba) oraz operatory pozwalające na wczytywanie i wypisywanie wartości wielkich liczb. Implementacja niektórych operatorów wykorzystuje inne — w takich przypadkach komentarze znajdujące się przed implementacją operatora informują, jakie inne operatory należy również zaimplementować. Najlepszym sposobem przeanalizowania sposobu działania arytmetyki `BigNum` jest prześledzenie jej kodu źródłowego oraz komentarzy. Poniżej znajduje się lista wspomaganych operacji, wraz z ich krótkim opisem. Zakładamy, że w przypadku opisu operatorów, liczby cyfr liczb, na których wykonywane są

operacje, to odpowiednio n i m. Przez p będzie oznaczana natomiast podstawa, przy której wykonywane są operacje na wielkich liczbach. Zaimplementowane operatory to:

- `Bignum(int a, int b)` — konstruktor tworzący nową wielką liczbę równą a, mającą zaalokowaną pamięć na b cyfr. W przypadku, gdy na skutek wykonywanych operacji, liczba cyfr przekroczy przeznaczoną na nie pamięć, nastąpi automatyczne zaalokowanie nowej pamięci. Jeśli z góry wiemy, że wynik który obliczamy będzie miał przykładowo 500 cyfr, to ze względów efektywnościowych należy poinformować o tym konstruktor;

- `BigNum(const BigNum&)` — konstruktor tworzący nową liczbę równą innej, istniejącej już wielkiej liczbie;

- `res(int)` — funkcja służy do alokowania nowej pamięci dla cyfr (o ile zachodzi taka potrzeba);

- `przen(int)` — implementacja typu `BigNum` wykorzystuje do przechowywania cyfr zmiennych typu **long long** oraz podstawy 1 000 000 000. Dzięki temu, podczas wykonywania operacji dodawania czy mnożenia dwóch liczb, nie następują przepełnienia wartości zmiennych i operatory nie muszą „zajmować się" przenoszeniem nadmiaru do starszych cyfr. Procesem tym, po zakończeniu wykonywania operacji arytmetycznej, zajmuje się funkcja `przen(int)`, której parametrem jest liczba cyfr (licząc od najmniej znaczącej), dla których należy wykonać przeniesienie. Takie podejście zmniejsza nieco efektywność, ale w istotny sposób upraszcza implementację;

- operatory porównawcze dwóch liczb typu `BigNum`: `==`, `<`, `>`, `<=`, `>=`, `!=` — ich znaczenie jest standardowe, a złożoność czasowa to $O(min(n, m))$;

- `BigNum&` **operator**`=(int)` — przypisanie wartości zmiennej typu **int** do wielkiej liczby. Złożoność to $O(n)$ (liczba cyfr wielkiej liczby, której wartość jest zmieniana);

- operatory **void operator**`+=(int)` oraz **void operator**`-=(int)` — ich pesymistyczna złożoność zależy od wykonywanych przenoszeń, koszt zamortyzowany wykonania sekwencji inkrementacji bądź dekrementacji (przy wykonywaniu operacji tylko jednego typu) to $O(1)$;

- operatory **void operator**`*=(int)`, **int operator**`/=(int)` oraz **int operator** `%(int)` — ich złożoność to $O(n)$. Operator **int operator**`/=(int)` działa dość nietypowo; dzieli wartość wielkiej liczby przez zadany parametr oraz zwraca resztę z dzielenia;

- `BigNum&` **operator** `+=(const BigNum&)`, `BigNum` **operator** `+(const BigNum&)`, `BigNum&` **operator** `-=(const BigNum&)` oraz `BigNum` **operator** `-(const BigNum&)` — ich zamortyzowana złożoność to $O(m)$;

- `BigNum&` **operator** `*=(const BigNum&)`, `BigNum` **operator** `*(const BigNum &)` — ich złożoność to $O(n * m)$;

- `BigNum&` **operator** `/=(const BigNum&)`, `BigNum` **operator** `/(const BigNum&)`, `BigNum&` **operator** `%=(const BigNum&)` oraz `BigNum` **operator** `%(const BigNum&)` — ich złożoność to $O(n * m * log(p))$;

- `BigNum&` **operator** `=(const BigNum&)` — złożoność tego operatora to $O(n + m)$;

- `BigNum& operator =(string)` — operator ten przypisuje zmiennej `BigNum` wartość liczby reprezentowanej przez zadany tekst. Jego złożoność jest liniowa ze względu na długość tekstu;
- **void** `write()` oraz **void** `write(char*)` — wypisywanie wartości liczby na standardowe wyjście / do bufora. Złożoność to $O(n)$;
- `BigNum& operator<<=(int)`, `BigNum operator<<(int)`, `BigNum& operator>>=(int)` oraz `BigNum operator>>(int)` — dokonują przesunięcia cyfr liczby o zadaną liczbę pozycji;
- `BigNum sqrt()` — wyznacza pierwiastek całkowity z wielkiej liczby. Jego złożoność to $O(n^2 * log(p))$.

Listing 5.28: Implementacja struktury `BigNum`

```
// Implementacja struktury BigNum, realizującej arytmetykę wielkich liczb
001 struct BigNum {
// Makro służące do eliminowania wiodących zer
002 #define REDUCE() while(len>1 && !cyf[len-1]) len--;
// Podstawa, przy której wykonywane są obliczenia oraz liczba zer w podstawie
003    static const int BASE = 1000000000, BD = 9;
// Zmienna len reprezentuje aktualną długość liczby (liczbę cyfr), a al
// wielkość zaalokowanej pamięci do przechowywania cyfr liczby
004    int len, al;
// Wskaźnik do tablicy cyfr liczby
005    LL *cyf;
// Konstruktor liczby o wartości v i zaalokowanej pamięci dla l cyfr
006    BigNum(int v = 0, int l = 2) : len(1), al(l), cyf(new LL[l]) {
007       REP(x, al) cyf[x] = 0;
008       if ((cyf[0] = v) >= BASE) przen(1);
009    }
// Konstruktor, przypisujący wartość innej liczby typu BigNum
010    BigNum(const BigNum & a) : len(a.len), al(len), cyf(new LL[al]) {
011       REP(x, al) cyf[x] = a.cyf[x];
012    }
// Destruktor
013    ~BigNum() {
014       delete cyf;
015    }
// Funkcja przyjmuje jako parametr zapotrzebowanie na liczbę cyfr i jeśli
// zapotrzebowanie jest większe od aktualnego rozmiaru tablicy cyfr, to dokonuje
// realokacji
016    void Res(int l) {
017       if (l > al) {
018          LL *n = new LL[l = max(l, 2 * al)];
019          REP(x, l) n[x] = x >= al ? 0 : cyf[x];
020          delete cyf;
021          cyf = n;
022          al = l;
023       }
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
024    }
// Funkcja dokonuje przenoszenia do starszych cyfr nadmiaru powstałego na skutek
// wykonywania operacji. Jest to jedyne miejsce w całej strukturze, gdzie
// wykonuje się tą operację. Parametr określa liczbę cyfry, do której należy
// wykonać przenoszenie nadmiaru
025    void przen(int p) {
026       int x = 0;
027       for (; x < p || cyf[x] < 0 || cyf[x] >= BASE; x++) {
028          Res(x + 2);
// W razie potrzeby wykonaj zapożyczenie od starszej cyfry...
029          if (cyf[x] < 0) {
030             LL i = (-cyf[x] - 1) / BASE + 1;
031             cyf[x] += i * BASE;
032             cyf[x + 1] -= i;
033          } else
// lub wykonaj przeniesienie powstałego nadmiaru
034          if (cyf[x] >= BASE) {
035             LL i = cyf[x] / BASE;
036             cyf[x] -= i * BASE;
037             cyf[x + 1] += i;
038          }
039       }
040       len = max(len, x + 1);
041       REDUCE();
042    }
// Od tego miejsca zaczyna się implementacja operatorów. Przepisywanie tej
// części kodu nie jest wymagane - należy przepisywać tylko te operatory, z
// których się korzysta. Niektóre operatory korzystają z innych - w takich
// przypadkach, przy każdym operatorze napisane jest, implementacji jakich
// operatorów on wymaga
// Poniższe makro pozwala skrócić zapis nagłówków operatorów
043 #define OPER1(op) bool operator op (const BigNum &a) const
// Operatory porównawcze
044    OPER1(==) {
045       if (a.len != len) return 0;
046       REP(x, len) if (cyf[x] != a.cyf[x]) return 0;
047       return 1;
048    }
049    OPER1(<) {
050       if (len != a.len) return len < a.len;
051       int x = len - 1;
052       while (x && a.cyf[x] == cyf[x]) x--;
053       return cyf[x] < a.cyf[x];
054    }
// Operator ten wymaga implementacji operatora <(BigNum)
055    OPER1(>) {
056       return a < *this;
057    }
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
// Operator ten wymaga implementacji operatora <(BigNum)
058   OPER1(<=) {
059      return !(a < *this);
060   }
// Operator ten wymaga implementacji operatora <(BigNum)
061   OPER1(>=) {
062      return !(*this < a);
063   }
// Operator ten wymaga implementacji operatora ==(BigNum)
064   OPER1(!=) {
065      return !(*this == a);
066   }
// Operacje dla liczb typu int
067   BigNum & operator=(int a) {
068      REP(x, len) cyf[x] = 0;
069      len = 1;
070      if (cyf[0] = a >= BASE) przen(1);
071      return *this;
072   }
073   void operator+=(int a) {
074      cyf[0] += a;
075      przen(1);
076   }
077   void operator-=(int a) {
078      cyf[0] -= a;
079      przen(1);
080   }
081   void operator*=(int a) {
082      REP(x, len) cyf[x] *= a;
083      przen(len);
084   }
// Poniższy operator zwraca jako wynik resztę z dzielenia liczby typu BigNum
// przez liczbę typu int
085   int operator/=(int a) {
086      LL w = 0;
087      FORD(p, len - 1, 0) {
088         w = w * BASE + cyf[p];
089         cyf[p] = w / a;
090         w %= a;
091      }
092      REDUCE();
093      return w;
094   }
095   int operator%(int a) {
096      LL w = 0;
097      FORD(p, len - 1, 0) w = (w * BASE + cyf[p]) % a;
098      return w;
099   }
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
// Operacje wyłącznie na liczbach typu BigNum
#define OPER2(op) BigNum& operator op (const BigNum &a)
  OPER2(+=) {
    Res(a.len);
    REP(x, a.len) cyf[x] += a.cyf[x];
    przen(a.len);
    return *this;
  }
  OPER2(-=) {
    REP(x, a.len) cyf[x] -= a.cyf[x];
    przen(a.len);
    return *this;
  }
  OPER2(*=) {
    BigNum c(0, len + a.len);
    REP(x, a.len) {
      REP(y, len) c.cyf[y + x] += cyf[y] * a.cyf[x];
      c.przen(len + x);
    }
    *this = c;
    return *this;
  }
// Operator ten wymaga implementacji następujących operatorów: <(BigNum),
// +=(BigNum), *=(BigNum), +(BigNum), *(BigNum), <<(int),
// <<=(int)
  OPER2(/=) {
    int n = max(len - a.len + 1, 1);
    BigNum d(0, n), prod;
    FORD(i, n - 1, 0) {
      int l = 0, r = BASE - 1;
      while (l < r) {
        int m = (l + r + 1) / 2;
        if (*this < prod + (a * m << i)) r = m - 1;
        else l = m;
      }
      prod += a * l << i;
      d.cyf[i] = l;
      if (l) d.len = max(d.len, i + 1);
    }
    *this = d;
    return *this;
  }
// Operator ten wymaga implementacji następujących operatorów: <(BigNum),
// +=(BigNum), *=(BigNum), +(BigNum), *(BigNum), <<(BigNum),
// <<=(BigNum)
  OPER2(%=) {
    BigNum v = *this;
    v /= a;
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
141       v *= a;
142       *this -= v;
143       return *this;
144    }
145    OPER2(=) {
146       Res(a.len);
147       FORD(x, len - 1, a.len) cyf[x] = 0;
148       REP(x, a.len) cyf[x] = a.cyf[x];
149       len = a.len;
150       return *this;
151    }
152 // Operatory służące do wczytywania i wypisywania liczb
153 // Funkcja przypisuje liczbie BigNum wartość liczby z przekazanego wektora,
154 // zapisanej przy podstawie p
155 // Operator ten wymaga implementacji +=(int), *=(int)
156    void read(const VI & v, int p) {
157       *this = 0;
158       FORD(x, SIZE(v), 0) {
159          *this *= p;
160          *this += v[x];
161       }
162    }
163 // Funkcja przypisuje liczbie BigNum wartość liczby z napisu zapisanego przy
164 // podstawie 10
165 // Operator ten wymaga implementacji =(int)
166    BigNum & operator=(string a) {
167       int s = a.length();
168       *this = 0;
169       Res(len = s / BD + 1);
170       REP(x, s) cyf[(s - x - 1) / BD] = 10 * cyf[(s - x - 1) / BD] + a[x] - '0';
171       REDUCE();
172       return *this;
173    }
174 // Funkcja wypisuje wartość liczby BigNum zapisanej przy podstawie 10
175    void write() const {
176       printf("%d", int (cyf[len - 1]));
177       FORD(x, len - 2, 0) printf("%0*d", BD, int (cyf[x]));
178    }
179 // Funkcja wypisuje do przekazanego bufora wartość liczby zapisanej przy
180 // podstawie 10
181    void write(char *buf) const {
182       int p = sprintf(buf, "%d", int (cyf[len - 1]));
183       FORD(x, len - 2, 0) p += sprintf(buf + p, "%0*d", BD, int (cyf[x]));
184    }
185 // Funkcja zwraca wektor cyfr liczby zapisanej przy podstawie pod. Funkcja ta
186 // wymaga implementacji /=(int), =(BigNum)
187    VI write(int pod) const {
188       VI w;
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
177       BigNum v;
178       v = *this;
179       while (v.len > 1 || v.cyf[0]) w.PB(v /= pod);
180       return w;
181    }
// Operator przesunięcia w prawo o n cyfr
182    BigNum & operator>>=(int n) {
183       if (n >= len) n = len;
184       REP(x, len - n) cyf[x] = cyf[x + n];
185       FOR(x, len - n, n) cyf[x] = 0;
186       len -= n;
187       if (len == 0) len = 1;
188       return *this;
189    }
// Operator przesunięcia w lewo
190    BigNum & operator<<=(int n) {
191       if (cyf[0] == 0 && len == 1) return *this;
192       Res(len + n);
193       FORD(x, len - 1, 0) cyf[x + n] = cyf[x];
194       REP(x, n) cyf[x] = 0;
195       len += n;
196       return *this;
197    }
// Funkcja wyznaczająca pierwiastek całkowity z liczby
// Funkcja ta wymaga implementacji <(BigNum), +=(BigNum), *=(BigNum),
// <<=(int), +(BigNum), *(BigNum), <<(int)
198    BigNum sqrt() {
199       int n = (len + 1) / 2;
200       BigNum a(0, n), sq;
201       FORD(i, n - 1, 0) {
202          int l = 0, r = BASE - 1;
203          while (l < r) {
204             int m = (l + r + 1) / 2;
205             if (*this < sq + (a * 2 * m << i) + (BigNum(m) * m << 2 * i)) r = m -
1;
206             else l = m;
207          }
208          sq += (a * 2 * l << i) + (BigNum(l) * l << 2 * i);
209          a.cyf[i] = l;
210          a.len = n;
211       }
212       return a;
213    }
// Makra pozwalające na skrócenie zapisu nagłówków poniższych operatorów
214 #define OPER3(op) BigNum operator op(const BigNum &a) \
215 const {BigNum w=*this; w op ## = a; return w; }
216 #define OPER4(op) BigNum operator op(int a) \
217 {BigNum w = *this; w op ## = a; return w; }
```

Listing 5.28: (c.d. listingu z poprzedniej strony)

```
// Operator wymaga implementacji +=(BigNum)
218    OPER3(+);
// Operator wymaga implementacji -=(BigNum)
219    OPER3(-);
// Operator wymaga implementacji *=(BigNum)
220    OPER3(*);
// Operator wymaga implementacji <(BigNum), +=(BigNum), *=(BigNum),
// /=(BigNum), +(BigNum), *(BigNum), <<(int)
221    OPER3(/);
// Operator wymaga implementacji <(BigNum), +=(BigNum), -=(BigNum),
// *=(BigNum), /=(BigNum), %=(BigNum), +(BigNum), *(BigNum)
222    OPER3(%);
// Operator wymaga implementacji <<=(int)
223    OPER4(<<);
// Operator wymaga implementacji >>=(int)
224    OPER4(>>);
225 };
```

Listing 5.29: Przykład wykorzystania arytmetyki wielkich liczb

```
a = 348732498327423984324198321740932174
b = 765372810192887283823287914321 4
a+b = 348740152055525913197036554620075388
a-b = 348724844599322055451360088861788960
a*b = 266910372250446859326893728437546561477829181520333480555480636723 6
a/b = 45563
sqrt(a) = 590535772267374150
```

Listing 5.30: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.29. Pełny kod źródłowy programu znajduje się w pliku `bignum.cpp`

```
01 int main() {
02    BigNum a, b;
03    string ta, tb;
// Wczytaj tekst reprezentujący dwie wielkie liczby
04    cin >> ta >> tb;
// Przekonwertuj tekst na liczby oraz wykonaj podstawowe operacje
05    a = ta;
06    b = tb;
07    cout << "a = ";
08    a.write();
09    cout << endl << "b = ";
10    b.write();
11    cout << endl << "a+b = ";
12    (a + b).write();
13    cout << endl << "a-b = ";
14    (a - b).write();
15    cout << endl << "a*b = ";
```

Listing 5.30: (c.d. listingu z poprzedniej strony)

```
16    (a * b).write();
17    cout << endl << "a/b = ";
18    (a / b).write();
19    cout << endl << "sqrt(a) = ";
20    a.sqrt().write();
21    return 0;
22 }
```

Arytmetyka wielkich liczb realizowana przez strukturę `BigNum` pozwala na wykonywanie operacji w ramach zbioru liczb naturalnych. W zdecydowanej większości zadań arytmetyka taka jest wystarczająca, jednak czasem pojawia się potrzeba wykonywania operacji na zbiorze liczb całkowitych. W tym celu została stworzona struktura `Integer`, której implementacja wykorzystuje strukturę `BigNum`. Sposób działania operatorów jest w zasadzie taki sam jak w przypadku struktury `BigNum`. Kod źródłowy przedstawiony jest na listingu 5.31.

Listing 5.31: Implementacja struktury `Integer`

```
// Implementacja liczb całkowitych bazująca na typie BigNum
01 struct Integer {
02    BigNum x;
// Znak liczby (-1 dla liczby ujemnej, 0 dla zera, 1 dla liczby
// dodatniej)
03    int Sgn;
// Konstruktor, przyjmujący jako parametry wartość bezwzględną oraz znak
// tworzonej liczby
04    Integer(const BigNum & a, int s = 1) {
05       x = a;
06       Sgn = (a == BigNum(0)) ? 0 : !s ? 1 : s;
07    }
// Konstruktor tworzący zmienną równą liczbie a
08    Integer(int a = 0) {
09       x = BigNum(abs(a));
10       Sgn = sgn(a);
11    }
// Operator zwraca liczbę o przeciwnym znaku
12    Integer operator-() const {
13       return Integer(x, -Sgn);
14    }
// Operatory porównawcze
15    bool operator<(const Integer & b) const {
16       if (Sgn != b.Sgn) return Sgn < b.Sgn;
17       return (Sgn == -1) ? b.x < x : x < b.x;
18    }
19    bool operator==(const Integer & b) const {
20       return Sgn == b.Sgn && x == b.x;
21    }
// Makra pozwalające na skrócenie zapisu nagłówków operatorów
22 #define OPER5(op) Integer operator op (const Integer &b) const
```

Listing 5.31: (c.d. listingu z poprzedniej strony)

```
#define OPER6(op) Integer &operator op ## = (const Integer &b) \
{return *this = *this op b;}
// Operator +(Integer)
   OPER5(+) {
      if (Sgn == -1) return -(-(*this) + (-b));
      if (b.Sgn >= 0) return Integer(x + b.x, min(1, Sgn + b.Sgn));
      if (x < b.x) return Integer(b.x - x, -1);
      return Integer(x - b.x, x > b.x);
   }
// Operator -(Integer), wykorzystuje +(Integer)
   OPER5(-) {
      return *this + (-b);
   }
// Operator *(Integer)
   OPER5(*) {
      return Integer(x * b.x, Sgn * b.Sgn);
   }
// Operator /(Integer)
   OPER5(/) {
      return Integer(x / b.x, Sgn * b.Sgn);
   }
// Operator %(Integer), wykorzystuje -(Integer)
   OPER5(%) {
      return Sgn == -1 ? Integer((b.x - (x % b.x)) % b.x) : Integer(x % b.x);
   }
// Operator +=(Integer), wykorzystuje +(Integer)
   OPER6(+)
// Operator -=(Integer), wykorzystuje -(Integer)
   OPER6(-)
// Operator *=(Integer), wykorzystuje *(Integer)
   OPER6(*)
// Operator /=(Integer), wykorzystuje /(Integer)
   OPER6(/)
// Operator %=(Integer), wykorzystuje %(Integer)
   OPER6(%)
// Funkcja wypisuje liczbę Integer zapisaną przy podstawie 10 na
// standardowe wyjście
   void write() const {
      if (Sgn == -1) printf("-");
      x.write();
   }
};
```

Dopełnieniem do przedstawionych dotychczas arytmetyk wielkich liczb jest struktura `Frac`, która umożliwia wykonywanie operacji na liczbach wymiernych postaci $\frac{p}{q}$. Struktura ta zawiera implementację najczęściej wykorzystywanych operatorów. Kod źródłowy znajduje się na listingu 5.32.

Listing 5.32: Implementacja struktury `Frac`

```
// Funkcja wyznaczająca największy wspólny dzielnik dwóch liczb.
// Jest ona używana do skracania ułamków
BigNum NatGCD(const BigNum &a, const BigNum& b) {
   return b == BigNum(0) ? a : NatGCD(b, a % b);
}
// Implementacja liczb ułamkowych, bazująca na typie Integer oraz BigNum
struct Frac {
// Licznik a oraz mianownik b ułamka
   Integer a, b;
// Konstruktor typu Frac, tworzy liczbę o zadanym liczniku i mianowniku
   Frac(const Integer &aa = 0,const Integer &bb = 1) {
      a = aa;
      b = bb;
      if(b < 0) {
         a = -a;
         b = -b;
      }
      Integer d = Integer(NatGCD(aa.x, bb.x));
      a /= d;
      b /= d;
   }
// Operatory porównawcze
   bool operator<(const Frac &x) const {
      return a*x.b < x.a*b;
   }
   bool operator==(const Frac &x) const {
      return a == x.a && b == x.b;
   }
// Makra pozwalające na skrócenie zapisu nagłówków operatorów
#define OPER7(op) Frac operator op (const Frac &x) const
#define OPER8(op) Frac &operator op ## =(const Frac &b) \
{return *this = *this op b;}
// Operator +(Frac)
   OPER7(+) {
      return Frac(a * x.b + b * x.a, b * x.b);
   }
// Operator -(Frac)
   OPER7(-) {
      return Frac(a * x.b - b * x.a, b * x.b);
   }
// Operator *(Frac)
   OPER7(*) {
      return Frac(a * x.a, b * x.b);
   }
// Operator /(Frac)
   OPER7(/) {
      return Frac(a * x.b, b * x.a);
```

Listing 5.32: (c.d. listingu z poprzedniej strony)

```
37    }
// Operator +=(Frac), wykorzystuje +(Integer)
38    OPER8(+)
// Operator -=(Frac), wykorzystuje -(Integer)
39    OPER8(-)
// Operator *=(Frac), wykorzystuje *(Integer)
40    OPER8(*)
// Operator /=(Frac), wykorzystuje /(Integer)
41    OPER8(/)
// Funkcja zwraca prawdę, jeśli dany ułamek jest równy 0
42    bool isZero(){
43       return a==Integer(0);
44    }
// Funkcja wypisująca ułamek postaci licznik/mianownik zapisany przy podstawie
// 10 na standardowe wyjście
45    void write(){
46       a.write();
47       printf("/");
48       b.write();
49    }
50 };
```

Listing 5.33: Przykład użycia arytmetyki ułamkowej

```
a = 2/3
b = 15/28
a+b = 101/84
a-b = 11/84
a*b = 5/14
a/b = 56/45
```

Listing 5.34: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.33. Pełny kod źródłowy programu znajduje się w pliku `frac.cpp`

```
01 int main() {
// Skonstruuj dwie liczby typu Frac
02    Frac a(10, 15);
03    Frac b(30, 56);
04    cout << "a = ";
05    a.write();
06    cout << endl << "b = ";
07    b.write();
08    cout << endl << "a+b = ";
09    (a + b).write();
10    cout << endl << "a-b = ";
11    (a - b).write();
12    cout << endl << "a*b = ";
13    (a * b).write();
```

Listing 5.34: (c.d. listingu z poprzedniej strony)

```
14    cout << endl << "a/b = ";
15    (a / b).write();
16    return 0;
17 }
```

W rozdziale dotyczącym sprawdzania pierwszości liczb przedstawiliśmy implementację testu Millera-Rabina, który dla dowolnej liczby naturalnej nie większej od 2 000 000 000 odpowiada, czy liczba ta jest pierwsza. Istnieje możliwość zaadaptowania tego algorytmu w celu sprawdzenia, czy dana wielka liczba jest pierwsza. Jedyna różnica w implementacji polega na zmianie liczby wykonywanych testów (w przypadku liczb z przedziału $[1 \ldots 2\,000\,000\,000]$ wystarczyły tylko 4 — 2, 3, 5 oraz 7. Prawdopodobieństwo uzyskania błędnej odpowiedzi w przypadku testowania liczby złożonej, przy wykonywaniu l testów wynosi $(\frac{1}{4})^l$. Implementacja funkcji **bool** `IsBPrime(`**const** `BigNum&, int)` przedstawiona na listingu 5.35 realizuje ten test pierwszości.

Listing 5.35: Implementacja funkcji **bool** `IsBPrime(`**const** `BigNum&, int)`

```
// Funkcja przeprowadza test Millera-Rabina dla liczby n przy podstawie x
01 bool PBWit(BigNum x, BigNum n) {
02    if (x >= n) return 0;
03    BigNum d = 1, y, l = n - 1;
04    int t = 0;
05    bool st = 0;
06    while ((l % 2) == 0) {
07       ++t;
08       l /= 2;
09    }
10    for (; l > 0 || t--; l /= 2) {
11       if ((l % 2) == 1) d = (d * x) % n;
12       else if (!st && l == 0) {
13          st = 1;
14          x = d;
15       }
16       y = (x * x) % n;
17       if (y == 1 && x != 1 && x != n - 1) return 1;
18       x = y;
19    }
20    return x != 1;
21 }
// Funkcja sprawdza, czy dana liczba typu BigNum jest pierwsza. W tym celu
// wykonuje t razy test Millera-Rabina
22 bool IsBPrime(const BigNum & x, int t) {
23    if (x < 2) return 0;
24    REP(i, t) if (PBWit(max(rand(), 2), x)) return 0;
25    return 1;
26 }
```

Listing 5.36: Wyznaczenie wszystkich liczb pierwszych w przedziale [1 000 000 000 000 000 000 000 . . . 1 000 000 000 000 000 000 300] przy użyciu funkcji **bool** `IsBPrime(const BigNum&, int)`

```
10000000000000000117
10000000000000000193
10000000000000000213
10000000000000000217
10000000000000000289
```

Listing 5.37: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 5.36. Pełny kod źródłowy programu znajduje się w pliku `isbprime.cpp`

```
01 int main() {
02    BigNum b, e;
03    string tb, te;
// Wczytaj zakres testowanych liczb
04    cin >> tb >> te;
05    b = tb;
06    e = te;
// Dla każdej liczby z zakresu sprawdź, czy jest pierwsza
07    while (b < e) {
08       if (IsBPrime(b, 20)) {
09          b.write();
10          cout << endl;
11       }
12       b += 1;
13    }
14    return 0;
15 }
```

Zadanie: Łańcuch

Pochodzenie: VIII Olimpiada Informatyczna

Rozwiązanie: `chain.cpp`

Bajtocja nie zawsze była państwem demokratycznym. W jej historii były również czarne karty. Pewnego razu, generał Bajtelski — przywódca junty żelazną ręką rządzącej Bajtocją — postanowił zakończyć stan wojenny, trwający od momentu przejęcia władzy, i zwolnić więzionych działaczy opozycji. Nie chciał jednak uwolnić przywódcy opozycji Bajtazara. Postanowił przykuć go do murów więzienia za pomocą bajtockiego łańcucha. Bajtocki łańcuch składa się z połączonych ze sobą ogniw oraz przymocowanego do muru pręta. Choć ogniwa nie są połączone z prętem, to bardzo trudno jest je z niego zdjąć.

— Generale, czemuś mnie przykuł do murów więzienia, miast uwolnić, jako to uczyniłeś z moimi kamratami! — wołał Bajtazar.

— Ależ Bajtazarze, wszak nie jesteś przykuty i z pewnością potrafisz sam zdjąć trzymające cię ogniwa z pręta wystającego z murów więzienia. — przewrotnie stwierdził generał Bajtelski, po czym dodał – Uporaj się z tym jednak przed godziną cyfracyjną i nie dzwoń łańcuchami po nocy, gdyż w przeciwnym przypadku będę zmuszony wezwać funkcjonariuszy Cyfronicji Obywatelskiej.

Pomóż Bajtazarowi! Ponumerujmy kolejne ogniwa łańcucha liczbami $1, 2, ..., n$. Ogniwa te możemy zakładać i zdejmować z pręta zgodnie z następującymi zasadami:

- jednym ruchem możemy zdjąć lub założyć na pręt tylko jedno ogniwo,
- ogniwo nr 1 można zawsze zdjąć lub założyć na pręt,
- jeżeli ogniwa o numerach $1, \ldots, k-1$ (dla $1 \leqslant k \leqslant n$) są zdjęte z pręta, a ogniwo nr k jest założone, to możemy zdjąć lub założyć ogniwo nr $k+1$.

Zadanie

Napisz program, który:

- wczyta opis bajtockiego łańcucha,
- obliczy minimalną liczbę ruchów, które należy wykonać, aby zdjąć wszystkie ogniwa bajtockiego łańcucha z pręta,
- wypisze wynik.

Wejście

W pierwszym wierszu zapisano jedną dodatnią liczbę całkowitą $n, 1 \leqslant n \leqslant 10,000$. W drugim wierszu zapisano n liczb całkowitych $o_1, o_2, \ldots, o_n \in 0, 1$, pooddzielanych pojedynczymi odstępami. Jeśli $o_i = 1$, to ogniwo nr i jest założone na pręt, a jeśli $o_i = 0$, to jest z niego zdjęte.

Wyjście

Twój program powinien wypisać jedną liczbę całkowitą, równą minimalnej liczbie ruchów potrzebnych do zdjęcia wszystkich ogniw bajtockiego łańcucha z pręta.

Przykład

Dla następującego wejścia:

```
4
1 0 1 0
```

Poprawnym rozwiązaniem jest:

```
6
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 424	acm.uva.es - zadanie 10183	acm.uva.es - zadanie 623
acm.uva.es - zadanie 10018	acm.uva.es - zadanie 495	spoj.sphere.pl - zadanie 31
acm.uva.es - zadanie 324	spoj.sphere.pl - zadanie 54	spoj.sphere.pl - zadanie 291
acm.uva.es - zadanie 10035	spoj.sphere.pl - zadanie 328	spoj.sphere.pl - zadanie 279
spoj.sphere.pl - zadanie 123	acm.sgu.ru - zadanie 111	spoj.sphere.pl - zadanie 115
spoj.sphere.pl - zadanie 362	acm.sgu.ru - zadanie 193	acm.sgu.ru - zadanie 247
acm.sgu.ru - zadanie 112	acm.sgu.ru - zadanie 197	
spoj.sphere.pl - zadanie 94	acm.sgu.ru - zadanie 299	

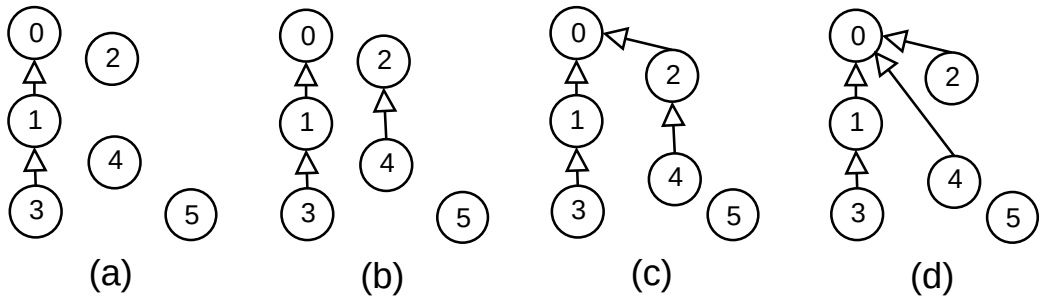
Rozdział 6

Struktury danych

Literatura
[WDA] - 3
[ASP] - 4

Podstawą pomyślnego rozwiązania każdego zadania jest wymyślenie odpowiedniego algorytmu. W przypadku podjęcia dobrej decyzji, rozwiązanie zadania jest już tylko kwestią czasu. Niestety, podczas zawodów, trzeba się liczyć dodatkowo ze ścisłymi ograniczeniami czasowymi. W takiej sytuacji nie tylko algorytm, ale również i sposób implementacji jest niezwykle ważny. Rozwiązanie wielu zadań wymaga użycia różnego rodzaju struktur danych. Zazwyczaj istnieje możliwość wyboru pomiędzy bardziej efektywną metodą lecz trudniejszą do zaimplementowania oraz wolniejszą i prostszą. Bardzo ważną decyzją projektową jest wybór odpowiedniej struktury danych, która jest wystarczająco efektywna, aby rozwiązanie zmieściło się w limitach czasowych przygotowanych przez organizatorów, a jednocześnie jej implementacja była jak najprostsza. Jest to niezwykle ważna umiejętność, którą nabywa się z czasem. Najlepszą decyzją jest zazwyczaj po prostu użycie — o ile to możliwe, gotowej struktury danych z biblioteki STL. Umiejętne wykorzystanie tej biblioteki pozwala skrócić długość implementowanego programu niekiedy i kilkukrotnie.

W niektórych sytuacjach okazuje się jednak, że struktury danych udostępnione przez bibliotekę STL są niewystarczające. W niniejszym rozdziale przedstawimy implementację kilku najczęściej wykorzystywanych podczas zawodów struktur danych, które nie są udostępnione w bibliotece STL.



Rysunek 6.1: **(a)** Reprezentacja struktury zbiorów rozłącznych $\{0, 1, 3\}$, $\{2\}$, $\{4\}$, $\{5\}$. **(b)** Stan struktury danych po wykonaniu operacji złączenia zbiorów $\{2\}$ i $\{4\}$. **(c)** Stan struktury po złączeniu zbiorów $\{0, 1, 3\}$ i $\{2, 4\}$. **(d)** Wyszukanie zbioru, do którego należy element 4 spowodowało skompresowanie ścieżki od elementu 4 do korzenia jego drzewa.

6.1. Struktura danych do reprezentacji zbiorów rozłącznych

Literatura
[WDA] - 22
[ASD] - 4

Załóżmy, że mamy dane n rozłącznych zbiorów jednoelementowych — $\{0\}, \{1\}, \ldots, \{n-1\}$. Chcielibyśmy wykonywać na tych zbiorach dwie operacje: łączenie ze sobą dwóch zbiorów oraz wyznaczanie zbioru, do którego należy dany element $k \in \{0, 1, \ldots, n-1\}$.

Jednym ze sposobów rozwiązania tego zadania jest przypisanie każdej liczbie k numeru zbioru, do którego ona należy (na początku liczbie k przypisywany jest zbiór o numerze k). Sprawdzenie, do którego zbioru należy dany element sprowadza się wtedy do zwrócenia numeru przypisanego zbioru. Złączenie dwóch zbiorów P i Q natomiast można zrealizować poprzez zamianę numeru p wszystkich elementów zbioru P na numer zbioru Q. Widać, że wykonanie złączenia dwóch zbiorów wymaga wyznaczenia wszystkich elementów należących do złączanych zbiorów, zatem operacja ta ma pesymistycznie złożoność liniową — nie jest to zatem rozwiązanie szczególnie efektywne.

Efektywną, a zarazem dość prostą strukturą danych, pozwalającą na wykonywanie powyższych dwóch operacji, jest tzw. struktura Find and Union. Struktura ta reprezentuje zbiory w postaci lasu, w którym wierzchołkami są elementy ze zbioru $\{0, 1, \ldots, n-1\}$. Każdy zbiór jest reprezentowany jako drzewo, w którym każdy element przechowuje numer swojego ojca. Operacja łączenia dwóch zbiorów polega na dodaniu krawędzi między korzeniami drzew reprezentujących łączone zbiory, natomiast operacja wyznaczania zbioru, do którego należy element k — na znalezieniu numeru korzenia drzewa, do którego należy element k. W ten sposób dwa elementy należą do tego samego zbioru, jeśli mają wspólny korzeń w drzewie.

Zamortyzowana złożoność wykonania m operacji na tej strukturze danych to $O(m * log^*(m))$, gdzie log^* jest odwrotnością funkcji Ackermana. Funkcja ta bardzo wolno rośnie (istotnie wolniej od logarytmu) — w praktycznych zastosowaniach można zakładać zatem, że złożoność wykonania m operacji to po prostu $O(m)$). Uzyskanie takiej złożoności jest możliwe dzięki wykorzystaniu dwóch metod:

- podczas wyszukiwania korzenia drzewa, do którego należy element k, ścieżka do korzenia ulega kompresji — wszystkie wierzchołki, leżące na ścieżce między k a jego korzeniem, zostają bezpośrednio połączone z korzeniem. W ten sposób wykonywanie operacji wyszukiwania powoduje skracanie ścieżek w drzewach, dzięki czemu kolejne wyszukiwania będą wykonywać się szybciej.

- operacja łączenia dwóch drzew reprezentujących zbiory realizowana jest według rang,

co sprowadza się do przyłączania mniejszego drzewa do korzenia drzewa większego, a nie na odwrót. Takie podejście minimalizuje sumaryczną długość powstających ścieżek.

Przykładowy sposób realizacji łączenia i wyszukiwania zbiorów został przedstawiony na rysunku 6.1.

Implementacja struktury FAU jest przedstawiona na listingu 6.1. Operacja Union(x,y) powoduje złączenie zbiorów zawierających elementy x oraz y, natomiast operacja Find(x) na wyznacza numer zbioru, do którego należy element x. Na listingu 6.2 przedstawiony jest sposób działania tej struktury dla 6-elementowago zbioru oraz sekwencji przykładowych operacji Find i Union.

Listing 6.1: Implementacja struktury FAU

```
// Struktura danych do reprezentacji zbiorów rozłącznych
struct FAU {
   int *p, *w;
// Konstruktor tworzący reprezentację n jednoelementowych zbiorów rozłącznych
   FAU(int n) : p(new int[n]), w(new int[n]) {
      REP(x, n) p[x] = w[x] = -1;
   }
// Destruktor zwalniający wykorzystywaną pamięć
   ~FAU() {
      delete[]p;
      delete[]w;
   }
// Funkcja zwraca numer reprezentanta zbioru, do którego należy element x
   int Find(int x) {
      return (p[x] < 0) ? x : p[x] = Find(p[x]);
   }
// Funkcja łączy zbiory zawierające elementy x oraz y
   void Union(int x, int y) {
      if ((x = Find(x)) == (y = Find(y))) return;
      if (w[x] > w[y]) p[y] = x;
      else p[x] = y;
      if (w[x] == w[y]) w[y]++;
   }
};
```

Listing 6.2: Przykład działania struktury FAU.

```
Zlaczenie zbiorow zawierajacych elementy 0 i 1
Find(0) = 1  Find(1) = 1  Find(2) = 2  Find(3) = 3  Find(4) = 4  Find(5) = 5
Zlaczenie zbiorow zawierajacych elementy 3 i 4
Find(0) = 1  Find(1) = 1  Find(2) = 2  Find(3) = 4  Find(4) = 4  Find(5) = 5
Zlaczenie zbiorow zawierajacych elementy 1 i 5
Find(0) = 1  Find(1) = 1  Find(2) = 2  Find(3) = 4  Find(4) = 4  Find(5) = 1
Zlaczenie zbiorow zawierajacych elementy 0 i 5
Find(0) = 1  Find(1) = 1  Find(2) = 2  Find(3) = 4  Find(4) = 4  Find(5) = 1
```

Listing 6.3: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.2. Pełny kod źródłowy programu znajduje się w pliku `fau.cpp`

```
01 int main() {
02   int n, m, e1, e2;
// Wczytaj liczbę elementów oraz operacji do wykonania
03   cin >> n >> m;
04   FAU fau(n);
05   REP(x, m) {
// Wczytaj numery dwóch elementów oraz złącz zbiory je zawierające
06     cin >> e1 >> e2;
07     fau.Union(e1, e2);
08     cout << "Zlaczenie zbiorow zawierajacych elementy " <<
09     e1 << " i " << e2 << endl;
10     REP(y, n) cout << "Find(" << y << ") = " << fau.Find(y) << " ";
11     cout << endl;
12   }
13   return 0;
14 }
```

Zadanie: Małpki

Pochodzenie: X Olimpiada Informatyczna

Rozwiązanie: `monkey.cpp`

Na drzewie wisi n małpek ponumerowanych od 1 do n. Małpka z nr 1 trzyma się gałęzi ogonkiem. Pozostałe małpki albo są trzymane przez inne małpki, albo trzymają się innych małpek, albo jedno i drugie równocześnie. Każda małpka ma dwie przednie łapki, każdą może trzymać co najwyżej jedną inną małpkę (za ogon). Rozpoczynając od chwili 0, co sekundę jedna z małpek puszcza jedną łapkę. W ten sposób niektóre małpki spadają na ziemię, gdzie dalej mogą puszczać łapki (czas spadania małpek jest pomijalnie mały).

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis tego, w jaki sposób małpki trzymają się oraz w jakiej kolejności puszczają łapki,
- obliczy dla każdej małpki, kiedy spadnie ona na ziemię,
- wypisze wynik na standardowe wyjście.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie dodatnie liczby całkowite n i m ($1 \leqslant n \leqslant 200\,000$, $1 \leqslant m \leqslant 400\,000$). Liczba n oznacza liczbę małpek, a liczba m czas (w sekundach) przez jaki obserwujemy małpki. Kolejne n wierszy zawiera opis sytuacji początkowej. W wierszu $k+1$ ($1 \leqslant k \leqslant n$) znajdują się dwie liczby całkowite oznaczające numery małpek trzymanych przez małpkę nr k. Pierwsza z tych liczb to numer małpki trzymanej lewą łapką, a druga — prawą. Liczba -1 oznacza, że małpka ma wolną łapkę. Kolejne m wierszy opisuje wyniki obserwacji małpek. W i-tym spośród tych wierszy ($1 \leqslant i \leqslant m$) znajdują się dwie

liczby całkowite. Pierwsza z nich oznacza numer małpki, a druga numer łapki (1 — lewa, 2 — prawa), którą małpka puszcza w chwili $i - 1$

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie n liczb całkowitych, po jednej w wierszu. Liczba w wierszu i powinna oznaczać chwilę, w której małpka nr i spadła na ziemię, lub być równa -1, jeśli małpka nie spadła na ziemię w trakcie obserwacji.

Przykład

Dla następującego wejścia:

```
3 2
−1 3
3 −1
1 2
1 2
3 1
```

Poprawnym rozwiązaniem jest:

```
−1
1
1
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10583	spoj.sphere.pl - zadanie 116	spoj.sphere.pl - zadanie 264
acm.uva.es - zadanie 10608	acm.sgu.ru - zadanie 174	spoj.sphere.pl - zadanie 188
	acm.uva.es - zadanie 793	acm.uva.es - zadanie 10158

6.2. Drzewa poszukiwań binarnych

Literatura
[ASD] - 3.3
[SZP] - 2.3
[ASP] - 4.4
[MD] - 6.4

W niniejszym rozdziale zajmiemy się realizacją struktur danych, pozwalających na wykonywanie operacji dodawania oraz usuwania elementów, na których określony jest porządek liniowy. W zależności od rodzaju przedstawianej struktury, dostępne będą również inne operacje, takie jak wyznaczanie liczby elementów o wartościach w zadanym przedziale czy wyszukiwanie k-tego najmniejszego elementu. Zaprezentowana również zostanie specjalna „wzbogacalna" struktura, pozwalająca na tworzenie własnych operacji w prosty sposób. Na wstępie należy omówić sposób reprezentacji tych struktur danych — wszystkie omawiane struktury są pewnego rodzaju drzewami poszukiwań binarnych.

Definicja 6.2.1 Drzewo poszukiwań binarnych *jest to takie ukorzenione drzewo, które spełnia następujące warunki:*

- *każdy węzeł w drzewie ma co najwyżej* 2 *synów — lewego oraz prawego. Węzły drzewa nie posiadające synów nazywane są liśćmi,*
- *każdy węzeł posiada przypisany mu element. Na zbiorze elementów określony jest porządek liniowy,*
- *dla każdego węzła p drzewa zawierającego element v_p, jeśli ma on lewego syna l z elementem v_l, to $v_l < v_p$,*

- *dla każdego węzła p drzewa zawierającego element v_p, jeśli ma on prawego syna r z elementem v_r, to $v_r > v_p$*

Przykład drzewa poszukiwań binarnych przedstawiony jest na rysunku 6.2.a. Drzewa tego typu wykorzystuje się do konstrukcji różnych struktur danych, jak choćby słowników. Wstawianie elementu do słownika odbywa się poprzez dodanie do drzewa binarnych poszukiwań nowego węzła zawierającego dodawany element (miejsce umieszczenia nowego węzła w drzewie musi zostać tak dobrane, aby zachować zgodność z definicją drzew poszukiwań binarnych). Sprawdzenie przynależności elementu jest równoważne przeszukaniu drzewa od korzenia w poszukiwaniu odpowiedniego węzła.

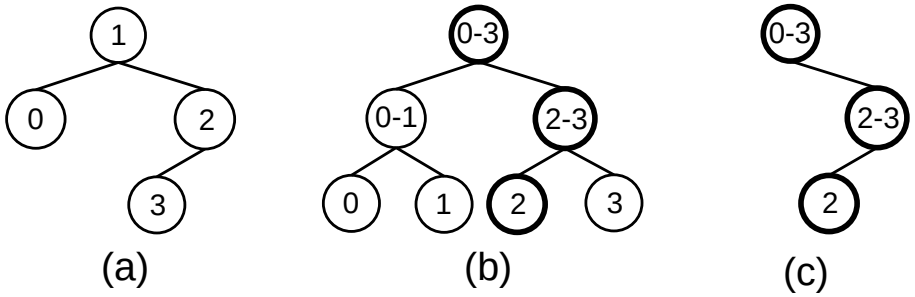
Wprowadzenie takiej struktury danych, jaką są drzewa binarnych poszukiwań, ma na celu uzyskanie efektywnych operacji. Czas potrzebny na wykonanie operacji na węźle jest wprost proporcjonalny do jego głębokości (odległości od korzenia). W przypadku, gdy drzewo jest zrównoważone (głębokość najpłytszego liścia w drzewie nie różni się wiele od głębokości najgłębszego liścia), to odległość każdego węzła od korzenia jest logarytmiczna w stosunku do liczby węzłów w całym drzewie. Wprawdzie w przypadku wykonywania losowych operacji na drzewie binarnych poszukiwań, pozostaje ono zrównoważone, to wykonanie n operacji wstawienia elementu do drzewa w kolejności rosnących wartości spowoduje powstanie w drzewie łańcucha o długości n (kolejno wstawiane elementy będą dodawane jako prawy syn poprzedniego). Istnieje możliwość zabezpieczenia się przed takimi sytuacjami poprzez implementację zrównoważonych drzew binarnych (takich jak drzewa czarno-czerwone, czy AVL), które po wykonaniu operacji modyfikującej strukturę drzewa wykonują reorganizację układu węzłów, mającą na celu wyrównanie głębokości liści. Implementacja tych struktur danych jest jednak czasochłonna, co podczas zawodów jest bardzo dużą wadą.

Na szczęście istnieją inne, bardzo proste sposoby zapewnienia szybkiego czasu działania wszystkich operacji. Jest to możliwe dzięki zmodyfikowaniu postaci drzew poszukiwań binarnych, które będziemy dalej nazywali statycznymi drzewami binarnych poszukiwań. Przykład takiego drzewa jest przedstawiony na rysunku 6.2.b. Różnica w stosunku do zwykłych drzew polega na tym, że podczas konstrukcji drzewa statycznego podaje się liczbę węzłów, które powinny znaleźć się w takim drzewie. Narzuca to pewne ograniczenia na dziedzinę, z której pochodzą wstawiane elementy — musi być ona skończona. W drzewach statycznych właściwe węzły umieszczone są tylko w liściach — wewnętrzne węzły drzewa reprezentują natomiast całe zbiory elementów z ich poddrzew. Takie podejście ułatwia realizację różnych dodatkowych operacji — węzły wewnętrzne mogą przechowywać specjalne informacje, utrzymywane w celu umożliwienia efektywnej realizacji tych operacji. Dodanie nowego elementu do drzewa statycznego polega na zaznaczeniu odpowiedniego liścia jako aktywnego oraz zaktualizowaniu wartości w węzłach wewnętrznych drzewa, leżących na ścieżce od dodawanego liścia do korzenia. Podobnie realizowana jest operacja usuwania elementów.

Drzewa statyczne, oprócz ograniczenia na wielkość dziedziny mają jeszcze jedną wadę — zużycie pamięci. Nawet jeśli liczba elementów wstawionych do drzewa jest mała, to i tak zużycie pamięci jest liniowe ze względu na wielkość dziedziny. Usunięcie tej wady jest możliwe poprzez zastosowanie tzw. drzew statycznych dynamicznie alokowanych, których struktura jest taka sama jak drzew statycznych, jednak przydzielają one pamięć na węzły dopiero wtedy, gdy są one potrzebne. Przykład takiego drzewa znajduje się na rysunku 6.2.c.

W tym rozdziale przedstawimy implementację czterech prostych struktur danych, bazujących na koncepcji statycznych drzew binarnych poszukiwań:

- drzewa maksimów — pozwalają na przypisywanie dodatkowych wartości elementom oraz wyznaczanie największej wartości w obrębie elementów z zadanego przedziału,



Rysunek 6.2: **(a)** Drzewo binarnych poszukiwań zawierające cztery elementy — 0, 1, 2 i 3. **(b)** Statyczne drzewo binarnych poszukiwań skonstruowane dla czterech elementów. Aktualnie w drzewie tym jest wstawiony (aktywny) tylko jeden element — 2. **(c)** Statyczne drzewo binarne dynamicznie alokowane. Struktura takiego drzewa jest z góry określona, lecz nieaktywnym węzłom nie jest przydzielana pamięć.

- drzewa licznikowe — pozwalają na przypisywanie dodatkowych wartości elementom oraz wyznaczanie sumy wartości w obrębie elementów z zadanego przedziału,
- drzewa pozycyjne — pozwalają na wyznaczanie statystyk pozycyjnych (znajdowanie k-tego najmniejszego elementu),
- drzewa pokryciowe — pozwalają na dodawanie oraz usuwanie przedziałów liczb (które można również traktować jako odcinki) oraz obliczanie powierzchni przez nie pokrytej na zadanym obszarze.

Na przykładzie drzew pokryciowych przedstawimy również modyfikację drzew statycznych w drzewa dynamicznie alokowane. Pod koniec rozdziału przedstawimy implementację klasycznych drzew binarnych, które wspomagają dodatkowe wzbogacanie (możliwość dodawania własnych operacji). Drzewa te nie są drzewami zrównoważonymi, co może spowodować, że dla specyficznych scenariuszy, czas wykonywania pojedynczych operacji może być liniowy. Pokażemy również realizację randomizowanych drzew binarnych, które dodatkowo losowo modyfikują swoją strukturę; stanowią one bardzo prostą implementację zrównoważonej struktury danych (takich jak drzewa czarno-czerwone czy AVL).

6.2.1. Drzewa maksimów

Drzewa maksimów pozwalają na dodawanie i usuwanie elementów, którym przypisane są dodatkowe wartości ze zbioru liczb naturalnych. Pozwalają one również na wyznaczanie największej wartości na zadanym spójnym przedziale elementów. Węzły wewnętrzne drzewa maksimów zawierają informację na temat największej wartości umieszczonej w ich poddrzewach, co pozwala na efektywną realizację zapytań.

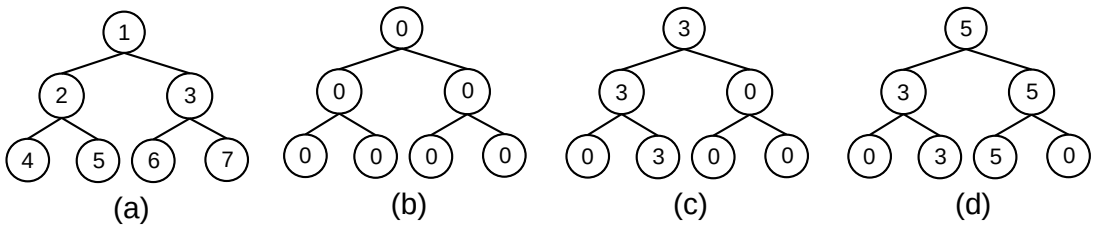
Reprezentacja statycznego drzewa poszukiwań binarnych jest realizowana jako zwykła tablica liczb. Taka reprezentacja może zostać wykorzystana ze względu na fakt, iż drzewa statyczne są pełne i nie zmieniają swej struktury w czasie. Element 1 tablicy reprezentuje korzeń drzewa, 2 to lewy syn korzenia, 3 to syn prawy ... Wyznaczanie lewego i prawego syna dla węzła o numerze k odbywa się w bardzo prosty sposób — lewy syn ma przypisany numer $2 * k$, natomiast prawy — $2 * k + 1$.

Implementacja tych drzew zrealizowana jest jako struktura `MaxTree`, której kod źródłowy przedstawiony jest na listingu 6.4. Funkcja **void** `Set(`**int, int**`)` pozwala na zmianę wartości

elementów, natomiast **int** Find(**int**,**int**) wyznacza maksymalną wartość wśród elementów na zadanym przedziale.

Listing 6.4: Implementacja struktury MaxTree

```
// Drzewo MaxTree umożliwia dodawanie elementów z przypisaną im wartością oraz
// wyszukiwanie największej wartości na dowolnym spójnym przedziale elementów
01 struct MaxTree {
02   int *el, s;
// Konstruktor przyjmuje jako parametr wysokość konstruowanego drzewa (dziedzina
// elementów to [0..2^size-1])
03   MaxTree(int size) {
04     el = new int[2 * (s = 1 << size)];
05     REP(x, 2 * s) el[x] = 0;
06   }
// Destruktor zwalnia zaalokowaną pamięć
07   ~MaxTree() {
08     delete[]el;
09   }
// Funkcja zmienia wartość elementu p na v
10   void Set(int p, int v) {
// Ustaw wartość elementu p na v, oraz zaktualizuj wierzchołki na
// ścieżce do korzenia, wyliczając dla nich maksimum z ich lewego i prawego
// syna
11     for (p += s, el[p] = v, p >>= 1; p > 0; p >>= 1)
12     el[p] = max(el[p << 1], el[(p << 1) + 1]);
13   }
// Funkcja wyznacza największą wartość na przedziale elementów [p..k]
14   int Find(int p, int k) {
15     int m = -INF;
16     p += s;
17     k += s;
// Przeszukiwanie drzewa rozpoczyna się od liści reprezentujących elementy
// p i k. Dopóki węzeł p jest różny od węzła k...
18     while (p < k) {
// Jeśli przedziały reprezentowane przez aktualne węzły p i k
// zawierają się całkowicie w przeszukiwanym przedziale, to następuje
// aktualizacja wyniku
19       if ((p & 1) == 1) m = max(m, el[p++]);
20       if ((k & 1) == 0) m = max(m, el[k--]);
// Przejdź do ojców węzłów p i k
21       p >>= 1;
22       k >>= 1;
23     }
24     if (p == k) m = max(m, el[p]);
25     return m;
26   }
27 };
```



Rysunek 6.3: **(a)** Sposób numeracji węzłów w drzewach statycznych. **(b)** Puste drzewo maksimów. Kolejne liście (patrząc od lewej strony) reprezentują klucze 0, 1, 2 i 3. **(c)** Stan drzewa po dodaniu elementu 1 o wartości 3. **(d)** Stan drzewa po dodaniu elementu 2 o wartości 5.

Listing 6.5: Przykład wykorzystania struktury `MaxTree` dla elementów z dziedziny $\{0, 1, \ldots, 15\}$

```
Zmiana wartosci elementu 2 na 2
Maksimum na przedziale 0..4 = 2
Maksimum na przedziale 2..2 = 2
Maksimum na przedziale 3..15 = 0
Zmiana wartosci elementu 3 na 10
Maksimum na przedziale 0..2 = 2
Maksimum na przedziale 3..10 = 10
Zmiana wartosci elementu 3 na 3
Maksimum na przedziale 0..15 = 3
```

Listing 6.6: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.5. Pełny kod źródłowy programu znajduje się w pliku `maxtree.cpp`

```
01 int main() {
02    int w1, w2, w3;
// Skonstruuj drzewo maksimów dla elementów [0..15]
03    MaxTree tree(4);
// Wykonaj listę operacji...
04    while(cin >> w1 >> w2 >> w3) {
05       if (w1 == 0) {
// Operacja zmiany wartości klucza...
06          tree.Set(w2, w3);
07          cout << "Zmiana wartosci elementu " << w2 << " na " << w3 << endl;
08       } else
// Operacja wypisania maksimum na przedziale [w2..w3]
09       cout << "Maksimum na przedziale " << w2 << ".." << w3 <<
10          " = " << tree.Find(w2, w3) << endl;
11    }
12    return 0;
13 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 105	acm.uva.es - zadanie 10635	acm.uva.es - zadanie 497
acm.uva.es - zadanie 231	spoj.sphere.pl - zadanie 130	spoj.sphere.pl - zadanie 57

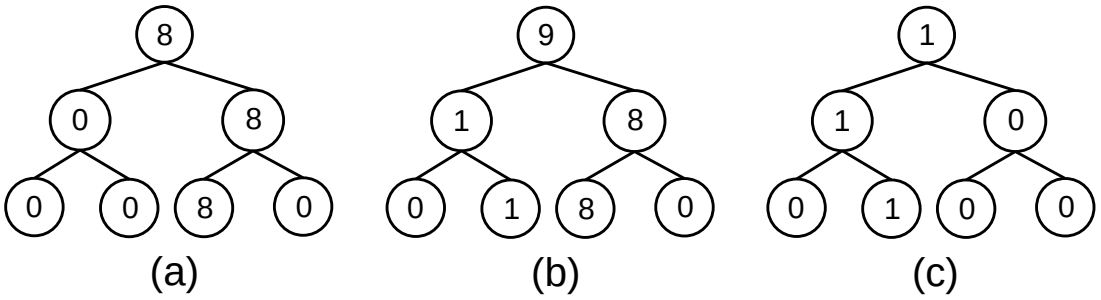
6.2.2. Drzewa licznikowe

Kolejną prezentowaną strukturą danych są drzewa licznikowe. Pozwalają one na przypisywanie elementom wartości ze zbioru liczb całkowitych oraz wyznaczanie sumy wartości na spójnym przedziale elementów. Węzły wewnętrzne drzewa licznikowego przechowują sumę wartości ze swoich poddrzew.

Implementacja struktury `CountTree` przedstawiona jest na listingu 6.7, natomiast na listingu 6.8 przedstawiony jest ich przykładowy sposób użycia.

Listing 6.7: Implementacja struktury `CountTree`

```
// Struktura umożliwia dodawanie elementów z przypisaną wartością oraz sprawdzanie
// sumy wartości na dowolnym przedziale elementów
01 struct CountTree {
02    int *el, s;
// Konstruktor przyjmuje jako parametr wysokość konstruowanego drzewa (dziedzina
// kluczy to [0..2^size-1])
03    CountTree(int size) {
04       el = new int[2 * (s = 1 << size)];
05       REP(x, 2 * s) el[x] = 0;
06    }
// Destruktor zwalnia zaalokowaną pamięć
07    ~CountTree() {
08       delete[]el;
09    }
// Funkcja zmienia wartość klucza p na v
10    void Set(int p, int v) {
// Do wszystkich węzłów na ścieżce do korzenia dodawana jest wartość v
11       for (p += s; p; p >>= 1)
12       el[p] += v;
13    }
// Funkcja wyznacza sumę wartości w spójnym przedziale elementów [p..k]
14    int Find(int p, int k) {
15       int m = 0;
16       p += s;
17       k += s;
18       while (p < k) {
19          if (p & 1) m += el[p++];
20          if (!(k & 1)) m += el[k--];
21          p >>= 1;
22          k >>= 1;
23       }
24       if (p == k) m += el[p];
25       return m;
```



Rysunek 6.4: **(a)** Stan drzewa licznikowego po wstawieniu elementu 2 o wartości 8. **(b)** Zmiana stanu drzewa po dodaniu elementu 1 o wartości 1. **(c)** Stan drzewa po usunięciu elementu 2.

Listing 6.7: (c.d. listingu z poprzedniej strony)

```
26    }
27 };
```

Listing 6.8: Przykład wykorzystania struktury `CountTree` na przykładzie elementów z dziedziny $\{0, 1, \ldots, 15\}$

```
Zmiana wartosci elementu 2 o 2
Suma na przedziale 0..15 = 2
Suma na przedziale 5..10 = 0
Zmiana wartosci elementu 3 o 10
Suma na przedziale 0..15 = 12
Suma na przedziale 0..2 = 2
Zmiana wartosci elementu 3 o 3
Suma na przedziale 0..15 = 15
```

Listing 6.9: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.8. Pełny kod źródłowy programu znajduje się w pliku `counttree.cpp`

```
01 int main() {
02    int w1, w2, w3;
// Skonstruuj drzewo dla dziedziny elementów [0..15]
03    CountTree tree(4);
// Wykonaj listę operacji...
04    while(cin >> w1 >> w2 >> w3) {
05       if (w1 == 0) {
// Operacja zmiany wartości elementu...
06          tree.Set(w2, w3);
07          cout << "Zmiana wartosci elementu " << w2 << " o " << w3 << endl;
08       } else
// Wyznaczenie sumy wartości na przedziale [w2..w3]
09       cout << "Suma na przedziale " << w2 << ".." << w3 <<
10          " = " << tree.Find(w2, w3) << endl;
11    }
12    return 0;
13 }
```

Ćwiczenia

Proste	Średnie	Trudne
spoj.sphere.pl - zadanie 227	acm.uva.es - zadanie 10200	

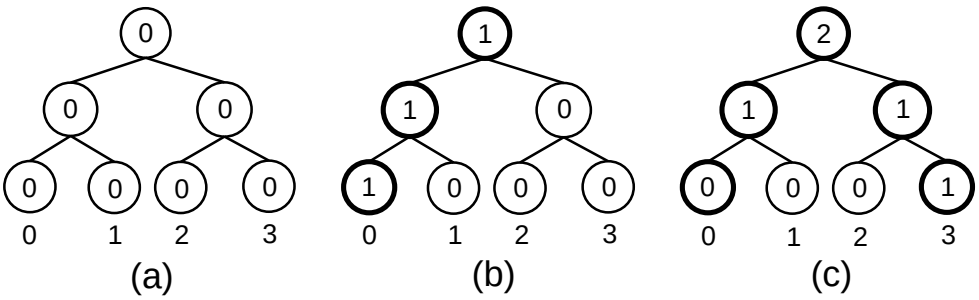
6.2.3. Drzewa pozycyjne

Drzewa pozycyjne pozwalają na dodawanie i usuwanie elementów oraz wyznaczanie statystyk pozycyjnych (k-tego najmniejszego elementu). Węzły wewnętrzne drzewa pozycyjnego przechowują liczbę elementów znajdujących się w ich poddrzewach, dzięki czemu w łatwy sposób można wyznaczać statystyki pozycyjne, przeszukując drzewo od korzenia.

Implementacja struktury `PosTree` przedstawiona jest na listingu 6.10, natomiast na listingu 6.11 przedstawiony jest ich przykładowy sposób użycia.

Listing 6.10: Implementacja struktury `PosTree`

```
// Struktura umożliwia dodawanie elementów i wyznaczanie statystyk pozycyjnych
01 struct PosTree {
02    int *el, s;
// Konstruktor przyjmuje jako parametr wysokość konstruowanego drzewa (dziedzina
// elementów to [0..2^size-1])
03    PosTree(int size) {
04       el = new int[1 << ((s = size) + 1)];
05       REP(x, 1 << (s + 1)) el[x] = 0;
06    }
// Destruktor zwalnia zaalokowaną pamięć
07    ~PosTree() {
08       delete[]el;
09    }
// Funkcja dodaje v wystąpień elementu p (v może być ujemne)
10    void Add(int p, int v) {
// Dla każdego węzła drzewa od liścia p do korzenia, aktualizowana jest
// liczba elementów w poddrzewie
11       for (p = (1 << s) + p; p > 0; p = p >> 1)
12       el[p] += v;
13    }
// Funkcja wyznacza p-tą statystykę pozycyjną
14    int Find(int p) {
// Przeszukiwanie rozpoczynane jest od korzenia drzewa
15       int po = 1;
16       REP(x, s) {
// Następuje przejście do lewego syna aktualnego węzła
17          po <<= 1;
// Jeśli aktualne poddrzewo zawiera mniej elementów niż wynosi numer
// wyszukiwanej statystyki pozycyjnej, to następuje przejście do prawego
// syna
18          if (el[po] < p) p -= el[po++];
19       }
// Zwracany jest numer znalezionego elementu
20       return po - (1 << s);
```



Rysunek 6.5: **(a)** Stan pustego drzewa pozycyjnego **(b)** Dodanie do drzewa elementu 0. **(c)** Dodanie do drzewa elementu 3.

Listing 6.10: (c.d. listingu z poprzedniej strony)

```
21    }
22 };
```

Listing 6.11: Wykorzystanie struktury `PosTree` na przykładzie elementów z dziedziny $\{0, 1, \ldots, 15\}$

```
Do drzewa dodano element 0
1-statystyka pozycyjna = 0
Do drzewa dodano element 7
2-statystyka pozycyjna = 7
Do drzewa dodano element 3
2-statystyka pozycyjna = 3
Z drzewa usunieto element 3
2-statystyka pozycyjna = 7
```

Listing 6.12: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.11. Pełny kod źródłowy programu znajduje się w pliku `postree.cpp`

```
01 int main() {
02    int w1, w2, w3;
// Skonstruuj drzewo pozycyjne dla elementów [0..15]
03    PosTree tree(4);
// Wykonaj listę operacji...
04    while (cin >> w1 >> w2) {
05       if (w1 == 0) {
// Operacje dodawania / usuwania elementów
06          cin >> w3;
07          tree.Add(w2, w3);
08          if (w3 == 1) cout << "Do drzewa dodano element " << w2 << endl;
09          else cout << "Z drzewa usunieto element " << w2 << endl;
10       } else
// Wyznaczanie statystyki pozycyjnej
11       cout << w2 << "-statystyka pozycyjna = " << tree.Find(w2) << endl;
12    }
13    return 0;
14 }
```

Zadanie: Kodowanie permutacji

Pochodzenie: II Olimpiada Informatyczna

Rozwiązanie: `perm.cpp`

Każdą permutację $A = (a_1, .., a_n)$ liczb $1, ..., n$ można zakodować za pomocą ciągu $B = (b_1, .., b_n)$, w którym b_i jest równe liczbie wszystkich a_j takich, że: ($j \leqslant i$ oraz $a_j \geqslant a_i$) dla każdego $i = 1, ..., n$.

Przykład:

Kodem permutacji $A = (1, 5, 2, 6, 4, 7, 3)$ jest ciąg: $B = (0, 0, 1, 0, 2, 0, 4)$.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia długość n i kolejne wyrazy ciągu liczb B,
- sprawdzi, czy ciąg B jest kodem jakiejś permutacji liczb $1, ..., n$,
- jeżeli tak, to znajdzie tę permutację i wypisze ją na standardowe wyjście,
- w przeciwnym przypadku wypisze jedno słowo NIE

Wejście

W pierwszym wierszu wejścia zapisana jest dodatnia liczba całkowita $n \leqslant 1\,000\,000$. Jest to liczba wyrazów ciągu B. W każdym z kolejnych n wierszy jest zapisana jedna liczba całkowita nieujemna nie większa niż 1 000 000. Jest to kolejny wyraz ciągu B.

Wyjście

Program powinien wypisać w każdym z kolejnych n wierszy jeden wyraz permutacji A, której kodem jest dany ciąg B, lub jedno słowo NIE, jeśli ciąg B nie jest kodem żadnej permutacji.

Przykład

Dla następującego wejścia:

```
7
0
0
1
0
2
0
4
```

Poprawnym rozwiązaniem jest:

```
1
5
2
6
4
7
3
```

Dla następującego wejścia:

```
4
0
2
0
0
```

Poprawnym rozwiązaniem jest:

```
NIE
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 440 acm.uva.es - zadanie 151	acm.uva.es - zadanie 10015	acm.sgu.ru - zadanie 254

6.2.4. Drzewa pokryciowe

Ostatnim przykładem drzew statycznych jakie zaprezentujemy są drzewa pokryciowe. Elementami wstawianymi do tych drzew, w odróżnieniu od dotychczasowych, są przedziały, których końce należą do zbioru $[0..2^n-1]$. Dodatkową wspomaganą operacją jest wyznaczanie dla danego spójnego podprzedziału dziedziny, wielkości obszaru pokrytego przez jakikolwiek z wstawionych odcinków. Wierzchołki drzewa pokryciowego reprezentują odcinki jednostkowe — $(0,1)$, $(1,2)$, ..., natomiast wierzchołki wewnętrzne drzewa przechowują informacje dotyczące pokrytej powierzchni przez odcinki z ich poddrzew.

Implementacja drzew pokryciowych zrealizowana jest jako struktura `CoverTree`, której kod źródłowy znajduje się na listingu 6.13. Listing 6.14 prezentuje natomiast sposób użycia struktury `CoverTree` w praktyce. Korzystanie ze struktury `CoverTree` sprowadza się do wykorzystania dwóch funkcji — **void** `Mark(int p, int k, int w)`, która dodaje w wystąpień odcinka (p,k) (jeśli w jest ujemne, to następuje usunięcie odpowiedniej liczby wystąpień odcinka (p,k)) oraz **int** `Find(int p,int k)`, która wyznacza pokrycie przedziału $[p..k]$. Podczas korzystania ze struktury `CoverTree` należy pamiętać, że nie wolno z niej usuwać odcinków, które nie zostały wcześniej wstawione.

Listing 6.13: Implementacja struktury `CoverTree`.

```
// Struktura umożliwia dodawanie i usuwanie odcinków oraz wyznaczanie
// wielkości obszaru przez nie pokrytego
01 struct CoverTree {
02 #define nr (wp + wk + 1) >> 1
03   int *el, *ma, s, p, k, il;
// Konstruktor przyjmuje jako parametr wysokość konstruowanego drzewa (końce
// odcinków, na których wykonywane są operacje należą do przedziału [0..2^size-1])
04   CoverTree(int size) {
// Tablica el przechowuje liczbę odcinków pokrywających w całości przedział
// reprezentowany przez odpowiedni węzeł
05     el = new int[s = 1 << (size + 1)];
// Tablica ma przechowuje sumaryczną pokrytą powierzchnię przedziału
// reprezentowanego przez odpowiedni węzeł
06     ma = new int[s];
07     REP(x, s) el[x] = ma[x] = 0;
08   }
// Destruktor zwalnia zaalokowaną pamięć
09   ~CoverTree() {
10     delete[] el;
11     delete[] ma;
12   }
// Funkcja pomocnicza wykorzystywana przez operację dodawania i usuwania odcinków.
// Wykonuje ona aktualizację wartości w węzłach
13   void Mark(int wp, int wk, int g) {
```

Listing 6.13: (c.d. listingu z poprzedniej strony)

```
// Jeśli dodawany/usuwany odcinek jest rozłączny z przedziałem
// reprezentowanym przez aktualny węzeł, to przerwij
14      if(k<=wp || p>=wk) return;
// Jeśli odcinek pokrywa w całości przedział aktualnego węzła, to zmień wartość el,
15      if(p<=wp && k>=wk) el[g] += il; else {
// a jeśli nie, to wykonaj aktualizację zmiennych synów aktualnego węzła
16         Mark(wp, nr, 2 * g);
17         Mark(nr, wk, 2 * g + 1);
18      }
// Dokonaj aktualizacji pokrytego obszaru przetwarzanego przedziału
19      ma[g] = el[g] > 0 ? wk - wp :
20         (wk - 1 == wp ? 0 : ma[2 * g] + ma[2 * g + 1]);
21    }
// Funkcja dodająca i1 wystąpień odcinka [p1..k1] do drzewa. W przypadku,
// gdy i1 jest wartością ujemną, następuje usunięcie odcinka
22    void Add(int p1, int k1, int i1) {
23      p = p1;
24      k = k1;
25      il = i1;
26      Mark(0, s / 2, 1);
27    }
// Funkcja pomocnicza służąca do wyznaczania wielkości obszaru
// [wp, wk] pokrytego przez odcinki
28 int F(int wp, int wk, int g) {
// Jeśli testowany odcinek jest rozłączny z aktualnym przedziałem,
// to pokrycie jest równe 0
29      if (wp >= k || wk <= p) return 0;
// Jeśli przedział jest w całości pokryty, to zwróć wielkość
// części wspólnej testowanego odcinka i przedziału
30      if (el[g] > 0) return min(k, wk) - max(p, wp);
// Jeśli odcinek pokrywa w całości przedział, to zwróć pokrycie przedziału
31      if (p <= wp && wk <= k) return ma[g];
// Zwróć jako wynik sumę pokryć swoich synów
32      return wp == wk - 1 ? 0 : F(wp, nr, 2 * g) + F(nr, wk, 2 * g + 1);
33 }
// Właściwa funkcja realizująca wyznaczanie wielkości obszaru
// [p1, k1] pokrytego przez odcinki
34 int Find(int p1, int k1) {
35      p = p1;
36      k = k1;
37      return F(0, s / 2, 1);
38    }
39 };
```

Listing 6.14: Wykorzystanie struktury `CoverTree` na przedziale $\{0, 1, \ldots, 15\}$

```
Pokrycie odcinka [0,15] = 0
Dodanie odcinka [1,10]
```

Listing 6.14: (c.d. listingu z poprzedniej strony)

```
Pokrycie odcinka [0,15] = 9
Pokrycie odcinka [5,15] = 5
Dodanie odcinka [5,15]
Pokrycie odcinka [0,15] = 14
Usuniecie odcinka [1,10]
Pokrycie odcinka [0,15] = 10
```

Listing 6.15: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.14. Pełny kod źródłowy programu znajduje się w pliku `covertree.cpp`

```
01 int main() {
02    int w1, w2, w3;
// Skonstruuj drzewo pokryciowe dla przedziału [0..15]
03    CoverTree tree(4);
// Wczytaj polecenia i wykonaj je...
04    while(cin >> w1 >> w2 >> w3) {
05       if (w1 == 0) {
// Operacja dodawania nowego odcinka
06          tree.Add(w2, w3, 1);
07          cout << "Dodanie odcinka [" << w2 << "," << w3 << "]" << endl;
08       } else if (w1 == 1) {
// Operacja usuwania odcinka
09          tree.Add(w2, w3, -1);
10          cout << "Usuniecie odcinka [" << w2 << "," << w3 << "]" << endl;
11       } else
// Wyznaczanie pokrycia na przedziale [w2..w3]
12          cout << "Pokrycie odcinka [" << w2 << "," << w3 <<
13             "] = " << tree.Find(w2, w3) << endl;
14    }
15    return 0;
16 }
```

Ćwiczenia

Proste	Średnie	Trudne
	spoj.sphere.pl - zadanie 61	

6.3. Binarne drzewa statyczne dynamicznie alokowane

W poprzednich kilku rozdziałach przedstawiliśmy proste w realizacji struktury danych, bazujące na statycznych drzewach binarnych poszukiwań. Struktura tych drzew jest łatwa w użyciu — bez większych problemów można zmieniać zestaw operacji przez nie realizowanych. Jednak wielką wadą tych drzew jest fakt, iż zużycie pamięci (oraz czas konstrukcji drzewa) jest liniowy ze względu na wielkość dziedziny, z której pochodzą elementy. W przypadku gęstego wykorzystania elementów z dziedziny nie stanowi to wielkiego problemu, jednak w

przypadku, gdy elementy, które są wstawiane do drzewa, umieszczone są stosunkowo rzadko, to może się okazać, że nie dysponujemy odpowiednią ilością pamięci, aby skonstruować odpowiednie drzewo. Przyglądając się strukturze drzewa statycznego z rysunku 6.2.b widać, że wiele wierzchołków takich drzew jest przez większość czasu działania programu zupełnie niepotrzebnych. Zmodyfikowaną strukturą danych, dobrze zachowującą się w takiej sytuacji jest przedstawione w niniejszym rozdziale drzewo statyczne alokowane dynamicznie. Drzewo takie (przykład przedstawiony jest na rysunku 6.2.c) nie tworzy podczas konstrukcji węzłów dla wszystkich elementów dziedziny. Węzły te są konstruowane dopiero w momencie, gdy są faktycznie potrzebne, co w wielu przypadkach oznacza, że znaczna ich część nigdy nie zostanie skonstruowana. Podejście takie jednak istotnie komplikuje implementację struktury — węzły wewnętrzne muszą pamiętać wskaźniki na swoich synów — w przypadku drzew statycznych „wskaźniki" te były wyliczane na podstawie numeru węzła — synowie węzła k mieli numery $2 * k$ oraz $2 * k + 1$.

Drzewa dynamicznie alokowane zaprezentujemy na przykładzie drzew pokryciowych, omówionych w poprzednim podrozdziale. Drzewa te umożliwiają nam realizowanie takich operacji, jak wstawianie oraz usuwanie odcinków oraz wyznaczanie obszaru przez nie pokrytego.

Implementacja tych drzew jest realizowana przez strukturę `CoverBTree`, której kod źródłowy znajduje się na listingu 6.16. Przykładowe użycie tej struktury danych zostało zaprezentowane na listingu 6.17.

Listing 6.16: Implementacja struktury `CoverBTree`.

```
// Struktura umożliwia dodawanie i usuwanie odcinków oraz wyznaczanie obszaru
// przez nie pokrytego
01 struct CoverBTree {
// Struktura wierzchołka drzewa
02   struct Vert {
// Wskaźniki na odpowiednio lewego i prawego syna
03     Vert *s[2];
// Zmienna v reprezentuje pokryty obszar aktualnego wierzchołka, natomiast
// c zlicza liczbę odcinków, które pokrywają w całości przedział
// reprezentowany przez wierzchołek
04     int v, c;
05     Vert() {
06       s[v = c = 0] = s[1] = 0;
07     }
08   };
// Korzeń drzewa
09   Vert *root;
// Wartość reprezentująca początek i koniec przedziału, dla którego zostało
// skonstruowane drzewo
10   int zp, zk;
// Funkcja usuwa pamięć zaalokowaną dla danego wierzchołka oraz jego poddrzewa
11   void Rem(Vert * p) {
12     if (p) {
13       Rem(p->s[0]);
14       Rem(p->s[1]);
15       delete p;
16     }
17   }
```

Listing 6.16: (c.d. listingu z poprzedniej strony)

```
// Destruktor zwalnia całą pamięć przydzieloną na drzewo
18    ~CoverBTree() {
19       Rem(root);
20    }
// Konstruktor tworzy nowe drzewo dla przedziału [p..k]
21    CoverBTree(int p, int k) : zp(p), zk(k) {
22       root = new Vert;
23    }
// Zmienne pomocnicze dla operatorów (przypisywane są im odpowiednio początek i
// koniec odcinka oraz licznik określający ile kopii odcinka jest dodawanych /
// usuwanych)
24    int pp, kk, cc;
// Funkcja pomocnicza dla Add, dodająca lub usuwająca odcinek [pp..kk].
// Parametry p i k oznaczają przedział, który reprezentuje wierzchołek
// v
25    void Ad(int p, int k, Vert * v) {
26       if (kk <= p || pp >= k) return;
// Jeśli odcinek w całości pokrywa aktualny przedział, to następuje
// modyfikacja zmiennej c aktualnego wierzchołka
27       if (p >= pp && k <= kk) v->c += cc;
28       else {
29          int c = (p + k) / 2;
// Jeśli odcinek zachodzi na przedział lewego syna, to aktualizuj go
30          if (pp <= c) {
31             if (!v->s[0]) v->s[0] = new Vert;
32             Ad(p, c, v->s[0]);
33          }
// Jeśli odcinek zachodzi na przedział prawego syna, to aktualizuj go
34          if (kk >= c) {
35             if (!v->s[1]) v->s[1] = new Vert;
36             Ad(c, k, v->s[1]);
37          }
38       }
// Aktualizacja pokrycia przedziału. Jeśli zmienna c jest dodatnia, to
// odcinek jest pokryty w całości, a jeśli nie, to wielkość jego pokrycia jest
// zależna od jego synów
39       v->v = v->c ? k - p :
40       (v->s[0] ? v->s[0]->v : 0) + (v->s[1] ? v->s[1]->v : 0);
41    }
// Funkcja dodaje lub usuwa z drzewa odcinek [p..k]. Parametr c określa,
// czy odcinek jest dodawany (1), czy usuwany (-1)
42    void Add(int p, int k, int c) {
43       pp = p;
44       kk = k;
45       cc = c;
46       Ad(zp, zk, root);
47    }
// Funkcja pomocnicza, wyznaczająca wielkość pokrycia przedziału [pp..kk]
```

Listing 6.16: (c.d. listingu z poprzedniej strony)

```
48    int Fi(int p, int k, Vert * v) {
// Jeśli wierzchołek nie istnieje lub jego przedział jest rozłączny z
// odcinkiem to wyjdź
49       if (!v || p >= kk || k <= pp) return 0;
// Jeśli przedział jest pokryty w całości, to zwróć wielkość przecięcia z
// odcinkiem
50       if (v->c) return min(k, kk) - max(p, pp);
// Jeśli odcinek zawiera cały przedział, to zwróć pokrycie przedziału
51       if (p >= pp && k <= kk) return v->v;
// Wyznacz pokrycie dla obu synów
52       int c = (p + k) / 2;
53       return Fi(p, c, v->s[0]) + Fi(c, k, v->s[1]);
54    }
// Funkcja wyznaczająca pokrycie przedziału [p..k]
55    int Find(int p, int k) {
56       pp = p;
57       kk = k;
58       return Fi(zp, zk, root);
59    }
60 };
```

Listing 6.17: Przykład wykorzystania struktury danych `CoverBTree`

```
Pokrycie odcinka [0,1000000000] = 0
Dodanie odcinka [1,10000]
Pokrycie odcinka [0,1000000000] = 9999
Pokrycie odcinka [5,100000] = 9995
Dodanie odcinka [5,50000]
Pokrycie odcinka [0,100000] = 49999
Usuniecie odcinka [1,10000]
Pokrycie odcinka [0,100000] = 49995
```

Listing 6.18: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.17. Pełny kod źródłowy programu znajduje się w pliku `coverbtree.cpp`

```
01 int main() {
02    int w1, w2, w3;
// Skonstruuj dynamiczne drzewo pokryciowe dla przedziału [0..10^9]
03    CoverBTree tree(0, 1000000000);
// Wczytaj polecenia i wykonaj je...
04    while(cin >> w1 >> w2 >> w3) {
05       if (w1 == 0) {
// Operacja dodawania nowego odcinka
06          tree.Add(w2, w3, 1);
07          cout << "Dodanie odcinka [" << w2 << "," << w3 << "]" << endl;
08       } else if (w1 == 1) {
// Operacja usuwania odcinka
09          tree.Add(w2, w3, -1);
```

Listing 6.18: (c.d. listingu z poprzedniej strony)

```
10          cout << "Usuniecie odcinka [" << w2 << "," << w3 << "]" << endl;
11        } else
// Wyznaczanie pokrycia na przedziale [w2..w3]
12          cout << "Pokrycie odcinka [" << w2 << "," << w3 <<
13             "] = " << tree.Find(w2, w3) << endl;
14    }
15    return 0;
16 }
```

Zadanie: Marsjańskie mapy

Pochodzenie: Olimpiada Informatyczna Krajów Bałtyckich 2001

Rozwiązanie: `mars.cpp`

W roku 2051 przeprowadzonych zostało kilka ekspedycji na Marsie, w wyniku czego stworzone zostały mapy zbadanych obszarów. BaSA (Baltic Space Agency) chce stworzyć mapę całej planety — w celu określenia kosztów operacji, należy policzyć obszar planety, dla której mapy zostały już skonstruowane. Twoim zadaniem jest napisać taki program.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis map,
- wyznaczy sumaryczny obszar pokryty przez mapy,
- wypisze wynik na standardowe wyjście.

Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 10\,000$), oznaczającą liczbę dostępnych map. Każdy z kolejnych n wierszy zawiera cztery liczby całkowite — x_1, y_1, x_2, y_2 ($0 \leqslant x_1 \leqslant x_2 \leqslant 1\,000\,000$, $0 \leqslant y_1 \leqslant y_2 \leqslant 1\,000\,000$). (x_1, y_1) oraz (x_2, y_2) są współrzędnymi odpowiednio dolnego-lewego oraz górnego-prawego rogu obszaru znajdującego się na mapie. Wszystkie mapy są prostokątne, a ich boki są równoległe do osi x i y.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie jedną liczbę całkowitą — zbadany obszar planety.

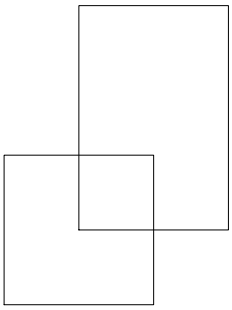
Przykład

Dla następującego wejścia:

```
2
10  10  20  20
15  15  25  30
```

Poprawnym rozwiązaniem jest:

```
225
```



Ćwiczenia

Proste	Średnie	Trudne
	acm.uva.es - zadanie 688	

6.4. Wzbogacane drzewa binarne

Literatura
[WDA] - 15
[SZP] - 6.2.2, 6.2.3

Po wnikliwej analizie poprzednich rozdziałów uważny czytelnik zaobserwuje, że istnieje duże podobieństwo pomiędzy poszczególnymi rodzajami prezentowanych drzew. Pomimo różnych operacji przez nie realizowanych, w większości przypadków sposób wyznaczania dodatkowych informacji, które umożliwiają efektywną realizację tych operacji, jest taki sam — wartość informacji w węźle wewnętrznym drzewa jest zależna wyłącznie od tego węzła oraz od jego synów. Można w takim wypadku stworzyć szablon, pozwalający na konstruowanie różnego rozdziału drzew, których funkcje wstawiania oraz usuwania węzłów będą identyczne, a jedyne różnice polegać będą na zasobie informacji przechowywanych w węzłach oraz na sposobie wyliczania ich wartości. Różnice pojawiać się również będą w zakresie zbioru funkcji, pozwalających na realizację dodatkowych operacji. Implementacja tych funkcji jest jednak prosta, gdyż nie są w nich dokonywane modyfikacje struktury drzewa (dodawania i usuwania wierzchołków). W niniejszym rozdziale przedstawimy dwie takie struktury danych. Pierwszą z nich będą wzbogacane drzewa binarnych poszukiwań, o których wiadomo, że w pesymistycznych przypadkach czas wykonywania operacji może być liniowy. Następnie przedstawimy modyfikację tych drzew, które będą różniły się realizacją funkcji wstawiania nowych węzłów — modyfikacja polega na wykonywaniu losowych zmian struktury drzewa, mających na celu zapobieganie zbytniemu rozrastaniu się drzewa w głąb.

Przedstawiona na listingu 6.19 implementacja struktury `BST` stanowi klasyczną realizację binarnego drzewa wyszukiwań. Główną zaletą tej implementacji jest możliwość określenia dodatkowych informacji, które należy przechowywać w wierzchołkach drzewa. Podczas konstrukcji drzewa `BST` podaje się strukturę `element`, która ma być częścią tworzonych wierzchołków oraz określa się dwie funkcje f i g. Zadaniem funkcji f o sygnaturze **int** `f(`**const** `element&,` **const** `element&)` jest określanie liniowego porządku na zbiorze elementów przechowywanych w drzewie. Jeśli element określony przez pierwszy parametr jest większy od elementu drugiego, to funkcja ta powinna zwracać wartość 1. Jeśli element drugi jest mniejszy od pierwszego, to wartością zwracaną jest 0, natomiast jeśli elementy są równe, to funkcja powinna zwrócić -1. Druga funkcja — `g`, o sygnaturze **void** `g(element*, element*, element*)` przyjmuje jako parametry trzy wskaźniki na wierzchołki drzewa — odpowiednio ojca oraz jego lewego i prawego syna (w przypadku, gdy któryś z synów nie istnieje, to odpowiedni wskaźnik ma wartość `0`). Zadaniem funkcji jest zaktualizowanie informacji wierzchołka-ojca na podstawie informacji z jego synów. Przykładowa implementacja drzew pozycyjnych, pozwalająca na wyszukiwanie k-tego najmniejszego elementu w drzewie, która bazuje na strukturze `BST` została przedstawiona na listingu 6.20.

Listing 6.19: Implementacja struktury `BST`

```
// Implementacja struktury BST
01 template <typename _T> struct BST {
// Struktura węzła drzewa
02   struct Ve {
// Wskaźniki na lewego i prawego syna
03     Ve *s[2];
// Pole e zawiera element przechowywany w węźle
04     _T e;
05     Ve() {
06       s[0] = s[1] = 0;
```

Listing 6.19: (c.d. listingu z poprzedniej strony)

```
07    }};
// sygnatury funkcji do porównywania elementów oraz do aktualizacji ich
// zawartości
08 #define _Less int (*Less)(const _T&, const _T&)
09 #define _Upd void (*Upd)(_T*, _T*, _T*)
10    _Less;
11    _Upd;
// Korzeń drzewa oraz dodatkowa zmienna pomocnicza
12    Ve *root, *v;
// Wektor wskaźników do węzłów drzewa, wykorzystywany do wyznaczania ścieżki od
// przetwarzanego węzła do korzenia drzewa
13    vector<Ve *> uV;
// Funkcja usuwa dany węzeł wraz z jego całym poddrzewem
14    void Rem(Ve * p) {
15       if (p) {
16          REP(x, 2) Rem(p->s[x]);
17          delete p;
18       }
19    }
// Destruktor drzewa zwalnia zaalokowaną pamięć
20    ~BST() {
21       Rem(root);
22    }
// Konstruktor drzewa przyjmuje jako parametry dwie funkcje - funkcję
// określającą porządek na elementach oraz funkcję służącą do aktualizowania
// informacji elementów
23    BST(_Less, _Upd) : Less(Less), Upd(Upd) {
24       root = 0;
25    }
// Funkcja aktualizuje wartości elementów, znajdujących się na ścieżce od
// ostatnio przetwarzanego węzła do korzenia drzewa
26    void Update() {
27       for (int x = SIZE(uV) - 1; x >= 0 && (v = uV[x]); x--)
28       Upd(&v->e, v->s[0] ? &(v->s[0]->e) : 0, v->s[1] ? &(v->s[1]->e) : 0);
29       uV.clear();
30    }
// Funkcja pomocnicza wyszukująca węzeł w drzewie zawierający zadany element
31    Ve *Find(const _T & e, bool s) {
32       v = root;
33       if (s) uV.PB(v);
34       for (int c; v && (-1 != (c = Less(v->e, e)));) {
35          if (s) uV.PB(v);
36          v = v->s[c];
37       }
38       return v;
39    }
// Funkcja wyszukuje zadany element w drzewie
40    _T *Find(const _T & e) {
```

Listing 6.19: (c.d. listingu z poprzedniej strony)

```
41      return (v = Find(e, 0)) ? &(v->e) : 0;
42    }
// Funkcja wstawia zadany element do drzewa
43    void Insert(const _T & e) {
// Jeśli drzewo nie jest puste...
44      if (root) {
// Znajdź odpowiednie miejsce w drzewie dla dodawanego elementu i go dodaj
45        Find(e, 1);
46        v = uV.back();
47        int cmp = Less(v->e, e);
48        if (cmp != -1) uV.PB(v = v->s[cmp] = new Ve);
49      }
// Stwórz nowy korzeń drzewa z zadanym elementem
50      else uV.PB(v = root = new Ve);
51      v->e = e;
// Zaktualizuj elementy na ścieżce od dodanego elementu do korzenia
52      Update();
53    }
// Funkcja usuwa z drzewa węzeł zawierający zadany element
54    bool Delete(const _T & e) {
55      Ve *c = Find(e, 1), *k;
// Jeśli elementu nie ma w drzewie, to przerwij
56      if (!c) {
57        uV.resize(0);
58        return 0;
59      }
60      if (c != root) uV.PB(c);
// Jeśli węzeł ma obu synów, to musi nastąpić zamiana elementów w węźle
// aktualnym z kolejnym elementem...
61      if (c->s[0] && c->s[1]) {
62        for (k = c->s[1]; k->s[0]; k = k->s[0])
63        uV.PB(k);
64        uV.PB(k);
65        c->e = k->e;
66        c = k;
67      }
// Usunięcie odpowiednich węzłów z drzewa
68      if (SIZE(uV) > 1)
69        (k = uV[SIZE(uV) - 2])->s[k->s[0] != c] = c->s[0] ? c->s[0] : c->s[1];
70      else root = c->s[0] ? c->s[0] : c->s[1];
// Aktualizacja elementów na ścieżce
71      Update();
72      delete c;
73      return 1;
74    }
75 };
```

Listing 6.20: Przykład wykorzystania struktury `BST`.

```
Dodawanie do drzewa elementu 10
Dodawanie do drzewa elementu 15
1 statystyka pozycyjna to: 10
2 statystyka pozycyjna to: 15
Dodawanie do drzewa elementu 1
1 statystyka pozycyjna to: 1
2 statystyka pozycyjna to: 10
3 statystyka pozycyjna to: 15
W drzewie nie ma wezla bedacego 4 statystyka pozycyjna
Usuwanie z drzewa elementu 10
2 statystyka pozycyjna to: 15
```

Listing 6.21: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.20. Pełny kod źródłowy programu znajduje się w pliku `bst.cpp`

```
// Element stanowiący zawartość wierzchołków drzewa
01 struct Element {
02    int val, stat;
03 };
// Funkcja określająca porządek na wierzchołkach drzewa
// w kolejności rosnących wartości val
04 int Cmp(const Element &a, const Element &b) {
05    return (a.val == b.val) ? -1 : a.val < b.val;
06 }
// Funkcja aktualizująca wartość stat wierzchołka
07 void Update(Element *p, Element *ls, Element *rs){
08    p->stat = 1 + (ls ? ls->stat : 0);
09 }
// Funkcja zwraca wskaźnik do elementu drzewa,
// stanowiącego k-tą statystykę pozycyjną
10 Element* Stat(BST<Element> &t, int k) {
11    VAR(v, t.root);
12    while(v && k != v->e.stat) {
13       if (v->s[0] && v->s[0]->e.stat >= k) v = v->s[0];
14       else {
15          if (v->s[0]) k -= v->s[0]->e.stat;
16          k--;
17          v = v->s[1];
18       }
19    }
20    return v ? &(v->e) : 0;
21 }
22
23 int main() {
// Skonstruuj drzewo binarne wzbogacając je obiektami typu Element
// oraz wykorzystując funkcje Cmp i Update
24    BST<Element> Tree(Cmp,Update);
```

Listing 6.21: (c.d. listingu z poprzedniej strony)

```
25    Element e, *p;
26    int cmd;
// Wczytaj operacje do wykonania i je wykonaj
27    while(cin >> cmd >> e.val) {
28       if (cmd == 0) {
29          Tree.Insert(e);
30          cout << "Dodawanie do drzewa elementu " << e.val << endl;
31       } else if(cmd == 1) {
32          cout << "Usuwanie z drzewa elementu " << e.val << endl;
33          Tree.Delete(e);
34       } else {
35          p = Stat(Tree, e.val);
36          if (!p)
37             cout << "W drzewie nie ma wezla bedacego " << e.val <<
38                " statystyka pozycyjna" << endl;
39          else cout << e.val << " statystyka pozycyjna to: " << p->val << endl;
40       }
41    }
42    return 0;
43 }
```

Implementacja randomizowanych drzew binarnych, których kod źródłowy przedstawiony jest na listingu 6.22, różni się od BST tym, że podczas wykonywania operacji wstawiania do drzewa nowych elementów, wykonywane są losowe rotacje. W pozostałych operacjach, wykorzystanie drzew RBST jest dokładnie takie samo jak BST — przykład przedstawiony został na listingu 6.23.

Listing 6.22: Implementacja struktury RBST

```
// Implementacja struktury RBST
001 template <typename _T> struct RBST {
// Struktura węzła drzewa
002    struct Ve {
// Wskaźniki na lewego i prawego syna
003       Ve *s[2];
// Pole e zawiera element przechowywany w węźle
004       _T e;
// Zmienna wykorzystywana do realizacji rotacji na drzewie
005       int w;
006       Ve(){s[0] = s[1] = 0;}
007    };
// Sygnatury funkcji do porównywania elementów oraz do aktualizacji ich zawartości
008 #define _Less int (*Less)(const _T&, const _T&)
009 #define _Upd void (*Upd)(_T*, _T*, _T*)
010    _Less;
011    _Upd;
// Korzeń drzewa oraz dodatkowa zmienna pomocnicza
012    Ve *root, *v;
```

Listing 6.22: (c.d. listingu z poprzedniej strony)

```
// Wektor wskaźników do węzłów drzewa wykorzystywany do wyznaczania ścieżki
// od przetwarzanego węzła do korzenia drzewa
013   vector<Ve*> uV;
// Funkcja usuwa dany węzeł wraz z jego całym poddrzewem
014   void Rem(Ve* p) {
015     if (p) {
016       REP(x,2) Rem(p->s[x]);
017       delete p;
018     }
019   }
// Destruktor drzewa zwalnia zaalokowaną pamięć
020   ~RBST() {
021     Rem(root);
022   }
// Konstruktor drzewa przyjmuje jako parametry dwie funkcje - określającą porządek
// na elementach oraz służącą do aktualizowania informacji elementów
023   RBST(_Less, _Upd) : Less(Less), Upd(Upd) {
024     root = 0;
025   }
// Funkcja aktualizuje wartości elementów znajdujących się na ścieżce od ostatnio
// przetwarzanego węzła do korzenia drzewa
026   void Update(vector<Ve*> &uV) {
027     for(int x = SIZE(uV) - 1; x >= 0 && (v = uV[x]); x--) {
028       v->w = 1 + (v->s[0] ? v->s[0]->w : 0) + (v->s[1] ? v->s[1]->w : 0);
029       Upd(&v->e, v->s[0] ? &(v->s[0]->e) : 0,
030         v->s[1] ? &(v->s[1]->e) : 0);
031     }
032     uV.clear();
033   }
// Funkcja pomocnicza wyszukująca węzeł w drzewie zawierający zadany element
034   Ve* Find(const _T& e, bool s) {
035     v = root;
036     if (s) uV.PB(v);
037     for(int c; v && (-1 != (c = Less(v->e, e)));) {
038       v = v->s[c];
039       if (s) uV.PB(v);
040     }
041     return v;
042   }
// Funkcja wyszukuje zadany element w drzewie
043   _T* Find(const _T& e) {
044     return (v = Find(e,0)) ? &(v->e) : 0;
045   }
// Ponieważ operacja wykonywania rotacji na drzewie wymaga przechowywania
// w węzłach informacji na temat wielkości poddrzew, zatem informację tą można
// wykorzystać do wyznaczania statystyk pozycyjnych
046   _T* StatPos(int nr) {
047     Ve* v = root;
```

Listing 6.22: (c.d. listingu z poprzedniej strony)

```
048       if (!v || v->w < nr) return 0;
049       while (v && nr>0) {
050          if (v->s[0] && v->s[0]->w >= nr)
051             v = v->s[0];
052          else {
053             if (v->s[0]) nr -= v->s[0]->w;
054             if (--nr) v = v->s[1];
055          }
056       }
057       if (!v) return 0;
058       return &(v->e);
059    }
// Funkcja pomocnicza wykonująca rotacje na drzewie w celu jego równoważenia
060    Ve* AtRoot(Ve* p, const _T& e) {
061       Ve* d = new Ve, *a, *t, *r=0;
062       vector<Ve*> Up[2];
063       d->e = e;
064       int z = Less(p->e, e);
065       a = (t = p)->s[z];
066       p->s[z] = 0;
067       d->s[1 - z] = t;
068       d->s[z] = a;
069       Up[1 - z].PB((r = d)->s[1 - z]);
070       while(a) {
071          if (Less(e, a->e) == z) {
072             Up[z].PB(r=a);
073             a = a->s[1-z];
074          } else {
075             r->s[r->s[0] != a] = 0;
076             t->s[(t == d) ∧ z] = a;
077             t = r;
078             z = 1 - z;
079          }
080       }
081       REP(x, 2) Update(Up[x]);
082       uV.PB(d);
083       return d;
084    }
// Funkcja wstawia zadany element do drzewa
085    void Insert(const _T& e) {
086       if (!root) {
087          uV.PB(root = new Ve);
088          root->e = e;
089       } else if (v = Find(e, 1))
090          v->e = e;
091       else {
092          uV.clear();
093          v = root;
```

Listing 6.22: (c.d. listingu z poprzedniej strony)

```
094         int cmp = 0;
095         while(v && rand() % (v->w + 1)) {
096            uV.PB(v);
097            v = v->s[cmp = Less(v->e, e)];
098         }
099         if (!v) {
100            uV.PB(v = uV.back()->s[cmp] = new Ve);
101            v->e = e;
102         } else
103            SIZE(uV) ? uV.back()->s[cmp] = AtRoot(v, e) : root = AtRoot(v, e);
104      }
105      Update(uV);
106   }
// Funkcja usuwa określony element z drzewa
107   bool Delete(const _T& e) {
108      Ve* c = Find(e, 1), *k;
109      if (!c) {
110         uV.clear();
111         return 0;
112      }
113      if (c->s[0] && c->s[1]) {
114         for(k = c->s[1]; k->s[0]; k = k->s[0]) uV.PB(k);
115         uV.PB(k);
116         c->e = k->e;
117         c = k;
118      }
119      if (SIZE(uV)>=2)
120         (k = uV[SIZE(uV) - 2])->s[k->s[0] != c] = c->s[0] ? c->s[0] : c->s[1];
121      else
122         root = c->s[0] ? c->s[0] : c->s[1];
123      Update(uV);
124      delete c;
125      return 1;
126   }
127 };
```

Listing 6.23: Przykład wykorzystania struktury RBST.

```
W drzewie nie ma odcinka zawierajacego punktu 7
Dodawanie do drzewa odcinka (5,10)
Punkt 7 zawiera odcinek (5,10)
W drzewie nie ma odcinka zawierajacego punktu 2
Dodawanie do drzewa odcinka (1,20)
Punkt 2 zawiera odcinek (1,20)
Usuniecie z drzewa odcinka (1,20)
W drzewie nie ma odcinka zawierajacego punktu 2
```

Listing 6.24: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 6.23. Pełny kod źródłowy programu znajduje się w pliku `rbst.cpp`

```
// Element stanowiący zawartość wierzchołków drzewa
01 struct Element {
// Lewy i prawy koniec odcinka oraz maksimum z końców odcinków z poddrzewa
02    int l, r, mr;
03 };
// Funkcja określająca porządek liniowy na elementach
04 int Cmp(const Element &a, const Element &b) {
05    if (a.l == b.l && a.r == b.r) return -1;
06    return (a.l == b.l) ? a.r < b.r : a.l < b.l;
07 }
// Funkcja aktualizująca wartość mr elementów
08 void Update(Element *p, Element *ls, Element *rs){
09    p->mr = p->r;
10    if (ls) p->mr = max(p->mr, ls->mr);
11    if (rs) p->mr = max(p->mr, rs->mr);
12 }
13
// Funkcja znajduje odcinek zawierający punkt o współrzędnej k
14 Element* Find(RBST<Element> &t, int k) {
15    VAR(v, t.root);
16    while(v) {
17       if (v->e.l <= k && v->e.r >= k) return &(v->e);
18       v = (v->s[0] && v->s[0]->e.mr >= k) ? v->s[0] : v->s[1];
19    }
20    return 0;
21 }
22
23 int main() {
// Skonstruuj randomizowane drzewo binarne wzbogacone obiektami typu Element
// oraz wykorzystując funkcje Cmp i Update
24    RBST<Element> Tree(Cmp, Update);
25    Element e, *p;
26    int cmd;
// Wczytaj operacje do wykonania i je wykonaj
27    while(cin >> cmd >> e.l) {
28       if (cmd == 0) {
29          cin >> e.r;
30          Tree.Insert(e);
31          cout << "Dodawanie do drzewa odcinka ("
32             << e.l << "," << e.r << ")" << endl;
33       } else if(cmd == 1) {
34          cin >> e.r;
35          cout << "Usuniecie z drzewa odcinka ("
36             << e.l << "," << e.r << ")" << endl;
37          Tree.Delete(e);
38       } else {
```

Listing 6.24: (c.d. listingu z poprzedniej strony)

```
39          p = Find(Tree, e.l);
40          if (!p) cout << "W drzewie nie ma odcinka zawierajacego punktu "
41             << e.l << endl;
42          else cout << "Punkt " << e.l << " zawiera odcinek ("
43             << p->l << "," << p->r << ")" << endl;
44       }
45    }
46    return 0;
47 }
```

Zadanie: Puste prostopadłościany

Pochodzenie: VI Olimpiada Informatyczna

Rozwiązanie: `cuboids.cpp`

Prostopadłościan nazwiemy regularnym, gdy:

- jednym z jego wierzchołków jest punkt o współrzędnych $(0, 0, 0)$,
- krawędzie wychodzące z tego wierzchołka leżą na dodatnich półosiach układu współrzędnych,
- krawędzie te mają długości nie większe niż 10^6

Dany jest zbiór A punktów przestrzeni, których wszystkie współrzędne są całkowite i należą do przedziału $[1..10^6]$. Szukamy prostopadłościanu regularnego o maksymalnej objętości, który w swoim wnętrzu nie zawiera żadnego punktu ze zbioru A. Punkt należy do wnętrza prostopadłościanu, jeżeli jest punktem prostopadłościanu, ale nie jego ściany.

Zadanie

Napisz program, który:

- wczyta współrzędne punktów ze zbioru A,
- wyznaczy jeden z prostopadłościanów regularnych o maksymalnej objętości, nie zawierający w swoim wnętrzu punktów ze zbioru A,
- wypisze wynik

Wejście

W pierwszym wierszu znajduje się jedna całkowita nieujemna liczba n, $n \leqslant 5\,000$, będąca liczbą elementów zbioru A. W kolejnych n wierszach znajdują się trójki liczb całkowitych z przedziału $[1..10^6]$ będące współrzędnymi (odpowiednio x, y i z) punktów ze zbioru A. Liczby w wierszu pooddzielane są pojedynczymi odstępami.

Wyjście

W pierwszym i jedynym wierszu powinny znaleźć się trzy liczby całkowite oddzielone pojedynczymi odstępami, będące współrzędnymi (odpowiednio x, y i z) wierzchołka znalezionego prostopadłościanu regularnego, który ma wszystkie współrzędne dodatnie.

Przykład

Dla następującego wejścia:

```
4
3  3  300000
2  200000  5
90000  3  2000
2  2  1000
```

Poprawnym rozwiązaniem jest:

```
1000000  200000  1000
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10869	spoj.sphere.pl - zadanie 382	acm.sgu.ru - zadanie 263
acm.sgu.ru - zadanie 193	acm.sgu.ru - zadanie 199	acm.uva.es - zadanie 501
	acm.uva.es - zadanie 221	spoj.sphere.pl - zadanie 205

Rozdział 7

Algorytmy tekstowe

Literatura
[EAT] - 5
[WDA] - 34
[ASD] - 5

W tym rozdziale omówimy różne algorytmy tekstowe — większość z nich jest związana z zagadnieniem wyszukiwania wzorca. Zaprezentujemy między innymi algorytm KMP, pozwalający na znajdowanie wszystkich wystąpień wzorca w tekście oraz jego modyfikację, umożliwiającą wyszukiwanie wielu wzorców jednocześnie.

Przedstawimy również problemy wyznaczania maksymalnego leksykograficznie sufiksu słowa, wyliczania promieni palindromów będących podsłowami danego słowa, wyszukiwania minimalnej leksykograficznej cykliczności słowa oraz sprawdzania, czy dwa dane słowa są równoważne cyklicznie.

Bardzo ważną częścią rozdziału jest prezentacja implementacji drzew sufiksowych — struktury danych o liniowej wielkości, która umożliwia reprezentację wszystkich sufiksów danego słowa. Struktura ta pozwala na efektywne rozwiązywanie wielu problemów tekstowych, takich jak zliczanie liczby wystąpień wzorca w tekście, wyznaczanie najdłuższego wspólnego podsłowa wielu słów, czy znajdowanie najdłuższych podsłów występujących określoną liczbę razy.

Dla danego słowa s o długości n kolejne litery tego słowa będziemy numerowali od 1 do n i oznaczać je będziemy poprzez $s_1, s_2, \ldots, s_n$. W niektórych rozdziałach będą przedstawiane algorytmy, których czas działania zależeć będzie od wielkości alfabetu (wyznaczanie krawędzi wychodzących z wierzchołków etykietowanych literami wymaga wyszukania ich w słowniku, co zajmuje czas logarytmiczny ze względu na liczbę liter). W przypadku tego typu algorytmów będziemy pomijali logarytmiczny czynnik podczas analizy złożoności realizowanych operacji.

7.1. Algorytm KMP

Literatura
[WDA] - 34.4
[EAT] - 3.1
[ASD] - 5.1.2

Załóżmy, że mamy dane dwa teksty s oraz t i chcemy znaleźć wystąpienie (ewentualnie wiele wystąpień) t w s. Problem tego typu nazywamy wyszukiwaniem wzorca t w tekście s.

Wyszukiwanie wzorca w tekście można zrealizować w bardzo prosty sposób. Dla każdej pozycji przeszukiwanego tekstu można sprawdzić, czy kolejne jego litery, poczynając od tej pozycji, zgadzają się z literami wyszukiwanego wzorca. Implementacja takiego podejścia zajmuje dosłownie dwie linijki, ale niestety złożoność algorytmu jest w pesymistycznym przypadku kwadratowa, choć w średnim przypadku działa bardzo szybko. Wyszukując wzorca postaci $a^n b$ w tekście a^{2*n} (zapis a^n oznacza n-krotną konkatenację a), dla pierwszych n pozycji

będzie wykonywanych po n porównań ze wzorcem, zanim nastąpi znalezienie niezgodności pomiędzy literami a i b. Obserwując działanie takiego algorytmu widać, że wiele porównań mogło by zostać pominiętych na podstawie wiedzy uzyskanej o tekstach podczas wykonywania poprzednich kroków algorytmu. Istnieje możliwość skonstruowania algorytmu działającego w czasie liniowym, wykorzystującego tę wiedzę.

Algorytm KMP (którego nazwa pochodzi od nazwisk jego autorów — Knutha, Morrisa i Pratta) służy do wyszukiwania wzorca w tekście w czasie liniowym. Jest on stosunkowo prosty w implementacji — do swojego działania wykorzystuje tzw. tablicę prefiksową, która jest wyznaczana dla wyszukiwanego wzorca. Dla k-tej litery wzorca t, odpowiednia wartość tablicy jest równa długości najdłuższego właściwego prefiksu słowa $t_1 t_2 \dots t_k$, będącego równocześnie jego sufiksem (tzw. prefikso-sufiks). Dla słowa *abca*, najdłuższym właściwym prefikso-sufiksem jest a, natomiast w przypadku słowa *abcab* jest to *ab*. W przypadku wzorca *abcaba*, tablica prefiksowa wygląda następująco: $[0, 0, 0, 1, 2, 1]$.

Wyznaczanie wartości tablicy prefiksowej p dla tekstu t można wykonywać przyrostowo — na podstawie wartości $p[1], p[2], \dots, p[k-1]$ w prosty sposób wyznaczyć można wartość $p[k]$. Zauważmy, że jeżeli $t_{p[k-1]+1} = t_k$, to wtedy $p[k] = p[k-1] + 1$, czyli kolejny prefikso-sufiks stanowi wydłużenie prefikso-sufiksu dla słowa o jedną literę krótszego. Jeśli natomiast $t_{p[k-1]+1} \neq t_k$, to wówczas należy wyznaczyć najdłuższy prefikso-sufiks słowa $t_1 t_2 \dots t_{k-1}$ taki, aby litera występująca po nim w słowie t była równa t_k. Wyznaczenie takiego słowa można wykonać, sprawdzając kolejne prefiksy słowa t długości: $p[k-1], p[p[k-1]], \dots$ Jeśli w ten sposób nie zostanie znaleziony odpowiedni prefikso-sufiks, to $p[k] = 0$. Ponieważ w każdym kroku długość ostatnio znalezionego prefikso-sufiksu może wzrastać co najwyżej o 1, zatem wyznaczanie tablicy prefiksowej można zrealizować w czasie liniowym ze względu na długość wzorca.

Po wyznaczeniu tablicy prefiksowej, wyszukiwanie wzorca w tekście jest proste — wystarczy porównywać kolejne litery wzorca i tekstu, pamiętając długość zgadzającego się fragmentu tekstu. Jeśli porównywane litery wzorca i tekstu są różne, to należy wykorzystać tablicę prefiksową w celu cofnięcia się we wzorcu aż do momentu znalezienia takich samych liter (krok ten wygląda podobnie do konstrukcji tablicy prefiksowej).

Prezentowana na listingu 7.1 funkcja **void** `KMP(`**const char***, **const char***, **void** `(*)(`**int**`))` przyjmuje jako parametry: tekst, wyszukiwany wzorzec oraz wskaźnik do funkcji wywoływanej dla każdego wyznaczonego wystąpienia wzorca w tekście. Przykład wykorzystania tej funkcji został przedstawiony na listingu 7.2.

Listing 7.1: Implementacja funkcji **void** `KMP(`**const char***, **const char***, **void** `(*)(`**int**`))`

```
// Funkcja wyszukuje wystąpienia wzorca w tekście i dla każdego znalezionego
wystąpienia wywołuje funkcję fun
01 void KMP(const char* str,const char* wzo, void (*fun)(int)) {
02 #define KMPH(z) while(k > 0 && wzo[k] != z[q]) k = p[k]; if(wzo[k] == z[q]) k++;
03    int p[strlen(wzo) + 1], k = 0, q, m;
04    p[1] = 0;
// Wyznacz tablicę prefiksową dla wyszukiwanego wzorca
05    for (q = 1; wzo[q]; q++) {
06       KMPH(wzo);
07       p[q + 1] = k;
08    }
09    m = q;
10    k = 0;
// Dla kolejnych liter przeszukiwanego tekstu...
```

Listing 7.1: (c.d. listingu z poprzedniej strony)

```
11    for (q = 0; str[q]; q++) {
// Używając tablicy prefiksowej, wyznacz długość sufiksu tekstu
// str[0..q], będącego jednocześnie prefiksem wzorca
12      KMPH(str);
// Jeśli wyznaczony prefiks jest równy długości wzorca, to wywołaj
// funkcję fun dla znalezionego wystąpienia wzorca
13      if(m == k) {
14        fun(q - m + 1);
15        k = p[k];
16      }
17    }
18 }
```

Listing 7.2: Przykład wykorzystania funkcji **void** KMP(**const char***, **const char***, **void**(*)(**int**))

```
Wyszukiwanie abab w ababcabdababab
Znaleziono wystapienie wzorca na pozycji 0
Znaleziono wystapienie wzorca na pozycji 8
Znaleziono wystapienie wzorca na pozycji 10
```

Listing 7.3: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.2. Pełny kod źródłowy programu znajduje się w pliku `kmp.cpp`

```
// Funkcja wywoływana dla każdego znalezionego wystąpienia wzorca
01 void Disp(int x) {
02    cout << "Znaleziono wystapienie wzorca na pozycji " << x << endl;
03 }
04 int main() {
05    string pattern, text;
// Wczytaj tekst oraz wyszukiwany wzorzec
06    cin >> text >> pattern;
07    cout << "Wyszukiwanie " << pattern << " w " << text << endl;
// Wyznacz wszystkie wystąpienia wzorca w tekście
08    KMP(text.c_str(), pattern.c_str(), Disp);
09    return 0;
10 }
```

Zadanie: Szablon

Pochodzenie: XII Olimpiada Informatyczna

Rozwiązanie: `template.cpp`

Bajtazar chce umieścić na swoim domu długi napis. W tym celu najpierw musi wykonać odpowiedni szablon z wyciętymi literkami. Następnie przykłada taki szablon we właściwe miejsce do ściany i maluje po nim farbą, w wyniku czego na ścianie pojawiają się literki znajdujące się na szablonie. Gdy szablon jest przyłożony do ściany, to malujemy od razu wszystkie znajdujące się na nim literki (nie można tylko części). Dopuszczamy natomiast

możliwość, że któraś litera na ścianie zostanie narysowana wielokrotnie, w wyniku różnych przyłożeń szablonu. Literki na szablonie znajdują się koło siebie (nie ma tam przerw). Oczywiście można wykonać szablon zawierający cały napis. Bajtazar chce jednak zminimalizować koszty i w związku z tym wykonać szablon tak krótki, jak to tylko możliwe.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia napis, który Bajtazar chce umieścić na domu,
- obliczy minimalną długość potrzebnego do tego szablonu,
- wypisze wynik na standardowe wyjście.

Wejście

W pierwszym i jedynym wierszu standardowego wejścia znajduje się jedno słowo. Jest to napis, który Bajtazar chce umieścić na domu. Napis składa się z nie więcej niż 500 000 oraz nie mniej niż 1 małej litery alfabetu angielskiego.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia należy zapisać jedną liczbę całkowitą — minimalną liczbę liter na szablonie.

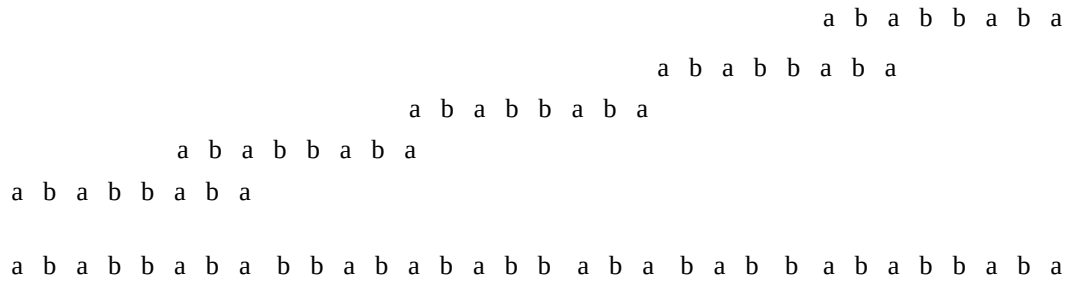
Przykład

Dla następującego wejścia:

```
abababbababababbababababbababbaba
```

Poprawnym rozwiązaniem jest:

```
8
```



7.2. Minimalny okres słowa

Tablica prefiksowa opisana w ramach algorytmu KMP ma wiele ciekawych własności. Przy jej użyciu można przykładowo wyznaczać minimalny okres danego słowa.

Definicja 7.2.1 Okresem słowa *s nazywamy takie słowo t, $|s| \geqslant |t|$, dla którego istnieją słowo l oraz liczba naturalna k, takie że $t^k = sl$. Innymi słowy, s jest prefiksem słowa t^k.*

Przykładowo, dla słowa *abcabcabc*, okresem jest zarówno słowo *abcabc*, jak i *abc*. Minimalnym okresem słowa jest najkrótszy okres tego słowa — w powyższym przykładzie, najkrótszym okresem jest *abc*.

Przypomnijmy teraz definicję tablicy prefiksowej p wyznaczonej dla słowa s o długości n — $p[x]$, $x \in \{1, 2, \ldots, n\}$ jest równe długości najdłuższego właściwego prefikso-sufiksu słowa $s_1 s_2 \ldots s_x$. Korzystając z tej definicji, można w prosty sposób pokazać, że długość minimalnego okresu słowa s jest równa $n - p[n]$.

Funkcja **int** `MinPeriod(`**const char***`)` przedstawiona na listingu 7.4 przyjmuje jako parametr tekst, a zwraca długość jego minimalnego okresu. Przykład jej użycia został przedstawiony na listingu 7.5.

Listing 7.4: Implementacja funkcji **int** `MinPeriod(`**const char***`)`

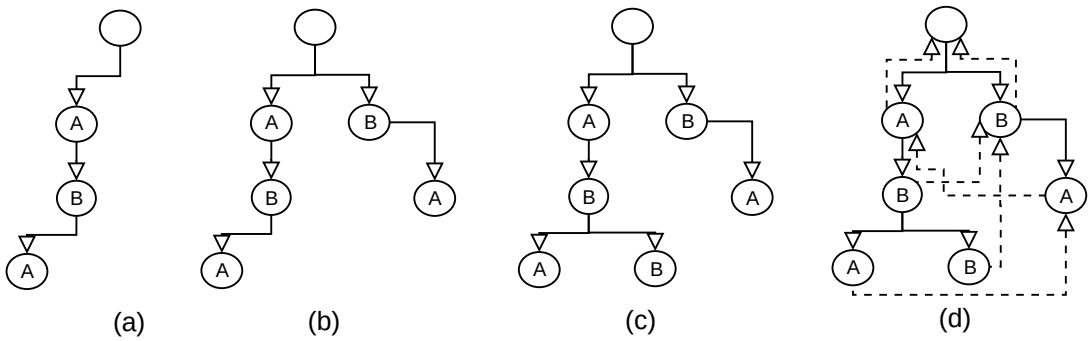
```
// Funkcja wyznacza długość minimalnego okresu słowa
01 int MinPeriod(const char *s) {
02    int p[strlen(s--) + 1], k = 0, q;
03    p[1] = 0;
// Dla tekstu s wyznacz funkcję prefiksową
04    for (q = 2; s[q]; q++) {
05       while (k > 0 && s[k + 1] != s[q]) k = p[k];
06       if (s[k + 1] == s[q]) k++;
07       p[q] = k;
08    }
// Minimalny okres tekstu s jest równy długości s pomniejszonej o długość
// maksymalnego prefikso-sufiksu s
09    return q - p[q - 1] - 1;
10 }
```

Listing 7.5: Przykład użycia funkcji **int** `MinPeriod(`**const char***`)`

```
MinPeriod(abcadc) = (abcadc)
MinPeriod(abcabc) = (abc)
MinPeriod(aaaaaa) = (a)
MinPeriod(ababa) = (ab)
```

Listing 7.6: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.5. Pełny kod źródłowy programu znajduje się w pliku `minperiod.cpp`

```
1 int main() {
2    string text;
// Wczytaj kolejne teksty
3    while(cin >> text) {
// Wyznacz długość minimalnego okresu tekstu oraz wypisz znaleziony wynik
4       cout << "MinPeriod(" << text << ") = " << "(" <<
5          text.substr(0,MinPeriod(text.c_str())) << ")" << endl;
6    }
7    return 0;
8 }
```



Rysunek 7.1: **(a)** Postać struktury `mkmp` po dodaniu wzorca ABA. **(b)** Postać struktury po dodaniu wzorca BA **(c)** Postać struktury po dodaniu kolejnego wzorca ABB. **(d)** Wyznaczenie funkcji prefiksowej

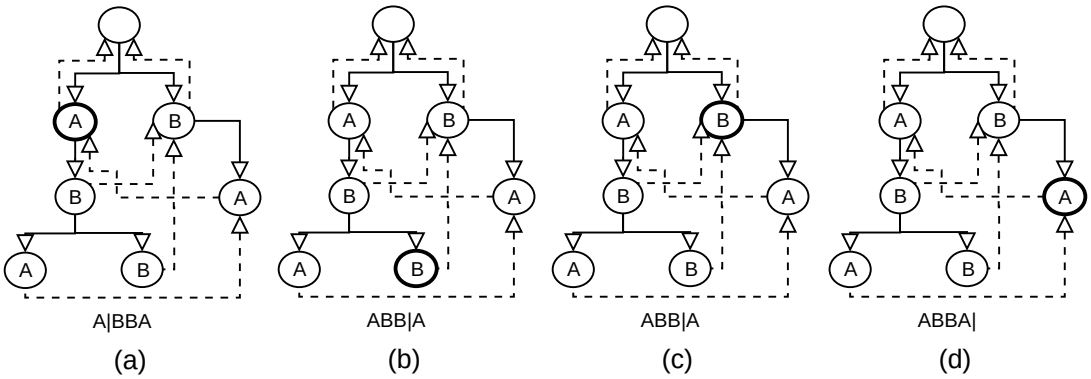
7.3. KMP dla wielu wzorców (algorytm Aho-Corasick)

Literatura
[EAT] - 7.1, 7.2
[ASD] - 5.1.3

Algorytm KMP do wyszukiwania wystąpień wzorca w w zadanym tekście t wykorzystuje do swojego działania tablicę prefiksową. Dzięki niej jest on w stanie w czasie stałym, znając najdłuższy sufiks tekstu $t_1t_2 \dots t_{k-1}$, będący równocześnie prefiksem wzorca w (oznaczmy go przez p_{k-1}), wyznaczać odpowiedni sufiks p_k tekstu $t_1t_2 \dots t_k$. Znalezienie wzorca w tekście jest w ten sposób równoważne z wyznaczeniem sufiksu $p_{|w|}$. Można się w takiej sytuacji zastanowić, co by było w przypadku wyszukiwania na raz wielu wzorców w tekście. Widać, że jest to możliwe pod warunkiem istnienia sposobu na wyznaczanie odpowiedniego sufiksu p_m, który tym razem musiałby być prefiksem dowolnego z wyszukiwanych wzorców. Okazuje się, że jest to rzeczywiście możliwe — algorytm ten znany jest pod nazwą algorytmu Aho-Corasick.

Przed rozpoczęciem wyszukiwania wzorców w tekście, podobnie jak klasyczny algorytm KMP, algorytm Aho-Corasick musi najpierw wyznaczyć postać tablicy prefiksowej, która w tym przypadku ma bardziej skomplikowaną strukturę (i przestaje być tablicą, więc od tej pory będziemy ją nazywać funkcją prefiksową). W tym celu budowane jest drzewo ze wszystkich wyszukiwanych wzorców (przykład przedstawiony jest na rysunkach 7.1.a-c). Kolejnym krokiem jest wyznaczenie funkcji prefiksowej — dla każdego węzła drzewa reprezentującego tekst t_1 wyznaczany jest węzeł reprezentujący tekst t_2 taki, że t_2 jest najdłuższym właściwym sufiksem tekstu t_1 spośród wszystkich tekstów posiadających wierzchołki w drzewie (rys. 7.1.d). Zauważmy, że w przypadku drzewa reprezentującego jeden wzorzec, wyznaczone pary wierzchołków (t_1, t_2) są dokładnym odpowiednikiem tablicy prefiksowej z algorytmu KMP. W tym momencie możliwe jest już przystąpienie do procesu przeszukiwania tekstu. Dla kolejnych liter tekstu wykonywane są przesunięcia wzdłuż odpowiednich krawędzi w drzewie. W przypadku, gdy nie istnieje krawędź dla kolejnej analizowanej litery a, wykonywane są kroki „w tył” wzdłuż funkcji prefiksowej tak długo, aż trafi się na wierzchołek posiadający krawędź dla litery a, lub wróci się do korzenia. Przykładowy proces wyszukiwania wzorca przedstawiony jest na rysunku 7.2.

Przedstawiona na listingu 7.7 struktura `mkmp` realizuje opisany algorytm Aho-Corasick. Udostępnia ona funkcje **int** `addWord(`**char***`)`, służącą do dodawania kolejnego wzorca oraz **void** `calcLink()`, której zadaniem jest wyznaczanie funkcji prefiksowej (należy ją wywołać po dodaniu do struktury ostatniego wzorca). Czas dodawania wzorca do struktury jest liniowy ze względu na jego długość, natomiast wyznaczanie funkcji prefiksowej jest liniowe ze względu na



Rysunek 7.2: **(a)** Rozpoczęcie przeszukiwania tekstu $ABBA$. Po wczytaniu pierwszej litery A aktywnym wierzchołkiem jest syn korzenia z literą A. **(b)** Wczytano kolejne dwie litery. Został znaleziony wzorzec ABB. **(c)** Kolejną literą tekstu jest A. Ponieważ aktualny wierzchołek drzewa nie ma syna dla tej litery, zatem wykonywane jest przejście zgodne z funkcją prefiksową. **(d)** Przetworzenie ostatniej litery i znalezienie wzorca BA.

liczbę wierzchołków w drzewie (która jest ograniczona przez sumaryczną długość wszystkich wzorców).

Do przeszukiwania tekstu służą dwie funkcje: **void** `SearchAll(`**const char***, **void** `(*)(`**int, int**`))` oraz **void** `SearchFirst(`**const char***, **void** `(*)(`**int, int**`))`. Obie z nich przyjmują jako parametry przeszukiwany tekst w oraz wskaźnik na funkcję f, która jest wywoływana dla każdego znalezionego wystąpienia wzorca w tekście. Funkcja f jako parametry otrzymuje odpowiednio numer znalezionego wzorca oraz jego pozycję w tekście. Różnica między tymi dwoma funkcjami polega na tym, że pierwsza z nich wyznacza wszystkie wystąpienia wzorców, podczas gdy druga dla każdego wzorca wyznacza tylko pierwsze jego wystąpienie. Złożoność algorytmów realizowanych przez te funkcje to $O(n+m)$, gdzie n to długość przeszukiwanego tekstu, a m to liczba znalezionych wystąpień wzorca.

Listing 7.7: Implementacja struktury `mkmp`

```
01 struct mkmp {
// Struktura reprezentująca wierzchołek w drzewie wzorców
02   struct leaf {
// Mapa son służy do reprezentowania krawędzi wychodzących z wierzchołka
03     map<char, leaf *> son;
// Element lnk wierzchołka v reprezentuje wartość funkcji prefiksowej
// wierzchołka, natomiast wo jest wskaźnikiem na najdłuższy wzorzec będący
// sufiksem tekstu wierzchołka v
04     leaf *lnk, *wo;
// Zmienna reprezentująca numer wzorca wierzchołka (jeśli wierzchołek nie
// reprezentuje żadnego wzorca, jest to -1)
05     int el;
06     leaf() : el(-1) { }
07   };
// Wektor zawierający długości poszczególnych wzorców wstawionych do struktury
08   VI len;
// Korzeń drzewa
09   leaf root;
// Pomocnicza funkcja przetwarzająca literę tekstu i dokonująca odpowiedniego
```

Listing 7.7: (c.d. listingu z poprzedniej strony)

```
// przejścia w drzewie
10    leaf *mv(leaf * w, char l) {
// Dopóki wierzchołek w nie ma syna dla litery l przesuwaj się wzdłuż
// funkcji prefiksowej
11        while (w != &root && w->son.find(l) == w->son.end()) w = w->lnk;
// Jeśli wierzchołek w posiada syna dla litery l, to przejdź do tego
// wierzchołka
12        if (w->son.find(l) != w->son.end()) w = w->son[l];
13        return w;
14    }
// Funkcja wstawia nowy wzorzec do struktury
15    int addWord(const char *s) {
16        int l = strlen(s);
17        leaf *p = &root;
// Przejdź od korzenia drzewa do wierzchołka reprezentującego cały wzorzec,
// ewentualnie tworząc jeszcze nieistniejące wierzchołki
18        for (; *s; ++s) {
19            VAR(e, p->son.find(*s));
20            p = (e == p->son.end())? p->son[*s] = new leaf : e->second;
21        }
// Jeśli aktualny wierzchołek nie reprezentuje jeszcze żadnego wzorca, to
// przypisz mu nowy identyfikator i zapamiętaj jego długość (wpp.
// wykorzystywany jest poprzedni identyfikator)
22        if (p->el == -1) {
23            p->el = SIZE(len);
24            len.PB(l);
25        }
// Zwróć identyfikator wzorca
26        return p->el;
27    }
// Funkcja wyznacza funkcję prefiksową dla wierzchołków drzewa. Wylicza ona
// również wartości pól wo
28    void calcLink() {
// Wektor l jest używany jako kolejka dla przeszukiwania drzewa metodą BFS
29        vector<leaf *> l;
30        leaf *w;
31        root.lnk = root.wo = 0;
// Wstaw do kolejki wszystkich synów korzenia oraz ustaw ich funkcję
// prefiksową na korzeń
32        FOREACH(it, root.son) {
33            l.PB(it->ND);
34            it->ND->lnk = &root;
35        }
// Dla wszystkich elementów z kolejki...
36        REP(x, SIZE(l)) {
// Wyliczenie pola wo - jego wartość to wierzchołek wskazywany przez
// funkcję prefiksową (jeśli reprezentuje on wzorzec), lub w przeciwnym
// przypadku wartość pola wo tego wierzchołka
```

Listing 7.7: (c.d. listingu z poprzedniej strony)

```
37          l[x]->wo = (l[x]->lnk->el != -1) ? l[x]->lnk : l[x]->lnk->wo;
// Dla każdego syna aktualnego wierzchołka (literę prowadzącą do syna
// reprezentuje it->ST)...
38          FOREACH(it, l[x]->son) {
// Wstaw go do kolejki
39            l.PB(it->ND);
// Wyznaczanie wartość funkcji prefiksowej
40            w = l[x]->lnk;
// Dokonaj przejścia z wierzchołka w
41            w = mv(w, it->ST);
// Ustaw wierzchołek w jako wartość funkcji prefiksowej
42            it->ND->lnk = w;
43          }
44        }
45      }
// Funkcja wyszukuje wszystkie wystąpienia wzorców w zadanym tekście i dla
// każdego znalezionego wystąpienia wywołuje funkcję fun
46      void searchAll(const char *s, void (*fun) (int, int)) {
47        leaf *p = &root;
// Dla każdej kolejnej litery przeszukiwanego tekstu...
48        for (int x = 0; s[x]; ++x) {
// Dokonaj przejścia dla litery s[x]
49          p = mv(p, s[x]);
// Jeśli aktualny wierzchołek (lub pewne jego sufiksy) reprezentują wzorzec,
// to wywołaj dla nich funkcję fun
50          for (VAR(r, p); r; r = r->wo)
51          if (r->el != -1) fun(x - len[r->el] + 1, r->el);
52        }
53      }
// Funkcja wyszukuje pierwsze wystąpienia każdego ze wzorców w zadanym tekście i
// dla każdego znalezionego wystąpienia wywołuje funkcję fun
54      void searchFirst(const char *s, void (*fun) (int, int)) {
55        leaf *p = &root, *r, *t;
// Dla każdej kolejnej litery przeszukiwanego tekstu...
56        for (int x = 0; s[x]; ++x) {
// Dokonaj przejścia dla litery s[x]
57          p = mv(p, s[x]);
// Wywołaj funkcję fun dla wszystkich znalezionych wzorców oraz usuń
// informacje o tych wzorcach, aby nie zostały znalezione ponownie
58          r = p;
59          while (r) {
// Jeśli wierzchołek reprezentuje wzorzec, to wywołaj funkcję fun
60            if (r->el != -1) fun(x - len[r->el] + 1, r->el);
// Usuń identyfikator wzorca, aby nie został on znaleziony po raz kolejny
61            r->el = -1;
// Przejdź do kolejnego wzorca i wyzeruj wskaźnik wo (aby nie wykonywać
// tego ruchu w przyszłości)
62            t = r;
```

Listing 7.7: (c.d. listingu z poprzedniej strony)

```
63            r = r->wo;
64            t->wo = 0;
65          }
66        }
67    }
68 };
```

Listing 7.8: Przykład wykorzystania struktury `mkmp`

```
Dodawanie wzorca ABA
Dodawanie wzorca ABB
Dodawanie wzorca BA
Przeszukiwanie tekstu ABABCABBCABCA
Znaleziono wzorzec 0 na pozycji 0
Znaleziono wzorzec 2 na pozycji 1
Znaleziono wzorzec 1 na pozycji 5
```

Listing 7.9: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.8. Pełny kod źródłowy programu znajduje się w pliku `mkmp.cpp`

```
// Funkcja wywoływana dla każdego znalezionego wzorca
01 void Disp(int x, int p) {
02    cout << "Znaleziono wzorzec " << p << " na pozycji " << x << endl;
03 }
04 int main() {
05    mkmp search;
06    string text;
07    int n;
// Wczytaj liczbę wzorców
08    cin >> n;
// Wczytaj kolejne wzorce
09    REP(x, n) {
10       cin >> text;
11       cout << "Dodawanie wzorca " << text << endl;
12       search.addWord(text.c_str());
13    }
// Wyznacz funkcję prefiksową
14    search.calcLink();
15    cin >> text;
// Przeszukaj zadany tekst
16    cout << "Przeszukiwanie tekstu " << text << endl;
17    search.searchAll(text.c_str(), Disp);
18    return 0;
19 }
```

Zadanie: MegaCube

Pochodzenie: V Obóz Olimpiady Informatycznej im. A. Kreczmara

Rozwiązanie: `megacube.cpp`

Grupa studentów pewnej uczelni, zafascynowana niedawno obejrzanym filmem — Cube2-Hypercube, postanowiła nakręcić kolejną serię tego wspaniałego filmu. Z uwagi na niski budżet, jedyne na co mogą sobie pozwolić, jest dwuwymiarowa kostka (skoro dwuwymiarowa, to niech będzie przynajmniej prostokątna). W celu urozmaicenia filmu, każda komnata zostanie pomalowana na jeden z dostępnych kolorów. Główni bohaterowie filmu jako wskazówkę, umożliwiającą im wydostanie się z labiryntu, otrzymają mapę z zaznaczonymi kolorami poszczególnych komnat. Bohaterowie będą chodzić po labiryncie i starać się zlokalizować swą pozycję na podstawie kolorów odwiedzanych komnat oraz mapy. Ponieważ jeden z aktorów przez przypadek zabrał ze sobą komputer więc postanowił napisać program, który pozwoli określić pozycję w labiryncie. Pomóż mu w tym niebanalnym zadaniu.

Zadanie

Napisz program, który:

- Wczyta opis labiryntu, po którym poruszają się bohaterowie oraz obszar zbadany przez nich (znają tylko pewien obszar labiryntu, który jest kształtu prostokąta),
- wyliczy, w ilu różnych miejscach mogą się znajdować bohaterowie filmu,
- wypisze wynik na standardowe wyjście.

Wejście

W pierwszym wierszu standardowego wejścia zapisane są cztery liczby naturalne c_x, c_y, m_x, m_y ($1 \leqslant m_x \leqslant c_x \leqslant 1\,000$, $1 \leqslant m_y \leqslant c_y \leqslant 1\,000$); c_x i c_y to odpowiednio wymiary labiryntu (czyli również jego mapy); a m_x i m_y to wymiary zwiedzonego obszaru. W kolejnych c_y wierszach znajduje się po c_x liczb naturalnych (nie większych od 100), reprezentujących kolory poszczególnych pomieszczeń na mapie. Następnie znajduje się m_y wierszy zawierających po m_x liczb naturalnych (podobnie nie większych od 100), reprezentujących kolory kolejnych pomieszczeń zwiedzonych przez bohaterów.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia powinna się znajdować jedna liczba naturalna, odpowiadająca liczbie miejsc, w których mogą się znajdować bohaterowie.

Przykład

Dla następującego wejścia:

```
5 3 2 2
1 2 2 1 2
2 3 1 2 3
1 2 2 3 3
1 2
2 3
```

Poprawnym rozwiązaniem jest:

```
3
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10010 spoj.sphere.pl - zadanie 48	acm.uva.es - zadanie 11019 spoj.sphere.pl - zadanie 263 acm.uva.es - zadanie 10298	acm.uva.es - zadanie 10679 spoj.sphere.pl - zadanie 413

7.4. Promienie palindromów w słowie

Palindromem nazywamy słowo, które czytane zarówno od początku, jak i od końca jest takie samo. Każde słowo o długości 1 jest palindromem. Odpowiedzenie na pytanie, czy dane słowo jest palindromem, czy nie, jest proste. Znacznie trudniejszym problemem jest znalezienie w danym tekście wszystkich palindromów. Postawmy sobie następujący problem — dla każdej litery tekstu należy obliczyć maksymalny promień palindromu o środku w tej literze. Przykładowo, dla tekstu *abbabbb*, poszukiwanym promieniem dla 4-tej litery jest 2, natomiast dla litery 6-tej jest 1.

Literatura
[EAT] - 8.3
[ASD] - 5.3.4

Poszukiwane promienie można wyznaczyć w czasie liniowym. Oznaczmy przetwarzany tekst przez $t = t_1 t_2 \dots t_n$, natomiast odpowiednie promienie palindromów tego tekstu przez $p_1, p_2, \dots, p_n$. Załóżmy, że wyznaczone zostały już promienie $p_1, p_2, \dots, p_k$. Własność, która pozwala na efektywne wyznaczanie kolejnych promieni ma postać:

$$p_{k+l} = min(p_{k-l}, p_k - l), l \in \{1, 2, \dots\} \text{ , jeśli } p_{k+l} \leqslant p_k - l$$

Dodatkowo, p_{k+l} jest nie mniejsze od $min(p_{k-l}, p_k - l)$. Złączenie tych dwóch prostych własności pozwala już na efektywne wyznaczenie wyniku.

Algorytm wyznaczający promienie palindromów zrealizowany został jako funkcja `VI PalRad(const char*, bool)`, której kod źródłowy został przedstawiony na listingu 7.10. Funkcja ta przyjmuje jako parametry tekst, dla którego wyznaczane są promienie palindromów oraz wartość logiczną p. W przypadku, gdy p ma wartość prawda, algorytm wyznacza promienie dla palindromów nieparzystych — takich, których środkiem jest litera tekstu. W przypadku, gdy p ma wartość fałsz, wyznaczane są promienie palindromów parzystych, czyli takich, które nie posiadają w środku niesparowanej litery (patrz przykład z listingu 7.11). Przedstawiona metoda znana jest jako algorytm Manachera.

Listing 7.10: Implementacja funkcji `VI PalRad(const char*, bool)`

```
// Funkcja wylicza promienie palindromów danego tekstu
01 VI PalRad(const char *x, bool p)
02 {
03    int n = strlen(x), i = 1, j = 0, k;
04    VI r(n, 0);
05    while(i < n) {
// Dopóki kolejno sprawdzane litery palindromu o środku na pozycji
// i są takie same, zwiększ promień
06       while(i + j + p < n && i > j && x[i - j - 1] == x[ i + j + p]) j++;
// Stosując zależności między promieniami palindromów wyznacz
// jak najwięcej kolejnych promieni
07       for(r[i] = j, k = 0; ++k <= j && r[i - k] != j - k;)
08          r[i + k] = min(r[i - k], j - k);
09       j = max(0, j - k);
```

Listing 7.10: (c.d. listingu z poprzedniej strony)

```
10        i += k;
11    }
12    return r;
13 }
```

Listing 7.11: Przykład wykorzystania funkcji `VI PalRad(const char*, bool)`

```
Analizowany tekst: aabaaabaabaca
Palindromy parzyste:   0 1 0 0 1 1 0 0 3 0 0 0 0
Palindromy nieparzyste:  0 0 2 0 4 0 2 0 0 1 0 1 0
```

Listing 7.12: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.11. Pełny kod źródłowy programu znajduje się w pliku `palrad.cpp`

```
01 int main() {
02    string text;
// Wczytaj tekst
03    cin >> text;
04    cout << "Analizowany tekst: " << text << endl;
// Wyznacz promienie palindromów parzystych
05    VI res = PalRad(text.c_str(), 0);
06    cout << "Palindromy parzyste: ";
07    FOREACH(it, res) cout << " " << (*it);
08    cout << endl;
// Wyznacz promienie palindromów nieparzystych
09    res = PalRad(text.c_str(), 1);
10    cout << "Palindromy nieparzyste: ";
11    FOREACH(it, res) cout << " " << (*it);
12    cout << endl;
13    return 0;
14 }
```

Zadanie: Palindromy

Pochodzenie: II Olimpiada Informatyczna

Rozwiązanie: `pal.cpp`

Słowo jest palindromem wtedy i tylko wtedy, gdy nie zmienia się, jeśli czytamy je wspak. Palindrom jest parzysty, gdy ma dodatnią parzystą liczbę liter.
Przykładem parzystego palindromu jest słowo *abaaba*.
Rozkładem słowa na palindromy parzyste jest jego podział na rozłączne części, z których każda jest palindromem parzystym.
Rozkładami słowa:
bbaabbaabbbaaaaaaaaaaaabbbaa
na palindromy parzyste są:
bbaabb aabbbaaaaaaaaaaaabbbaa
oraz:

bb aa bb aa bb baaaaaaaaaaaab bb aa

Pierwszy rozkład zawiera najmniejszą możliwą liczbę palindromów, drugi jest rozkładem na maksymalną możliwą liczbę palindromów parzystych. Zauważ, że słowo może mieć wiele różnych rozkładów na palindromy parzyste, albo nie mieć żadnego.

Zadanie

Napisz program, który:

- wczytuje słowo,
- bada, czy da się je rozłożyć na palindromy parzyste,
  - jeśli nie, to wypisuje słowo NIE
  - jeśli tak, to zapisuje jego rozkłady na minimalną i maksymalną liczbę palindromów parzystych.

Wejście

W pierwszym wierszu wejścia znajduje się jedno słowo złożone z co najmniej 1 i co najwyżej 200 małych liter alfabetu angielskiego, zapisane w jednym wierszu bez odstępów pomiędzy literami.

Wyjście

Twój program powinien wypisać:

- tylko jeden wyraz NIE
- albo:
  - w pierwszym wierszu ciąg słów oddzielonych odstępami stanowiący jeden rozkład danego słowa na minimalną liczbę palindromów parzystych,
  - w drugim wierszu jeden rozkład danego słowa na maksymalną liczbę palindromów parzystych.

Przykład

Dla następującego wejścia:

```
bbaabbaabbbaaaaaaaaaaaabbbaa
```

Poprawnym rozwiązaniem jest:

```
bbaabb aabbbaaaaaaaaaaaabbbaa
bb aa bb aa bb baaaaaaaaaaaab bb aa
```

Dla następującego wejścia:

```
abcde
```

Poprawnym rozwiązaniem jest:

```
NIE
```

Dla następującego wejścia:

```
abaaba
```

Poprawnym rozwiązaniem jest:

```
abaaba
abaaba
```

7.5. Drzewa sufiksowe

Literatura
[EAT] - 5
[CST]
[ASD] - 5.2

Drzewo sufiksowe jest strukturą danych, umożliwiającą reprezentację wszystkich sufiksów danego słowa. Prosta postać takiego drzewa ma rozmiar kwadratowy ze względu na długość słowa. Przykład został przedstawiony na rysunku 7.3.a. Drzewa sufiksowe stanowią bardzo pożyteczną strukturę danych, przy użyciu której można rozwiązać (zakładając, że drzewo sufiksowe jest już skonstruowane) bardzo wiele problemów tekstowych. Przykładowo, wyszukiwanie wzorca w tekście można zrealizować przechodząc od korzenia drzewa sufiksowego po krawędziach reprezentujących kolejne litery wyszukiwanego wzorca.

Wyznaczenie drzewa sufiksowego w takiej postaci, w jakiej jest ono przedstawione na rysunku 7.3.a nie jest trudne. Wystarczy dobudowywać odpowiednie wierzchołki drzewa dla kolejno przetwarzanych sufiksów tekstu. Skonstruowane w ten sposób drzewo ma niestety kwadratowy rozmiar. Istnieje jednak możliwość przedstawienia drzewa sufiksowego w innej postaci — skompresowanej. W drzewie takim spójne ścieżki bez rozgałęzień w wierzchołkach zostają zamienione przez jedną krawędź. Przykład takiego drzewa przedstawiony jest na rysunku 7.3.b. Drzewo tego typu ma co najwyżej dwa razy więcej wierzchołków, niż wynosi długość tekstu, dla którego zostało ono zbudowane. Konstrukcję skompresowanego drzewa sufiksowego można przeprowadzić w dwóch fazach — najpierw wyznaczyć nieskompresowane drzewo sufiksowe, a następnie zamienić w nim wszystkie spójne ścieżki bez rozgałęzień na odpowiednie krawędzie. Wynikiem takiego algorytmu jest drzewo o liniowej wielkości, ale proces konstrukcji cały czas działa w czasie kwadratowym.

Okazuje się jednak, że istnieje inny sposób konstruowania drzew sufiksowych, działający w czasie liniowym. Jedną z takich metod jest algorytm Ukkonena, którego opis można znaleźć w [CST]. Przedstawiona na listingu 7.13 implementacja struktury `SufTree` jest realizacją algorytmu opisanego w [ASD] (w celu dokładnej analizy działania algorytmu należy odwoływać się do literatury).

Drzewo sufiksowe `SufTree` zbudowane dla tekstu t składa się z wierzchołków o typie `SufV`. Każdy wierzchołek zawiera pola **int** `p` oraz **int** `k`, które określają podsłowo $t_p t_{p+1} \dots t_{k-1}$, będące etykietą krawędzi prowadzącej do tego wierzchołka. Oprócz tego, pole **bool** `s` oznacza, czy wierzchołek reprezentuje pełny sufiks (pogrubione wierzchołki na rysunku 7.3.b). Istnieje możliwość wzbogacenia każdego wierzchołka o dodatkowe pole `e`, w którym można przechowywać specjalne informacje zależne od algorytmów, do których wykorzystywane jest drzewo sufiksowe.

Listing 7.13: Implementacja struktury `SuffixTree`

```
// Drzewo sufiksowe
01 template<typename _T> struct SufTree {
// Struktura reprezentująca węzeł drzewa
02   struct SufV {
// Mapa synów węzła
03     map<char, SufV*> sons;
// Początek oraz koniec tekstu reprezentowanego przez krawędź prowadzącą do węzła
```

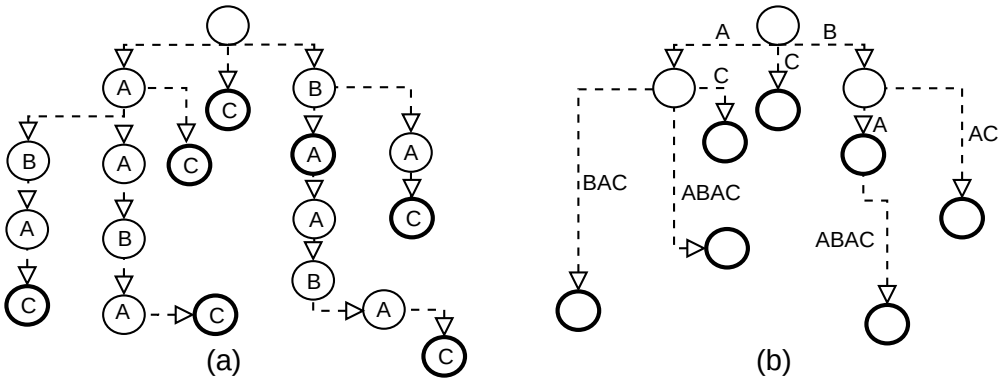
Listing 7.13: (c.d. listingu z poprzedniej strony)

```
04      int p, k;
// Czy węzeł reprezentuje sufiks słowa
05      bool s;
// Obiekt reprezentuje dodatkowe wzbogacenie węzła, które może być wykorzystywane
// przez algorytmy korzystające z drzew sufiksowych
06      _T e;
// Wskaźnik na ojca w drzewie
07      SufV *par;
// Konstruktor węzła
08      SufV(int p, int k, SufV *par, bool s) : p(p), k(k), par(par), s(s) {}
// Destruktor węzła usuwa wszystkich synów
09      ~SufV() {
10        FOREACH(it, sons) delete it->second;
11      }
12    };
// Struktura wykorzystywana do konstrukcji drzewa sufiksowego - służy do
// reprezentacji tablic lnk i test
13    struct VlP {
14      SufV *p;
15      char l;
16      VlP(SufV *p, char l) : p(p), l(l) {}
17      bool operator<(const VlP &a) const {
18        return (a.p > p) || (a.p == p && a.l > l);
19      }
20    };
// Korzeń drzewa
21    SufV root;
// Tekst, dla którego zostało zbudowane drzewo sufiksowe
22    string text;
23    SufTree(const string& t) : root(0, 0, 0, 0), text(t) {
24      map<VlP, SufV*> lnk;
25      set<VlP> test;
26      int len = t.length();
// Stwórz pierwszy węzeł drzewa reprezentujący ostatnią literę słowa
27      SufV *v = root.sons[t[len - 1]] = new SufV(len - 1, len, &root, 1);
28      lnk[VlP(&root, t[len - 1])] = v;
29      test.insert(VlP(&root, t[len - 1]));
// Dla kolejnych sufiksów słowa (od najkrótszych do najdłuższych)...
30      FORD(i, len - 2, 0) {
31        char a = t[i];
// Jeśli z korzenia nie wychodzi krawędź dla litery słowa na pozycji i...
32        if (!root.sons[a]) {
// Następuje aktualizacja tablicy test dla węzłów na ścieżce od węzła
// reprezentującego poprzedni sufiks do korzenia
33          SufV* it = v;
34          while(it) {
35            test.insert(VlP(it, a));
36            it = it->par;
```

Listing 7.13: (c.d. listingu z poprzedniej strony)

```
37            }
38            it = v;
// Dodanie nowego syna dla korzenia
39            lnk[VlP(it, a)] = v = root.sons[t[i]] = new SufV(i, len, &root, 1);
40          } else {
// Przy wykorzystaniu tablic test oraz lnk następuje wyznaczenie krawędzi drzewa,
// którą trzeba podzielić w celu dodania kolejnego sufiksu
41            char lw;
42            SufV *head, *head2, *x, *x1;
43            int lw2 = 0, gl = 0;
44            for(x = v; x != &root && test.find(VlP(x, a)) == test.end();
45              x = x->par) lw2 += x->k - x->p;
46            for(x1 = x; x1 && !lnk[VlP(x1, a)]; x1 = x1->par) {
47              gl += x1->k - x1->p;
48              lw = t[x1->p];
49            }
50            if (x1) gl--;
// Następuje podział krawędzi drzewa sufiksowego
51            SufV* head1 = x1 ? lnk[VlP(x1, a)] : &root;
52            if (x == x1) head = head1; else {
53              head2 = (!x1) ? root.sons[a] : head1->sons[lw];
54              head = head1->sons[t[head2->p]] =
55                new SufV(head2->p, head2->p + 1 + gl, head1, 0);
56              head->sons[t[head->k]] = head2;
57              head2->p = head->k;
58              head2->par = head;
59              for(VAR(it, test.lower_bound(VlP(head2, 0))); it->p == head2;)
60                test.insert(VlP(head, (it++)->l));
61            }
// Aktualizacja zawartości tablic test oraz lnk
62            for(SufV* it = v; it != x1; it = it->par) test.insert(VlP(it, a));
63            lnk[VlP(x, a)] = head;
64            SufV *v1 = v;
65            v = head->sons[t[len - lw2]] = new SufV(len - lw2, len, head, 1);
66            lnk[VlP(v1, a)] = v;
67          }
68        }
69      }
70  };
```

W kliku kolejnych rozdziałach zostanie przedstawiony sposób wykorzystania drzew sufiksowych. Przy ich użyciu zostaną rozwiązane problemy zliczania liczby wystąpień wzorca w tekście, wyznaczania liczby różnych podsłów czy szukania najdłuższego słowa występującego w tekście n razy. Wszystkie te problemy daje się rozwiązać w czasie liniowym.



Rysunek 7.3: **(a)** Nieskompresowane drzewo sufiksowe zbudowane dla tekstu $BAABAC$. Pogrubione węzły reprezentują sufiksy tekstu. **(b)** Skompresowane drzewo sufiksowe zbudowane dla tego samego tekstu.

7.5.1. Liczba wystąpień wzorca w tekście

Jednym z możliwych zastosowań drzew sufiksowych jest wyznaczanie liczby wystąpień wzorca w tekście.

Załóżmy, że chcemy wyznaczyć liczbę wystąpień wzorca p w tekście t. Jeśli do tego celu wykorzystamy nieskompresowane drzewo sufiksowe, to wystarczy znaleźć wierzchołek v drzewa sufiksowego, reprezentujący wzorzec p, a następnie wyznaczyć liczbę wierzchołków osiągalnych z v, które reprezentują sufiksy tekstu t. Przykładowo, rozpatrując drzewo sufiksowe dla tekstu $BAABAC$ przedstawione na rysunku 7.3.a oraz wzorzec A, poszukiwanym wierzchołkiem jest syn korzenia drzewa z etykietą A. Z wierzchołka tego osiągalne są trzy wierzchołki reprezentujące sufiksy, zatem liczba wystąpień litery A w tekście jest równa 3.

Przenosząc się teraz do skompresowanej postaci drzewa sufiksowego, można zastosować ten sam algorytm. Różni się tylko sposób wyznaczania wierzchołka v. W tym przypadku wyznaczając pozycję w drzewie sufiksowym reprezentującą wzorzec p, należy brać pod uwagę napisy umieszczone na krawędziach, a nie w wierzchołkach.

Pozostaje jeszcze jeden problem — aby było można wyznaczyć liczbę wystąpień wzorca p w czasie $O(|p|)$, musi istnieć możliwość szybkiego wyliczania liczby osiągalnych wierzchołków reprezentujących sufiksy. Można tego dokonać spamiętując wcześniej (przed przystąpieniem do przetwarzania wzorców) liczbę tych wierzchołków dla każdego węzła drzewa sufiksowego.

Na listingu 7.14 przedstawiony jest kod realizujący opisany pomysł. Przed przystąpieniem do zliczania liczby wystąpień wzorca, należy wywołać funkcję **int** `STSuffixC(SufTree<int>::SufV& v)`, przekazując jej jako parametr korzeń drzewa sufiksowego. Po zakończeniu tej fazy, wyznaczanie liczby wystąpień wzorca można wykonywać przy użyciu **int** `STFindCount(SufTree<int>&, const char*)`, której parametrami są drzewo sufiksowe dla tekstu oraz wyszukiwany wzorzec.

Listing 7.14: Implementacja funkcji **int** `STFindCount(SufTree<int>&, const char*)`

```
// Funkcja wylicza dla każdego węzła drzewa sufiksowego liczbę osiągalnych
// wierzchołków, które reprezentują sufiks słowa
01 int STSuffixC(SufTree<int>::SufV & v) {
02    v.e = v.s;
03    FOREACH(it, v.sons) v.e += STSuffixC(*(it->second));
04    return v.e;
05 }
```

Listing 7.14: (c.d. listingu z poprzedniej strony)

```
// Funkcja wyznacza liczbę wystąpień wzorca t w tekście, dla którego zostało
// skonstruowane drzewo sufiksowe st
06 int STFindCount(SufTree<int> &st, const char *t) {
07    int tp = 0, x;
08    VAR(v, &(st.root));
// Rozpoczynając od korzenia drzewa przesuwaj się po nim zgodnie z wczytywanymi
// literami wzorca
09    while (t[tp]) {
10       if (!(v = v->sons[t[tp]])) return 0;
11       for (x = v->p; x < v->k && t[tp];)
12       if (t[tp++] != st.text[x++]) return 0;
13    }
// Jeśli wzorzec występuje w tekście, to liczba jego wystąpień jest równa
// liczbie sufiksów osiągalnych z aktualnego węzła drzewa
14    return v->e;
15 }
```

Listing 7.15: Przykład wykorzystania funkcji **int** `STFindCount(SufTree<int>&, const char*)`

```
Przeszukiwany tekst: abcabcaaabcad
Wzorczec: abc wystepuje 3 razy
Wzorczec: a wystepuje 6 razy
Wzorczec: bcda wystepuje 0 razy
```

Listing 7.16: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.15. Pełny kod źródłowy programu znajduje się w pliku `stfind.cpp`

```
01 int main() {
02    string text;
// Wczytaj tekst, w którym będą wyszukiwane wzorce
03    cin >> text;
04    cout << "Przeszukiwany tekst: " << text << endl;
// Skonstruuj drzewo sufiksowe
05    SufTree<int> tree(text.c_str());
// Wyznacz liczbę osiągalnych wierzchołków-sufiksów dla każdego wierzchołka drzewa
sufiksowego
06    STSuffixC(tree.root);
// Dla poszczególnych wzorców wyznacz liczbę ich wystąpień w tekście
07    while(cin >> text) cout << "Wzorczec: " << text << " wystepuje " <<
08       STFindCount(tree, text.c_str()) << " razy" << endl;
09    return 0;
10 }
```

7.5.2. Liczba różnych podsłów słowa

Literatura
[ASD] - 5.3.9

Przy użyciu drzew sufiksowych istnieje możliwość wyznaczania liczby różnych podsłów występujących w słowie t. W tym celu przypatrzmy się przykładowemu nieskompresowanemu drzewu sufiksowemu z rysunku 7.3.a. Każdy wierzchołek v w drzewie sufiksowym można utożsamić z tekstem uzyskanym przez konkatenację etykiet wierzchołków na ścieżce od korzenia do v. Oczywistym jest, że każdemu wierzchołkowi został w ten sposób przyporządkowany inny tekst. Co więcej, każde podsłowo t ma przyporządkowany jakiś wierzchołek. Zatem liczba różnych podsłów w t jest równa liczbie wierzchołków w nieskompresowanym drzewie sufiksowym t. Przechodząc teraz do drzewa skompresowanego, wystarczy w tym przypadku wyznaczyć sumaryczną długość wszystkich krawędzi w drzewie.

Algorytm realizujący tę metodę został zaimplementowany jako funkcja **int** `STSubWords(SufTree<int>::SufV&)`, przedstawiona na listingu 7.17. Przyjmuje ona jako parametr drzewo sufiksowe reprezentujące tekst, dla którego wyznaczana jest liczba różnych niepustych podsłów.

Listing 7.17: Implementacja funkcji **int** `STSubWords(SufTree<int>::SufV& v)`

```
// Funkcja wyznacza liczbę różnych podsłów w tekście poprzez zliczanie długości
// krawędzi w drzewie sufiksowym
1 template <typename T> int STSubWords(typename SufTree<T>::SufV & v) {
2    int r = v.k - v.p;
3    FOREACH(it, v.sons) r += STSubWords<T> (*(it->second));
4    return r;
5 }
```

Listing 7.18: Przykład użycia funkcji **int** `STSubWords(SufTree<int>::SufV& v)`

```
test ma 9 podslow
aaaaa ma 5 podslow
abcabcaaabcad ma 69 podslow
```

Listing 7.19: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.18. Pełny kod źródłowy programu znajduje się w pliku `subwords.cpp`

```
1 int main() {
2    string text;
// Wczytaj kolejne teksty
3    while (cin >> text) {
// Zbuduj drzewo sufiksowe
4       SufTree<int> tree(text);
// Wyznacz liczbę różnych podsłów
5       cout << text << " ma " << STSubWords <
6       int >(tree.root) << " podslow" << endl;
7    }
8    return 0;
9 }
```

7.5.3. Najdłuższe podsłowo występujące n razy

Kolejnym zastosowaniem drzew sufiksowych jest wyznaczanie najdłuższego podsłowa, które występuje w słowie co najmniej n razy. Analizując strukturę drzew sufiksowych, szybko można dojść do wniosku, że rozwiązanie tego problemu sprowadza się do wyznaczenia w drzewie najgłębszego wierzchołka (najbardziej oddalonego od korzenia), z którego da się dojść do co najmniej n wierzchołków reprezentujących sufiksy tekstu.

Implementacja tego algorytmu została zrealizowana jako struktura `STLongestWord` przedstawiona na listingu 7.20. Do konstruktora tej struktury należy przekazać tekst t oraz liczbę n. Wynik — wyznaczone najdłuższe podsłowo, można odczytać z pól **int** `p` oraz **int** `k`. Reprezentują one odpowiednio początek i koniec podsłowa tekstu t.

Listing 7.20: Implementacja struktury **struct** `STLongestWord`

```
// Struktura wyznacza najdłuższe podsłowo występujące odpowiednią liczbę razy
01 struct STLongestWord {
// Zmienne p oraz k reprezentują odpowiednio początek oraz koniec
// wyznaczonego słowa
02   int p, k, n;
03   int Find(SufTree<int>::SufV & v, int d) {
// d jest głębokością węzła v
04     d += v.k - v.p;
05     v.e = v.s;
// Zlicz liczbę sufiksów osiągalnych z węzła v
06     FOREACH(it, v.sons) v.e += Find(*(it->ND), d);
// Jeśli liczba sufiksów jest nie mniejsza od c, oraz głębokość aktualnego
// węzła jest większa od długości aktualnie znalezionego słowa, to zaktualizuj
// wynik
07     if (v.e >= n && d > k - p) { k = v.k;
08       p = v.k - d;
09     }
10     return v.e;
11   }
12   STLongestWord(const string & t, int n) : p(0), k(0), n(n) {
// Skonstruuj drzewo sufiksowe oraz wyznacz wynik
13     SufTree<int> tr(t);
14     Find(tr.root, 0);
15   }
16 };
```

Listing 7.21: Przykład użycia struktury `STLongestWord`

```
Podslowo slowa abcabc wystepujace 2 razy:
abc
Podslowo slowa aaaaaaaa wystepujace 3 razy:
aaaaaa
Podslowo slowa baabaabaab wystepujace 3 razy:
baab
```

Listing 7.22: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.21. Pełny kod źródłowy programu znajduje się w pliku `multisubword.cpp`

```
01 int main() {
02    string text;
03    int n;
// Wczytaj dane
04    while(cin >> text >> n) {
// Wyznacz podsłowo słowa text występujące n razy
05       STLongestWord str(text, n);
06       cout << "Podslowo slowa " << text << " wystepujace " << n << " razy: "
07          << endl << text.substr(str.p, str.k-str.p) << endl;
08    }
09    return 0;
10 }
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10234	spoj.sphere.pl - zadanie 220	

7.6. Maksymalny leksykograficznie sufiks

Literatura
[EAT] - 3.1
[ASD] - 5.3.7

Drzewa sufiksowe mają znacznie więcej zastosowań niż te, które przedstawiliśmy w poprzednich rozdziałach. Przy ich użyciu można wyznaczać maksymalny leksykograficznie sufiks słowa. Przykładowo, maksymalnym leksykograficznie sufiksem dla słowa *ababbaabbaba* jest *bbaba*. Wyznaczenie takiego sufiksu wymaga przejścia po drzewie sufiksowym od korzenia do liścia, wybierając za każdym razem krawędź reprezentującą leksykograficznie największy tekst. Podejście takie jest dość skomplikowane ze względu na potrzebę konstruowania drzewa sufiksowego. Istnieje jednak znacznie prostsza metoda — tzw. algorytm Duvala, działający — podobnie do rozwiązania wykorzystującego drzewa sufiksowe, w czasie liniowym. Jego implementacja została zrealizowana jako funkcja **int** `maxSuf(`**const char***`)` przedstawiona na listingu 7.23. Parametrem tej funkcji jest tekst, dla którego wyznaczony ma zostać maksymalny leksykograficznie sufiks, natomiast jako wynik zwracana jest pozycja w tekście, od której zaczyna się wyznaczony maksymalny leksykograficznie sufiks.

Listing 7.23: Implementacja funkcji **int** `MaxSuf(`**const char***`)`

```
// Funkcja wyznacza maksymalny leksykograficznie sufiks słowa
01 int maxSuf(const char *x) {
02    int i = 0, j = 1, k = 1, p = 1;
03    char a, b;
04    while (x[j + k - 1]) {
05       if ((a = x[i + k - 1]) < (b = x[j + k - 1])) {
06          i = j++;
07          k = p = 1;
08       } else if (a > b) {
```

Listing 7.23: (c.d. listingu z poprzedniej strony)

```
09          j += k;
10          k = 1;
11          p = j - i;
12        } else if (a == b && k != p) k++;
13        else {
14          j += p;
15          k = 1;
16        }
17    }
18    return i;
19 }
```

Listing 7.24: Przykład wykorzystania funkcji **int** `MaxSuf(`**const char***`)`

```
Maksymalny sufiks ababbaabbaba = bbaba
Maksymalny sufiks abaabbbabba = bbbabba
Maksymalny sufiks algorytm = ytm
```

Listing 7.25: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.24. Pełny kod źródłowy programu znajduje się w pliku `maxsuf.cpp`

```
1 int main() {
2    string text;
// Wczytaj kolejne teksty i wyznacz dla nich maksymalne leksykograficzne sufiksy
3    while (cin >> text) cout << "Maksymalny sufiks " << text << " = " << text.
4    substr(maxSuf(text.c_str())) << endl;
5    return 0;
6 }
```

7.7. Równoważność cykliczna

Literatura
[EAT] - 15.4
[ASD] - 5.3.5

Dwa słowa $s = s_1 s_2 \dots s_n$ i $t = t_1 t_2 \dots t_l$ są równoważne cyklicznie, jeżeli można przenieść pewien sufiks słowa s na początek, w taki sposób, aby $s = t$. Innymi słowy, istnieje liczba $1 \leqslant k \leqslant n$, taka że $s_k s_{k+1} \dots s_n s_1 s_2 \dots s_{k-1} = t$. Dla danych dwóch słów s i t o długości n, można postawić pytanie, czy są one równoważne cyklicznie. Mając w pamięci algorytm KMP, sformułowanie rozwiązania tego problemu przy jego użyciu jest proste. Jeśli dwukrotnie skonkatenowany tekst s (ss) zawiera jako wzorzec słowo t, to s i t są równoważne cyklicznie.

Rozwiązanie takie jest zarówno proste koncepcyjnie jak i implementacyjnie, jednak wymaga wyznaczenia dla tekstu t tablicy prefiksowej. W przypadku, gdy badane teksty są bardzo długie i nie mamy wystarczająco dużo pamięci, przydatny może się okazać algorytm, który wymaga do działania zaledwie stałej ilości dodatkowej pamięci. Został on zaimplementowany jako funkcja **bool** `CycEq(`**const char***`,` **const char***`)`, której kod źródłowy znajduje się na listingu 7.26. Funkcja ta przyjmuje jako parametry dwa teksty, a zwraca wartość logiczną równą prawdzie wtedy, gdy podane teksty są swoimi równoważnościami cyklicznymi.

Listing 7.26: Implementacja funkcji **bool** cycEq(**const char***, **const char***)

```
// Funkcja sprawdza, czy dwa dane teksty są swoimi równoważnościami cyklicznymi
01 bool cycEq(const char *s1, const char *s2) {
02    int n = strlen(s1), i = 0, j = 0, k = 0;
03    if (n != strlen(s2)) return 0;
04    while (i < n && j < n && k < n) {
05       k = 1;
06       while (k <= n && s1[(i + k) % n] == s2[(j + k) % n]) k++;
07       if (k <= n) if (s1[(i + k) % n] > s2[(j + k) % n]) i += k;
08       else j += k;
09    }
10    return k > n;
11 }
```

Listing 7.27: Przykład wykorzystania funkcji **bool** cycEq(**const char***, **const char***)

```
Slowa babaa i ababa sa rownowazne cyklicznie
Slowa abbab i ababa nie sa rownowazne cyklicznie
Slowa abbab i babaa nie sa rownowazne cyklicznie
```

Listing 7.28: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.27. Pełny kod źródłowy programu znajduje się w pliku `cyceq.cpp`

```
01 int main() {
02    int n;
// Wczytaj liczbę tekstów oraz poszczególne teksty
03    cin >> n;
04    vector<string> text(n);
05    REP(x,n) cin >> text[x];
// Dla każdej pary tekstów wyznacz, czy są one równoważne cyklicznie
06    REP(x, n) REP(y, x)
07       cout << "Slowa " << text[x] << " i " << text[y] <<
08          (cycEq(text[x].c_str(), text[y].c_str()) ? "" : " nie") <<
09          " sa rownowazne cyklicznie" << endl;
10    return 0;
11 }
```

7.8. Minimalna leksykograficznie cykliczność słowa

Problem z poprzedniego rozdziału można rozwiązać w inny sposób. Pomysł polega na wyznaczeniu dla wszystkich tekstów równoważnych cyklicznie, wspólnej reprezentacji. W ten sposób dwa teksty są równoważne cyklicznie, o ile ich reprezentacje są identyczne. Jednym z nasuwających się pomysłów, po przeczytaniu rozdziału dotyczącego wyznaczania maksymalnego leksykograficznie sufiksu, jest wyznaczenie dla słowa maksymalnej leksykograficznej cykliczności...

W tym rozdziale przedstawimy algorytm pozwalający na podejście do problemu od innej

strony — dla danego słowa będzie wyznaczana minimalna leksykograficznie cykliczność. Minimalna leksykograficznie cykliczność słowa s jest to najmniejsze słowo t w porządku leksykograficznym, które jest równoważne cyklicznie ze słowem s. Funkcja **int** `minLexCyc(`**const char***`)` realizująca omawiany algorytm przyjmuje jeden parametrem — słowo. Wynikiem jej działania jest numer pierwszej litery, rozpoczynającej minimalną leksykograficznie cykliczność słowa. Implementacja algorytmu została przedstawiona na listingu 7.29. Jest on modyfikacją algorytmu Duvala do wyznaczania maksymalnego leksykograficznie sufiksu słowa. Zauważmy, że jeżeli dla danego słowa w poszukiwalibyśmy maksymalnego leksykograficznie sufiksu słowa ww, to musiałby się on zaczynać w pierwszej części słowa i mieć długość co najmniej $|w+1|$. Wybierając pierwszych w liter wyznaczonego sufiksu, otrzymalibyśmy maksymalną leksykograficzną cykliczność słowa w, zatem odwracając porządek na zbiorze liter alfabetu oraz postępując w dokładnie taki sam sposób, uzyskalibyśmy minimalną leksykograficznie równoważność w.

Listing 7.29: Implementacja funkcji **int** `minLexCyc(`**const char***`)`

```
// Funkcja wyznacza minimalną leksykograficzną równoważność cykliczną słowa
int minLexCyc(const char *x) {
   int i = 0, j = 1, k = 1, p = 1, a, b, l = strlen(x);
   while (j + k <= (l << 1)) {
      if ((a = x[(i + k - 1) % l]) > (b = x[(j + k - 1) % l])) {
         i = j++;
         k = p = 1;
      } else if (a < b) {
         j += k;
         k = 1;
         p = j - i;
      } else if (a == b && k != p) k++;
      else {
         j += p;
         k = 1;
      }
   }
   return i;
}
```

Listing 7.30: Przykład wykorzystania funkcji **int** `minLexCyc(`**const char***`)`

```
Minimalna rownowaznosc cykliczna aabaaabaabaca = aaabaaabaabac
Minimalna rownowaznosc cykliczna baaabaabacaaa = aaabaaabaabac
Minimalna rownowaznosc cykliczna algorytm = algorytm
```

Listing 7.31: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 7.30. Pełny kod źródłowy programu znajduje się w pliku `minlexcyc.cpp`

```
int main() {
   string text;
// Wczytaj kolejne teksty i wyznacz dla nich minimalną leksykograficznie
// równoważność cykliczną
   while (cin >> text) {
```

Listing 7.31: (c.d. listingu z poprzedniej strony)

```
04      int res = minLexCyc(text.c_str());
05      cout << "Minimalna rownowaznosc cykliczna " << text <<
06      " = " << text.substr(res, text.length() - res) <<
07      text.substr(0, res) << endl;
08    }
09    return 0;
10 }
```

Zadanie: Punkty

Pochodzenie: XII Olimpiada Informatyczna

Rozwiązanie: `points.cpp`

Dany jest zbiór punktów na płaszczyźnie o współrzędnych całkowitych, który będziemy nazywać wzorem oraz zestaw innych zbiorów punktów na płaszczyźnie (również o współrzędnych całkowitych). Interesuje nas, które z podanych zestawów są podobne do wzoru, tzn. czy można je tak przekształcić za pomocą obrotów, przesunięć, symetrii osiowej i jednokładności, aby były identyczne ze wzorem. Przykładowo: zbiór punktów $(0,0),(2,0),(2,1)$ jest podobny do zbioru $(6,1),(6,5),(4,5)$, ale nie do zbioru $(4,0),(6,0),(5,-1)$.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opisy: wzoru oraz zestawu badanych zbiorów punktów,
- wyznaczy, które z badanych zbiorów punktów są podobne do wzoru,
- wypisze wynik na standardowe wyjście.

Wejście

W pierwszym wierszu standardowego wejścia zapisana jest jedna liczba całkowita k ($1 \leqslant k \leqslant 25\,000$) — liczba punktów tworzących wzór. W kolejnych k wierszach zapisane są pary liczb całkowitych pooddzielanych pojedynczymi odstępami. W $i+1$-ym wierszu są współrzędne i-tego punktu należącego do wzoru: x_i i y_i ($-20\,000 \leqslant x_i, y_i \leqslant 20\,000$). Punkty tworzące wzór są (parami) różne. W kolejnym wierszu zapisana jest liczba zbiorów do zbadania n ($1 \leqslant n \leqslant 20$). Dalej następuje n opisów zbiorów punktów. Opis każdego zbioru rozpoczyna się od wiersza zawierającego jedną liczbę całkowitą l — liczbę punktów w danym zbiorze ($1 \leqslant l \leqslant 25\,000$). Punkty te są opisane w kolejnych wierszach, po jednym w wierszu. Opis punktu to dwie liczby całkowite oddzielone pojedynczym odstępem — jego współrzędne x i y ($-20\,000 \leqslant x, y \leqslant 20\,000$). Punkty tworzące jeden zbiór są parami różne.

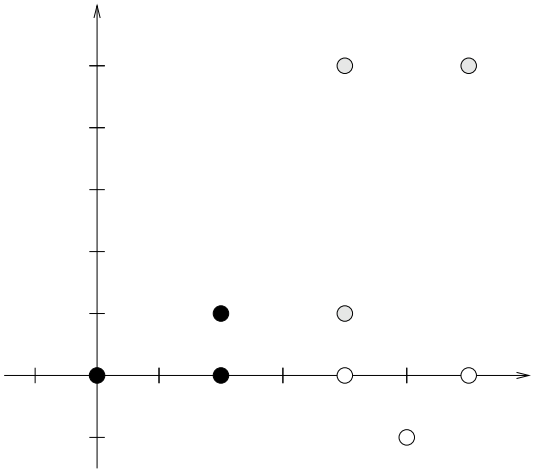
Wyjście

Twój program powinien wypisać na standardowe wyjście n wierszy — po jednym dla każdego badanego zbioru punktów. Wiersz nr i powinien zawierać słowo TAK, gdy i-ty z podanych zbiorów punktów jest podobny do podanego wzoru, lub słowo NIE w przeciwnym przypadku.

Przykład

Dla następującego wejścia:

```
3
0 0
2 0
2 1
2
3
4 1
6 5
4 5
3
4 0
6 0
5 −1
```



Poprawnym rozwiązaniem jest:

```
TAK
NIE
```

Rozdział 8

Algebra liniowa

Literatura
[NRP]

Przez Z_p, gdzie p jest liczbą pierwszą, oznaczać będziemy ciało liczb $\{0, 1, \ldots, p-1\}$ z operacjami realizowanymi modulo p. Przykładowo, w ciele Z_3 zachodzą następujące równości: $2 + 1 = 0$, $2 * 2 = 1$, natomiast w ciele Z_7 mamy $6 + 3 = 2$, $5 * 4 = 6$.

W rozdziale tym zajmiemy się zagadnieniem rozwiązywania układów równań liniowych w ciele Z_p. Problem ten polega na wyznaczeniu wartości zmiennych $x_0, x_1, .., x_{n-1}, x_n \in \{0, 1 \ldots, p-1\}$ takich, aby wszystkie zadane równania liniowe zmiennych ze zbioru X były spełnione. W kolejnych rozdziałach zostanie przedstawiony algorytm pozwalający na rozwiązywanie takich układów równań. Ciekawa sytuacja zachodzi w przypadku ciała Z_2, w którym zmienne przyjmują wartość 0 oraz 1. Istnieje wiele zadań, które można rozwiązać, przypisując występującym w tych zadaniach warunkom typu prawda-fałsz, zmienne z ciała Z_2, a następnie rozwiązując odpowiedni układ równań. Ze względu na dużą użyteczność rozwiązywania układów równań w ciele Z_2, przedstawimy implementację algorytmu, pozwalającego na efektywne rozwiązywanie tego typu układów równań.

Drugim problemem, jaki zostanie poruszony w tym rozdziale, jest zagadnienie programowania liniowego. Ogólnie polega ono na wyznaczeniu wartości zmiennych $x_0, x_1, .., x_{n-1} \in R^+$ takich, aby zmaksymalizować wartość funkcji liniowej $f(x_0, x_1, .., x_{n-1})$ oraz aby były spełnione zadane ograniczenia $g_1, g_2, \ldots, g_k$, będące nierównościami liniowymi zmiennych $x_0, x_1, .., x_{n-1}$. Przy użyciu programowania liniowego istnieje możliwość rozwiązywania takich problemów, jak wyznaczanie najcięższego/najlższejszego maksymalnego skojarzenia w grafie [PCC]. Algorytm rozwiązujący ten problem dla grafów dwudzielnych przy użyciu innych metod został przedstawiony w rozdziale poświęconym teorii grafów.

8.1. Eliminacja Gaussa

Literatura
[NRP] - 2.2

Eliminacja Gaussa jest jedną z najczęściej stosowanych metod rozwiązywania układów równań liniowych. Załóżmy, że mamy dany układ m równań z n niewiadomymi postaci:

$$\begin{cases} x_0 * a_{0,0} + x_1 * a_{0,1} + \ldots + x_{n-1} * a_{0,n-1} = b_0 \\ x_0 * a_{1,0} + x_1 * a_{1,1} + \ldots + x_{n-1} * a_{1,n-1} = b_1 \\ \ldots \\ x_0 * a_{m-1,0} + x_1 * a_{m-1,1} + \ldots + x_{n-1} * a_{m-1,n-1} = b_{m-1} \end{cases}$$

Taki układ równań można zapisać w postaci macierzowej $A * x = b$, gdzie:

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,n-1} \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix}$$

Metoda Gaussa wyznacza wartości zmiennych $x_0, x_1, .., x_{n-1}$, dla których wszystkie zadane równania liniowe są spełnione, bądź stwierdza, że podany układ równań jest sprzeczny.

Znalezienie rozwiązania układu równań metodą Gaussa polega na takim przekształceniu równania postaci: $A * x = b$ do postaci równoważnej $C * x = d$, że macierz C jest macierzą trójkątną górną:

$$\mathbf{C} = \begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,n-1} \\ 0 & c_{1,1} & \dots & c_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c_{m-1,n-1} \end{pmatrix}$$

Z takiej postaci układu równań (o ile jest niesprzeczny), w łatwy sposób można wyznaczyć wartości kolejnych niewiadomych: $x_{n-1}, x_{n-2}, \dots, x_0$. Z ostatniego równania można wyznaczyć $x_{n-1} = \frac{d_{m-1}}{c_{m-1,n-1}}$. Następnie wyznaczoną wartość można podstawić do kolejnego równania, które będzie zawierało tylko jedną niewiadomą x_{n-2} ...

W przypadku rozwiązywania układów równań w ciele liczb rzeczywistych, pojawia się dodatkowy problem dokładności wykonywanych obliczeń (wykonywane zaokrąglenia mogą się kumulować, wpływając znacząco na wartość obliczonego rozwiązania). W zadaniach konkursowych, układy równań, które wymagają rozwiązania, mają zazwyczaj specyficzną konstrukcję — wszystkie wartości niewiadomych oraz współczynników są liczbami całkowitymi z przedziału $\{0, 1, \dots, p-1\}$ dla p będącego liczbą pierwszą, a operacje arytmetyczne wykonuje się modulo p. Pozwala to zapomnieć o problemie zaokrągleń i skupić się tylko na operacjach całkowitoliczbowych. W kolejnych dwóch rozdziałach zostaną przedstawione implementacje eliminacji Gaussa odpowiednio w ciele Z_2 oraz Z_p. Oba przedstawiane algormytmy mają złożoność czasową $O(n^3)$.

8.1.1. Eliminacja Gaussa w Z_2

Literatura
[MD] - 3.6

Jako szczególny przypadek rozwiązywania układów równań w ciele Z_p, który często przydaje się w zadaniach, jest rozwiązywanie układów równań w Z_2. W ciele tym zmienne mogą przyjmować dwie możliwe wartości — 0 lub 1. Implementacja algorytmu rozwiązującego tego typu układ równań zrealizowana jest przez funkcję **template** <**const int** s> **int** GaussZ2(vector< bitset<s> >, VI, VI&), która została umieszczona na listingu 8.1. Funkcja ta przyjmuje jako parametry tablicę reprezentującą układ równań (macierz A zgodnie z oznaczeniami z poprzedniego rozdziału), wektor wartości b oraz wektor, w którym ma zostać umieszczony wynik (wektor x), o ile układ równań jest niesprzeczny. Funkcja zwraca wartość -1 w przypadku, gdy podany układ równań nie ma rozwiązania, 0 — w przypadku, gdy układ równań jest jednoznaczny, wartość większą od 0 — w przypadku, gdy układ równań nie jest jednoznaczny (zwrócona wartość jest wymiarem przestrzeni rozwiązań).

Listing 8.1: Implementacja funkcji **template** <**const int** s> **int** GaussZ2(vector< bitset<s> >, VI, VI&)

```
// Funkcja rozwiązuje układ równań w ciele Z2
01 template <const int s> int GaussZ2(vector < bitset<s> >equ, VI vals,
```

Listing 8.1: (c.d. listingu z poprzedniej strony)

```
02 VI & res) {
03    int n = SIZE(equ), a = 0, b = 0, c;
// ustaw odpowiednią wielkość wektora wynikowego
04    res.resize(s, 0);
// Dla kolejnej niewiadomej xb...
05    for (; a < n && b < s; b++) {
// Wyznacz pierwsze równanie spośród równań [a..n-1], w którym współczynnik
// przy zmiennej xb jest niezerowy
06       for (c = a; c < n && !equ[c][b]; c++);
// Jeśli znaleziono takie równanie...
07       if (c < n) {
// Jeśli w równaniu numer a współczynnik przy xb jest równy 0, to
// dodaj (modulo 2) równanie numer c do a
08          if (a != c) {
09             equ[a] ^= equ[c];
10             vals[a] ^= vals[c];
11          }
// Dla wszystkich równań różnych od a, wyeliminuj z nich współczynnik
// przy xb
12          REP(y, n) if (a != y && equ[y][b]) {
13             equ[y] ^= equ[a];
14             vals[y] ^= vals[a];
15          }
// Zapamiętaj numer równania, w którym współczynnik przy zmiennej xb jest
// różny od 0
16          res[b] = ++a;
17       }
18    }
// Dla każdej niewiadomej, jeśli istnieje dla niej równanie z niezerowym
// współczynnikiem, to wyznacz jej wartość z tego równania
19    REP(x, b) if (res[x]) res[x] = vals[res[x] - 1];
// Sprawdź, czy wyznaczone rozwiązanie spełnia układ równań
20    REP(x, n) {
21       c = 0;
22       REP(z, s) if (equ[x][z]) c ^= res[z];
23       if (c != vals[x]) return -1;
24    }
// Zwróć wymiar przestrzeni rozwiązań
25    return s - a;
26 }
```

Dla następującego układu równań:

$$\begin{cases} x_1 + x_2 = 0 * x_0 + 1 * x_1 + 1 * x_2 + 0 * x_3 + 0 * x_4 = 0 \\ x_1 + x_3 = 0 * x_0 + 1 * x_1 + 0 * x_2 + 1 * x_3 + 0 * x_4 = 1 \\ x_1 + x_3 + x_4 = 0 * x_0 + 1 * x_1 + 0 * x_2 + 1 * x_3 + 1 * x_4 = 0 \end{cases}$$

rozwiązanie jest przedstawione na listingu 8.2.

Listing 8.2: Przykład wykorzystania funkcji **template** <**const int** s> **int** GaussZ2(vector< bitset<s> >, VI, VI&)

```
Wymiar przestrzeni rozwiazan: -1
x0 = 0
x1 = 1
x2 = 1
x3 = 0
```

Listing 8.3: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 8.2. Pełny kod źródłowy programu znajduje się w pliku `gaussz2.cpp`

```
01 int main() {
// Skonstruuj rozwiązywany układ równań
02   vector < bitset < 4 > >equ(4);
03   VI vals(4), res;
04   equ[0][1] = equ[0][2] = 1;
05   vals[0] = 0;
06   equ[1][1] = equ[1][3] = 1;
07   vals[1] = 1;
08   equ[2][1] = equ[2][3] = equ[2][4] = 1;
09   vals[2] = 0;
// Wyznacz jego rozwiązanie
10   cout << "Wymiar przestrzeni rozwiazan: " << GaussZ2(equ, vals, res) << endl;
11   REP(z, SIZE(res)) cout << "x" << z << " = " << res[z] << endl;
12   return 0;
13 }
```

Zadanie: Taniec

Pochodzenie: Szwajcarska Olimpiada Informatyczna 2004

Rozwiązanie: `dance.cpp`

Ostatnio oglądałeś program telewizyjny, w którym piosenkarz tańczył na szachownicy, składającej się z kolorowych, podświetlanych od dołu pól. Każdy jego krok na polu powodował przełączenie podświetlenia; dodatkowo wszystkie sąsiadujące pola również zmieniały stan swojego podświetlenia. Twoim zadaniem jest sprawdzenie, czy istnieje możliwość zapalenia wszystkich świateł na szachownicy, wykonując w tym celu odpowiedni taniec.

Na początku tańca niektóre obszary są już zapalone. Wolno tańczyć po wszystkich polach szachownicy. Każdy krok na polu powoduje zamianę stanu aktualnego pola oraz czterech pól sąsiednich (w przypadku pól leżących na krawędzi szachownicy odpowiednio mniejszej ich liczby) .

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia początkowy stan szachownicy,
- stwierdzi, czy da się wykonać pożądany taniec i jeśli tak, to wypisze listę kroków, które należy wykonać,

- wypisze wynik na standardowe wyjście

Wejście

Pierwsza linia wejścia zawiera dwie liczby naturalne x i y ($3 \leqslant x, y \leqslant 15$), oznaczające odpowiednio szerokość i wysokość szachownicy. Kolejnych y wierszy zawiera po x znaków, opisujących stany kolejnych pól szachownicy. 0 oznacza, że światło pod odpowiednim polem jest zgaszone, podczas gdy 1 oznacza, że światło jest zapalone.

Wyjście

W pierwszym wierszu wyjścia powinna znaleźć się jedna liczba całkowita n — minimalna liczba kroków potrzebnych do włączenia wszystkich świateł na szachownicy. Kolejnych n wierszy powinno zawierać po dwie liczby całkowite i oraz j. Każda para liczb wyznacza pojedynczy krok, jaki tancerz musi wykonać — nastąpienie na pole w i-tym wierszu j-tej kolumny. Jeśli istnieje wiele rozwiązań, Twój program powinien wypisać dowolne z nich. Jeśli rozwiązanie nie istnieje, program powinien wypisać -1.

Przykład

Dla następującego wejścia:

```
4 3
0111
1010
1000
```

Poprawnym rozwiązaniem jest:

```
3
2 1
3 1
3 4
```

8.1.2. Eliminacja Gaussa w Z_p

Literatura
[MD] - 3.6

Po przeanalizowaniu problemu rozwiązywania układów równań w Z_2, zajmiemy się przypadkiem ciała Z_p, gdzie p jest liczbą pierwszą.

Sposób realizacji eliminacji Gaussa jest w tym przypadku podobny. Różnica polega na sposobie wykonywania operacji arytmetycznych. W przypadku ciała Z_2 wykonywanie operacji arytmetycznych jest proste. W ogólnym przypadku natomiast konieczne jest wyznaczanie odwrotności liczb (modulo p) (była o tym mowa w rozdziale poświęconym teorii liczb). Dokładny opis realizacji tego algorytmu można znaleźć w literaturze.

Przedstawiona na listingu 8.4 funkcja **int** `Gauss(vector<VI>&, VI&, VI&, int)` przyjmuje jako parametry odpowiednio tablicę reprezentującą układ równań A, wektor wartości b, wektor x oraz liczbę pierwszą p. Funkcja zwraca -1 w przypadku, gdy podany układ równań nie ma rozwiązań, w przeciwnym przypadku zwraca wymiar przestrzeni rozwiązań oraz przykładowy wynik umieszcza w wektorze x.

Listing 8.4: Implementacja funkcji **int** `Gauss(vector<VI>&, VI&, VI&, int)`

```
// Funkcja rozwiązuje dany układ równań w ciele Zp
01 int Gauss(vector<VI>& A, VI &b, VI &x, int P){
02    int m = SIZE(A), n = SIZE(A[0]), k, r;
03    VI q;
04    for (k = 0; k < min(m, n); k++) {
05       int i, j;
06       for (j = k; j < n; j++)
```

Listing 8.4: (c.d. listingu z poprzedniej strony)

```
07          for (i = k; i < m; i++)
08             if (A[i][j] != 0) goto found;
09       break;
10       found:
11       if (j != k) REP(t, m) swap(A[t][j], A[t][k]);
12       q.PB(j);
13       if (i != k) {
14          swap(A[i], A[k]);
15          swap(b[i], b[k]);
16       }
17       FOR(j, k + 1, m - 1) if (A[j][k] != 0) {
18          int l = (A[j][k] * RevMod(A[k][k], P)) % P;
19          FOR(i, k + 1, n - 1) A[j][i] = (P + A[j][i] - (l * A[k][i]) % P) % P;
20          b[j] = (P + b[j] - (l * b[k]) % P) % P;
21       }
22    }
23    r = k;
24    x.clear();
25    x.resize(n, 0);
26    FOR(k, r, m - 1) if (b[k] != 0) return -1;
27    FORD(k, r - 1, 0) {
28       int s = b[k];
29       FOR(j, k + 1, r - 1) s = (P + s - (A[k][j] * x[j]) % P) % P;
30       x[k] = (s * RevMod(A[k][k], P)) % P;
31    }
32    FORD(k, r - 1, 0) swap(x[k], x[q[k]]);
33    return n - r;
34 }
```

Dla następującego układu równań:

$$\begin{cases} 1 * x_0 + 3 * x_1 + 7 * x2 = 0 \\ 4 * x_0 + 0 * x_1 + 14 * x2 = 1 \\ 2 * x_0 + 6 * x_1 + 8 * x2 = 2 \end{cases}$$

rozwiązanie w ciele Z_{19} jest przedstawione na listingu 8.5.

Listing 8.5: Przykład wykorzystania funkcji **int** `Gauss(vector<VI>&, VI&, VI&, int)`

```
Wymiar przestrzeni rozwiazan: 0
x0 = 3
x1 = 4
x2 = 6
```

Listing 8.6: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 8.5. Pełny kod źródłowy programu znajduje się w pliku `gausszp.cpp`

```
01 int main() {
// Skonstruuj układ równań
02   vector<VI> a(3, VI(3));
03   VI b(3), x;
04   a[0][0] = 1, a[0][1] = 3, a[0][2] = 7;
05   b[0] = 0;
06   a[1][0] = 4, a[1][1] = 0, a[1][2] = 14;
07   b[1] = 1;
08   a[2][0] = 2, a[2][1] = 6, a[2][2] = 8;
09   b[2] = 2;
// I rozwiąż go
10   cout << "Wymiar przestrzeni rozwiazan: " << Gauss(a, b, x, 19) << endl;
11   REP(i, SIZE(x)) cout << "x" << i << " = " << x[i] << endl;
12   return 0;
13 }
```

Zadanie: Sejf

Pochodzenie: Potyczki Algorytmiczne 2006

Rozwiązanie: `safe.cpp`

Bajtosmok posiada sejf, do zawartości którego dostępu broni zamek składający się z n szyfratorów. Każdy szyfrator jest po prostu pokrętłem, które można ustawić w p różnych pozycjach. Sejf otwiera się, gdy wszystkie szyfratory zostają ustawione w odpowiedniej pozycji.

Bajtosmok dawno nie używał sejfu, przez co zapomniał konfiguracji, która go otwiera. Udał się do zakładu produkującego szyfratory, jednak jedyne czego się dowiedział, to schemat konstrukcji zamka. Zamek składa się z n szyfratorów i tylu samo blokad — wszystkie one, zarówno szyfratory jak i blokady, mogą być ustawione w jednej z p pozycji numerowanych od 0 do $p-1$. Zamek otwiera się w momencie, gdy wszystkie blokady ustawione są w pozycji 0. Przekręcenie i-tego szyfratora o jedną pozycję (z pozycji 0 na 1, z pozycji 1 na 2, ..., z $p-2$ na $p-1$, z $p-1$ na 0) powoduje, że j-ta blokada przekręca się o $c_{i,j}$ pozycji (z pozycji l na pozycję $(l+c_{i,j}) \pmod p$). W celu umożliwienia rozszyfrowania kombinacji otwierającej sejf, Bajtosmok otrzymał również nowoczesny skaner trójwymiarowy (model „Wzrok Supersmoka"), który umożliwia mu sprawdzenie w jakiej konfiguracji znajdują się ukryte wewnątrz mechanizmu sejfu blokady.

Jako że sejfy, których używa Bajtosmok są zawsze pierwszej klasy, można założyć, że zawsze istnieje dokładnie jedna kombinacja otwierająca sejf.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia opis konstrukcji zamku w sejfie,
- wyznaczy ustawienie szyfratorów, przy którym zamek jest otwarty,
- wypisze wynik na standardowe wyjście

Wejście

Pierwszy wiersz zawiera dwie liczby całkowite n — liczbę szyfratorów, $1 \leqslant n \leqslant 300$ oraz liczbę pierwszą p — liczbę pozycji, w których może znajdować się jeden szyfrator, $3 \leqslant p \leqslant 40\,000$ (liczba pierwsza to taka, która ma dokładnie dwa dzielniki: 1 i samą siebie). Następny wiersz zawiera n liczb całkowitych z zakresu od 0 do $p-1$ — pozycje, w których są ustawione kolejne szyfratory. Kolejny wiersz zawiera również n liczb całkowitych od 0 do $p-1$ — pozycje, w których ustawione są blokady w zamku Bajtosmoka. Następnych n wierszy zawiera opisy poszczególnych szyfratorów — i-ty z tych wierszy zawiera dokładnie n liczb całkowitych — kolejne liczby $c_{i,0}$, $c_{i,1}$, ..., $c_{i,n-1}$, $0 \leqslant c_{i,j} < p$. W dowolnym wierszu liczby pooddzielane są pojedynczymi odstępami.

Wyjście

W pierwszym i jedynym wierszu wyjścia twój program powinien wypisać n liczb całkowitych z przedziału 0 do $p-1$ oddzielonych pojedynczymi odstępami — pozycje kolejnych szyfratorów, dla których zamek jest otwarty.

Przykład

Dla następującego wejścia:

```
2  3
1  1
2  2
1  0
0  1
```

Poprawnym rozwiązaniem jest:

```
2  2
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10309 acm.sgu.ru - zadanie 260	acm.uva.es - zadanie 10524 acm.uva.es - zadanie 10109	acm.uva.es - zadanie 10808

8.2. Programowanie liniowe

Literatura
[NRP] - 2.2
[PCC]
[WDA] - 25.5

Zagadnienie programowania liniowego polega na wyznaczeniu nieujemnych liczb rzeczywistych $x_0, x_1, .., x_{n-1}$ maksymalizujących bądź minimalizujących wartość danej funkcji liniowej postaci

$$f(x_0, x_1, .., x_{n-1}) = b_0 * x_0 + b_1 * x_1 + ... + b_{n-1} * x_{n-1}$$

przy jednoczesnym zachowaniu wszystkich warunków ograniczających, będących postaci nierówności liniowych:

$$\begin{cases} x_0 * a_{0,0} + x_1 * a_{0,1} + \ldots + x_{n-1} * a_{0,n-1} \leqslant c_0 \\ x_0 * a_{1,0} + x_1 * a_{1,1} + \ldots + x_{n-1} * a_{1,n-1} \leqslant c_1 \\ \ldots \\ x_0 * a_{m-1,0} + x_1 * a_{m-1,1} + \ldots + x_{n-1} * a_{m-1,n-1} \leqslant c_{m-1} \end{cases}$$

W tym rozdziale skupimy się na problemie maksymalizacji wartości funkcji f — problem minimalizacji można w łatwy sposób rozwiązać poprzez zamianę znaków przy współczynnikach w funkcji f:

$$f(x_0, x_1, .., x_{n-1}) = -b_0 * x_0 - b_1 * x_1 - ... - b_{n-1} * x_{n-1}$$

W podobny sposób, można uzyskać warunki ograniczające postaci:

$$a_0 * x_0 + a_1 * x_1 + ... + a_{n-1} * x_{n-1} \geqslant c$$

poprzez zamianę znaków współczynników:

$$-a_0 * x_0 - a_1 * x_1 - ... - a_{n-1} * x_{n-1} \leqslant -c$$

Podobnie jak w przypadku eliminacji Gaussa, problem można reprezentować w postaci macierzowej. Dla danych macierzy A oraz wektora b i c, należy wyznaczyć wektor x liczb rzeczywistych dodatnich, spełniający nierówność $A * x \leqslant c$ oraz maksymalizujący wartość $b * x$.

Istnieje wiele różnych sposobów wyznaczania wartości wektora x, które zależą nie tylko od postaci warunków ograniczających, ale również od ich liczby czy ewentualnych zależności pomiędzy nimi występującymi. Więcej informacji na ten temat można uzyskać w literaturze. My natomiast skupimy się nad implementacją funkcji `vector<long double> simplex(vector<vector<long double> >&, vector<long double>&, vector<long double>&)`, której kod źródłowy znajduje się na listingu 8.7. Funkcja przyjmuje jako parametry macierz A, wektor c oraz wektor współczynników b dla maksymalizowanej funkcji. Jako wynik działania, funkcja ta zwraca wektor x, dla którego wartość funkcji f jest zmaksymalizowana, lub pusty wektor w przypadku, gdy podana lista warunków jest sprzeczna (może się tak również zdarzyć, gdy wartość maksymalizowanej funkcji nie jest ograniczona). Implementacja programowania liniowego została zamknięta w przestrzeni nazw (ang. namespace), zatem aby odwołać się do funkcji `simplex`, należy w programie dopisać **using namespace** `Simplex` lub wywoływać programowanie liniowe poprzez instrukcję `Simplex::simplex`. Po dokładną analizę rozwiązywania problemów programowania liniowego należy odwoływać się do literatury.

Listing 8.7: Implementacja funkcji `vector<long double> simplex(vector<vector<long double> >&, vector<long double>&, vector<long double>&)`

```
01 namespace Simplex {
// Typ wykorzystywany do wykonywania obliczeń - domyślnie jest to long double
02    typedef long double T;
03    typedef vector<T> VT;
04    vector<VT> A;
05    VT b, c, res;
06    VI kt, N;
07    int m;
08    inline void pivot(int k, int l, int e) {
09       int x = kt[l];
10       T p = A[l][e];
11       REP(i, k) A[l][i] /= p;
12       b[l] /= p;
13       N[e] = 0;
14       REP(i, m) if (i != l)
```

Listing 8.7: (c.d. listingu z poprzedniej strony)

```
15          b[i] -= A[i][e] * b[l], A[i][x] = A[i][e] * -A[l][x];
16        REP(j, k) if (N[j]) { c[j] -= c[e] * A[l][j];
17          REP(i, m) if (i != l) A[i][j] -= A[i][e] * A[l][j];
18        }
19        kt[l] = e;
20        N[x] = 1;
21        c[x] = c[e] * -A[l][x];
22      }
23      VT doit(int k) {
24        VT res;
25        T best;
26        while (1) {
27          int e = -1, l = -1;
28          REP(i, k) if (N[i] && c[i] > EPS) {
29            e = i;
30            break;
31          }
32          if (e == -1) break;
33          REP(i, m) if (A[i][e] > EPS && (l == -1 || best > b[i] / A[i][e]))
34            best = b[l = i] / A[i][e];
35          if (l == -1) return VT();
36          pivot(k, l, e);
37        }
38        res.resize(k, 0);
39        REP(i, m) res[kt[i]] = b[i];
40        return res;
41      }
42      VT simplex(vector<VT> &AA, VT & bb, VT & cc) {
43        int n = AA[0].size(), k;
44        m = AA.size();
45        k = n + m + 1;
46        kt.resize(m);
47        b = bb;
48        c = cc;
49        c.resize(n + m);
50        A = AA;
51        REP(i, m) {
52          A[i].resize(k);
53          A[i][n + i] = 1;
54          A[i][k - 1] = -1;
55          kt[i] = n + i;
56        }
57        N = VI(k, 1);
58        REP(i, m) N[kt[i]] = 0;
59        int pos = min_element(ALL(b)) - b.begin();
60        if (b[pos] < -EPS) {
61          c = VT(k, 0);
62          c[k - 1] = -1;
```

Listing 8.7: (c.d. listingu z poprzedniej strony)

```
63          pivot(k, pos, k - 1);
64          res = doit(k);
65          if (res[k - 1] > EPS) return VT();
66          REP(i, m) if (kt[i] == k - 1)
67             REP(j, k - 1) if (N[j] && (A[i][j] < -EPS || EPS < A[i][j])) {
68             pivot(k, i, j);
69             break;
70          }
71          c = cc;
72          c.resize(k, 0);
73          REP(i, m) REP(j, k) if (N[j]) c[j] -= c[kt[i]] * A[i][j];
74       }
75       res = doit(k - 1);
76       if (!res.empty()) res.resize(n);
77       return res;
78    }
79 };
```

Dla następujących ograniczeń:

$$\begin{cases} -0.5 * x_0 - 1 * x_1 + 2 * x_2 \leqslant -2 \\ x_0 + 2 * x_1 \leqslant 100 \end{cases}$$

oraz funkcji celu postaci $5 * x0 - 1.5 * x1 + 0.1 * x2$ wyznaczony przez programowanie liniowe wynik jest przedstawiony na listingu 8.8.

Listing 8.8: Przykład wykorzystania programowania liniowego

```
Najlepsze rozwiazanie : x0 = 100      x1 = 0      x2 = 24
Wartosc funkcji celu = 502.4
```

Listing 8.9: Kod źródłowy programu użytego do wyznaczenia wyniku z listingu 8.8. Pełny kod źródłowy programu znajduje się w pliku `linearprog.cpp`

```
01 typedef long double LD;
02 int main() {
// Skonstruuj odpowiednie macierze oraz wektory
03    vector < vector<LD> >A(2, vector<LD> (3));
04    vector<LD> b(2), c(3), res;
05    A[0][0] = -0.5, A[0][1] = -1, A[0][2] = 2, b[0] = -2;
06    A[1][0] = 1, A[1][1] = 2, A[1][2] = 0, b[1] = 100;
07    c[0] = 5, c[1] = -1.5, c[2] = 0.1;
// Rozwiąż programowanie liniowe oraz wypisz wynik
08    res = Simplex::simplex(A, b, c);
09    cout << "Najlepsze rozwiazanie : ";
10    REP(i, SIZE(res)) cout << "x" << i << " = " << res[i] << "\t";
11    cout << endl;
12    LD acc = 0;
```

Listing 8.9: (c.d. listingu z poprzedniej strony)

```
13    REP(i, SIZE(res)) acc += res[i] * c[i];
14    cout << "Wartosc funkcji celu = " << acc << endl;
15    return 0;
16 }
```

Zadanie: Szalony malarz

Pochodzenie: Problem Set Archive with Online Judge (http://acm.zju.edu.cn/)

Rozwiązanie: `paint.cpp`

Szalony malarz — Henry Daub — planuje namalować nowe arcydzieło. Jak wszystkie jego obrazy, nowe malowidło będzie prostokątem o wymiarach $m \times n$ cali, a każdy cal kwadratowy będzie pomalowany na pewien określony kolor. Henry chce zminimalizować czas potrzebny na namalowanie obrazu. Stworzył już szkic i teraz planuje, w jaki sposób malować obraz.

Istnieją trzy podstawowe techniki malowania, które Henry wykorzystuje w swoich obrazach. W każdym kroku może on namalować poziomą linię, pionową linię lub pojedyncze pole o rozmiarze cala kwadratowego. Malowanie poziomej linii koloruje pewną liczbę poziomo-sąsiadujących ze sobą pól na ten sam kolor, podobnie — malowanie pionowej linii koloruje pewną liczbę pionowo-sąsiadujących ze sobą pól na ten sam kolor. Malowanie pojedynczego pola koloruje tylko to pojedyncze pole. Malowanie poziomej linii zajmuje h sekund, a pionowa linia może być namalowana w czasie v sekund. Pojedyncze pole może zostać pomalowane w s sekund. Namalowany obraz musi dokładnie odpowiadać sporządzonemu wcześniej szkicowi. Co więcej, nie jest dozwolone zmienianie koloru pojedynczych pól — mogą one być malowane wielokrotnie, ale za każdym razem musi to być farba tego samego koloru. Pomóż Henremu wyznaczyć czas potrzebny na namalowanie obrazu. Na początku, całe płótno jest puste.

Zadanie

Napisz program, który:

- wczyta ze standardowego wejścia rozmiar obrazu oraz opis jego szkicu,
- wyznaczy czas oraz listę ruchów, jakie należy wykonać, aby pomalować obraz,
- wypisze wynik na standardowe wyjście.

Wejście

Pierwszy wiersz wejścia zawiera liczby m, n, h, v oraz s ($1 \leqslant m, n \leqslant 30, 1 \leqslant h, v, s \leqslant 105$). Następnych m wierszy zawiera po n liter, oznaczających kolory poszczególnych pól obrazu — kolory reprezentowane są przez małe litery alfabetu angielskiego.

Wyjście

W pierwszym wierszu program powinien wypisać dwie liczby całkowite t — czas potrzebny na pomalowanie obrazu i k — liczbę ruchów, jakie należy wykonać. Kolejnych k wierszy musi zawierać opisy ruchów. Pierwszym elementem opisu ruchu jest litera, oznaczająca rodzaj wykonywanego ruchu: 'h' (ruch poziomy), 'v' (ruch pionowy) lub 's' (pojedynczy punkt). W przypadku pierwszych dwóch ruchów, po literze następują cztery liczby — współrzędne górnego-lewego i dolnego-prawego końca malowanej kreski. W ostatnim przypadku — dwie liczby całkowite będące współrzędnymi malowanego pola. Ostatnim elementem opisu ruchu jest kolor użyty do pomalowania pól.

Przykład

Dla następującego wejścia:

```
4 4 3 5 2
aabc
aabc
bbbb
ccbc
```

Poprawnym rozwiązaniem jest:

```
23 8
s 1 4 c
s 2 4 c
s 4 4 c
h 1 1 1 2 a
h 2 1 2 2 a
h 3 1 3 4 b
h 4 1 4 2 c
v 1 3 4 3 b
```

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 10817	acm.uva.es - zadanie 10498	acm.sgu.ru - zadanie 248

Rozdział 9

Elementy strategii podczas zawodów

W każdym rzemiośle niezwykle ważną rolę odgrywa praktyka. Nawet najwyższe teoretyczne wykształcenie jej nie zastąpi. Oczywiście bez wiedzy doświadczenie również nie ma swojej wartości. Oba te czynniki muszą ze sobą współgrać w odpowiednich proporcjach. Podobna sytuacja odnosi się do udziału w konkursach programistycznych. Bez znajomości algorytmiki nie ma się szans na poprawne i efektywne rozwiązywanie zadań. Jednak znajomość optymalnego rozwiązania nie poparta treningiem, nie zapewnia właściwego zaimplementowania zadania podczas konkursu. Uczestniczenie w wielu treningach przynosi „wiedzę praktyczną", o której nie można przeczytać w żadnej książce.

W niniejszym rozdziale poruszone zostały problemy, związane ze strategią brania udziału w konkursach. Wiedza ta nie pochodzi z teoretycznych rozważań, dotyczących sposobów przeprowadzania zawodów, lecz stanowi konstruktywne wnioski wyciągnięte z udziału w wielu konkursach — wnioski, które nie tylko pomogą w efektywnym rozwiązywaniu zadań, ale pozwolą również na uniknięcie wielu problemów, z których istnienia można sobie wcześniej nie zdawać sprawy, a które w istotny sposób wpływają na szybkość osiągania celu. W dziale Dodatki znajduje się niezwykle cenny materiał — zbiór dobrych rad pochodzących od wysokiej klasy zawodników. Stanowi on uzupełnienie zawartości niniejszego rozdziału.

9.1. Szacowanie oczekiwanej złożoności czasowej

Rozwiązywanie każdego zadania składa się z trzech faz: wymyślenia algorytmu, zaimplementowania rozwiązania oraz przetestowania go, połączonego zazwyczaj z dokonaniem odpowiednich poprawek. Czym mniej czasu spędza się sumarycznie podczas tych trzech faz, tym lepsze osiąga się efekty. Często początkujący zawodnicy, zdając sobie sprawę z tej zależności, zabierają się za rozwiązywanie zadań w sposób zachłanny — chcą jak najszybciej zakończyć każdą z tych trzech faz. Takie podejście doprowadza do sytuacji, w których podczas implementacji programu okazuje się, że należy uwzględnić jakieś dodatkowe przypadki skrajne, albo co gorsza, że zastosowany algorytm nie jest poprawny. Konsekwencją takich sytuacji jest istotne wydłużenie czasu spędzonego w drugiej i trzeciej fazie.

Pomijając aspekty poprawności samego rozwiązania, w wielu przypadkach okazuje się, że zadanie można zaimplementować na kilka sposobów — różnice często leżą w szczegółach. Niekiedy odpowiednie zainicjowanie zmiennych może pozwolić, w kolejnych fazach algorytmu, na pominięcie sprawdzania wielu skomplikowanych warunków. Umiejętność dostrzegania możliwych uproszczeń wymaga dużo praktyki, ale nawet najbardziej doświadczony zawodnik

potrzebuje czasu na ich wymyślenie. Poświęcenie kilku dodatkowych minut w fazie projektowania algorytmu, starając się dokładnie przemyśleć sposób implementacji oraz możliwe jej uproszczenia, zawsze zwrócą się w dwójnasób podczas dwóch kolejnych faz.

Istotny wpływ na złożoność implementowanych programów mają stosowane struktury danych. Rozwiązujący zadanie musi zdecydować, czy zastosować efektywniejszą i bardziej skomplikowaną strukturę danych, czy też prostszą i wolniejszą. W zależności od charakteru konkursu, w którym bierze się udział, w wielu przypadkach organizatorzy nie wymagają od zawodników implementacji asymptotycznie optymalnych rozwiązań. W takich przypadkach, umiejętność szacowania oczekiwanej złożoności jest bardzo istotna. Decyzja polegająca na wybraniu bardziej skomplikowanej struktury danych czy algorytmu, gwarantuje zmieszczenie się w limitach czasowych, ale zwiększa czas implementacji oraz ryzyko pomyłek. Wprawdzie nieefektywne podejście pozwala na uproszczenie implementacji, ale z drugiej strony naraża na przekroczenie limitów czasowych. Późniejsze gruntowne modyfikacje zaimplementowanego rozwiązania bywają bardzo trudne, zatem należy wystrzegać się sytuacji, które mogą prowadzić do konieczności zmiany podejścia do rozwiązywanego problemu.

Szacowanie oczekiwanej złożoności czasowej rozwiązania jest możliwe dzięki analizie wielkości limitów na dane wejściowe. Jeśli nie ma się w tym zakresie wprawy, najlepszym sposobem jest przeanalizowanie zadań konkursowych z lat ubiegłych wraz z sugerowanymi rozwiązaniami. Na tej podstawie można oszacować, w jaki sposób będą dobierane limity w zależności od złożoności rozwiązań dla zadań w przyszłości. Podczas dokonywania oszacowań, należy jednak pamiętać o nieustannie zwiększającej się szybkości komputerów, co pozwala organizatorom konkursu na nieznaczne zwiększanie limitów na maksymalną wielkość danych przy tych samych oczekiwanych asymptotycznych złożonościach rozwiązań.

Wyznaczenie oczekiwanej złożoności rozwiązania pozwala nie tylko na dobór odpowiednich struktur danych czy algorytmów, ale również daje pewną informację na temat metody rozwiązywania samego zadania. W ten sposób, bez czytania treści zadania, można bez problemu stwierdzić, że zadanie należy do klasy problemów NP-trudnych, jego rozwiązanie można oprzeć o programowanie dynamiczne lub że wymagany jest algorytm liniowy.

Zazwyczaj rozróżnia się cztery podstawowe klasy złożoności algorytmów:

- Programy o złożoności $O(n)$ — $O(n * log(n))$. Limity na dane wejściowe (oczywiście zmienia się to z czasem, o czym trzeba pamiętać) są rzędu $100\,000 - 1\,000\,000$. Istotny jest również fakt, iż jury nie jest zazwyczaj w stanie rozróżnić rozwiązań o złożoności $O(n)$ od $O(n * log(n))$, zatem mając wybór pomiędzy takimi dwoma algorytmami, bez obawy można przyjąć wolniejsze rozwiązanie (o ile oczywiście jest prostsze w implementacji). Czynnik logarytmiczny w złożoności pojawia się zazwyczaj ze względu na konieczność posortowania danych wejściowych.

- Programy o złożoności $O(n^2)$, ewentualnie z dodatkowymi czynnikami logarytmicznymi, mają limity nie przekraczające $10\,000 - 20\,000$. W tej klasie programów mieszczą się różne rozwiązania zachłanne oraz proste zadania na struktury danych, które często daje się również rozwiązać w złożoności $O(n*log(n))$, przy użyciu skomplikowanych struktur danych.

- Programy o złożoności $O(n^3)$ (z ewentualnymi czynnikami logarytmicznymi). W tej klasie rozwiązań znajdują się zazwyczaj algorytmy oparte o programowanie dynamiczne; zatem wielkość danych wejściowych zależy nie tylko od złożoności czasowej, ale również i pamięciowej, która często jest rzędu $O(n^2)$. Przy aktualnych limitach pamięciowych — 32-64 MB, ograniczenia na dane wejściowe nie przekraczają 1 000. Należy zauważyć,

iż rozwiązania sześcienne wielu zadań z tej klasy, mają małą stałą multiplikatywną, co pozwala na dobór stosunkowo dużych limitów na dane wejściowe.

- Rozwiązania wykładnicze, których złożoność czasowa może mieć we wzorze silnie, funkcje wykładnicze i dwumiany (przykładowo $O(n!)$, $O(2^n)$, $O(\binom{n}{k})$). W przypadku tego typu zadań, podejście polega zazwyczaj na generowaniu różnych rozwiązań postawionego problemu oraz sprawdzaniu, które z nich jest optymalne. W tego rodzaju zadaniach można, przy użyciu heurystyk, ograniczać przeszukiwaną przestrzeń rozwiązań, co w istotny sposób przyspiesza działanie programu. Ograniczenia przyjęte w tego typu zadaniach nie przekraczają wartości 50, a zazwyczaj oscylują w zakresie $10 - 25$.

9.2. Strategia pracy w drużynie

W przypadku brania udziału w zawodach drużynowych, takich jak ACM ICPC, bardzo ważne jest opracowanie strategii, zgodnie z którą muszą postępować wszyscy członkowie drużyny. Potrzeba dobrej organizacji nie wynika jedynie z konieczności współpracy między członkami zespołu, ale przede wszystkim z tego, że do dyspozycji trzech zawodników jest tylko jeden komputer i jego efektywne wykorzystanie jest nieodzowne. Dobra drużyna powinna przeprowadzać razem liczne treningi, mające na celu symulowanie zawodów. Umiejętność rozwiązywania zadań można ćwiczyć indywidualnie i nie są do tego konieczne treningi zespołowe. Po każdym treningu powinno nastąpić jego omówienie. Omówienie to jest tak samo istotne jak sam trening, gdyż ma ono na celu wyciągnięcie wniosków na przyszłość i usprawnienie współpracy w drużynie. Ważnymi elementami, na jakie należy zwracać uwagę podczas omówień, są:

- Czy któryś z zawodników stracił dużo czasu z określonego powodu, co w przyszłości można wyeliminować albo przynajmniej ograniczyć. Krytycznym elementem, z którym zawsze wiążą się problemy, jest oczekiwanie na komputer. Jednak i w tym zakresie istnieje możliwość wprowadzenia pewnych usprawnień, polegających na choćby zapisywaniu programów najpierw na kartce, a dopiero potem na komputerze. Odstępstwem od tej reguły są pierwsze oraz ostatnie zadania rozwiązywane podczas zawodów. Innym czynnikiem, który może powodować niepotrzebną stratę czasu podczas prawdziwych zawodów, jest oczekiwanie na dostarczenie przez organizatorów wydruku programu (poszukiwanie błędów w kodzie odbywa się bowiem na kartce). Dobrym pomysłem jest drukowanie każdego programu podczas wysyłania go do oceny (jeszcze przed uzyskaniem odpowiedzi). Jeśli okaże się, że rozwiązanie jest nieprawidłowe, to wydruk programu zostanie dostarczony o kilka cennych minut wcześniej i będzie można szybciej przystąpić do poszukiwania błędów.

- Czy poprawianie błędów w którymś z zadań zajęło bardzo dużo czasu. Sytuacje takie mogą być spowodowane czterema czynnikami. Pierwszym z nich jest losowy błąd w implementacji. Błędów takich trudno jest się wystrzec, nie ma też standardowego mechanizmu pozwalającego na ich wychwytywanie. W tym zakresie tylko częste treningi są w stanie pomóc.

  Drugim, bardzo niebezpiecznym problemem, jest zastosowanie nieprawidłowego algorytmu. Sytuacja taka może pojawić się po złym zrozumieniu treści zadania. Zdarza się, że pomimo przekonania o poprawności rozwiązania (co wymusza poszukiwanie błędu w samym programie), idea algorytmu okazać się może nieprawidłowa, a wówczas całą pracę należy rozpocząć od początku. Co więcej, sytuacje takie wprowadzają napięcie w zespole i dekoncentrują całą drużynę.

Trzecim problemem, który można ominąć, jest implementowanie programu przez nieodpowiednią osobę. Niektórzy zawodnicy preferują zadania z geometrii obliczeniowej, inni z teorii grafów. Ogólną zasadą podczas zawodów jest, aby zawodnicy nie wymieniali się zadaniami, o ile nie jest to konieczne. W przypadku, gdy zawodnik zdaje sobie jednak sprawę, że jego kolega z drużyny jest znacznie lepszy w rozwiązywaniu zadań określonego typu, należy zdecydować się na zamianę zadaniami.

Ostatnim powodem mogą być często powtarzające się błędy. Pojawiają się one zazwyczaj podczas zawodów, kiedy pracuje się pod presją. Przykładowo, może zdarzyć się, że zgłosi się do oceny program do innego zadania. Istnieje też wiele problemów natury implementacyjnej. Jednym z nich jest wykorzystywanie zmiennych globalnych oraz lokalnych o tej samej nazwie. Wydawać by się mogło, że wykrycie takiego błędu jest proste (debugując program na komputerze, bardzo szybko można to wykryć), lecz podczas zawodów, gdy programy poprawia się na kartce, jest to jedna z najtrudniejszych do wychwycenia usterek. Zazwyczaj szuka się błędów we wzorach i skomplikowanych ciągach instrukcji, a nie zwraca się uwagi na to, że dwie zmienne w programie mają taką samą nazwę. Bardzo dobrym pomysłem jest sporządzenie, na podstawie doświadczeń z treningów, listy powtarzających się błędów i za każdym razem odwoływanie się do niej podczas debugowania programu.

- Krytycznym momentem jest ostatnia godzina zawodów. Jeśli w drużynie każdy zawodnik przez cały czas pracuje nad swoimi zadaniami, to może okazać się pod koniec konkursu, że drużyna ma kilka prawie rozwiązanych zadań, ale nie jest możliwe zaimplementowanie w krótkim czasie żadnego z nich. Dobrym pomysłem jest, około godziny przed zakończeniem konkursu, przerwanie na chwilę rozwiązywania zadań i przedyskutowanie, jak wygląda sytuacja. Należy podjąć decyzję, nad którymi zadaniami będzie się dalej pracować i kto je powinien implementować. Nawet jeśli wydaje się, że istnieje szansa rozwiązania wszystkich rozpoczętych zadań, to należy dokładnie przemyśleć taką decyzję, gdyż praktyka pokazuje, że tego typu szarże zazwyczaj kończą się porażką.

- Ważną strategiczną decyzją jest również sposób korzystania z rankingu. Przez pierwsze godziny zawodów jest on dostępny publicznie, zatem można z niego czerpać bardzo pożyteczne informacje, takie jak ocena trudności zadań. Na podstawie rankingu można zadecydować, w jakiej kolejności należy rozwiązywać zadania. Bardzo ważna jest analiza, które zadania nie zostały jeszcze rozwiązane (może się to okazać o tyle istotne, iż istnieją zadania, które na pozór wydają się proste, ale kryją w sobie różnego rodzaju pułapki). W przypadku podjęcia próby rozwiązywania takiego zadania należy być szczególnie ostrożnym. Obserwowanie rankingu należy robić mądrze; jego zbyt częste wykorzystywanie może doprowadzić do sytuacji, w której zamiast rozwiązywać zadania, analizuje się pozycje poszczególnych drużyn. Ważne jest również to, że w sytuacjach ekstremalnych (choćby w sytuacjach gdy ma się szansę na czołową pozycję w konkursie), analizowanie rankingu wcale nie mobilizuje drużyny, lecz ją niepotrzebnie stresuje i zmniejsza efektywność pracy. Jednym z możliwych strategicznych podejść jest wyznaczenie jednego zawodnika o „najsilniejszych nerwach", odpowiedzialnego za sprawdzanie stanu rankingu i zasugerowanie drużynie, jakie zadania należy rozwiązywać w następnej kolejności.

Każdy zespół powinien opracować swoją własną strategię, dopasowaną do charakteru poszczególnych jego członków. W przypadku wielu czołowych drużyn strategie są stosunkowo podobne. Różnią się wprawdzie drobnymi szczegółami, ale zasadniczo przedstawiają się następująco:

Na samym początku zawodów zawodnicy rozdzielają między sobą zadania. Sposób podziału zadań powinien zostać ustalony przed zawodami. Istnieje możliwość podziału zadań w zależności od ich typu i preferencji zawodników, ale podejście takie wymaga wstępnej analizy treści zadań oraz wprowadza niepotrzebną dyskusję między członkami drużyny. W praktyce stosowane są inne metody rozdziału zadań.

Jedna z metod polega na rozdzieleniu zadań zgodnie z zasadą „modulo 3". Pierwszy zawodnik w drużynie otrzymuje zadania, których numery dzielą się przez 3 — $3, 6 \ldots$, drugi zawodnik dostaje zadania $1, 4 \ldots$, a trzeci $2, 5 \ldots$ Podejście takie jest o tyle wygodne, że gwarantuje równomierny rozkład liczby zadań oraz nie wymaga dodatkowego zastanawiania się (choćby sprawdzania, ile jest w sumie wszystkich zadań). Niestety, metoda ta ma też poważną wadę. Organizatorzy zawodów rzadko umieszczają dwa trudne zadania obok siebie (to samo tyczy się zadań łatwych). Przy podziale zadań zgodnym z zasadą „modulo 3", bardzo często zdarza się, że poziom trudności zadań, które dostali poszczególni zawodnicy, jest różny. Podczas wielu treningów przekonaliśmy się, że przy stosowaniu tej metody często jeden z członków drużyny dostawał najtrudniejsze zadania, co powodowało, że podczas gdy dwaj koledzy rozwiązali już po jednym zadaniu i zabierali się za kolejne, to „pechowiec" cały czas zastanawiał się nad rozwiązaniem swojego pierwszego zadania. Sytuacje takie nie tylko negatywnie wpływają na atmosferę w drużynie, ale również pogarszają pozycję w rankingu. Inną metodą podziału zadań jest przydzielenie pierwszych kilku zadań pierwszemu zawodnikowi, zadań środkowych drugiemu oraz reszty zadań trzeciemu.

Bardzo ważnym momentem podczas konkursu jest jego początek. Jeśli drużyna ma wygrać zawody na czasie, to musi mieć dobry start i w ciągu pierwszej godziny konkursu rozwiązać 3 - 4 zadania. Aby było to możliwe, rozwiązywanie zadań należy rozpocząć od najprostszych. Każdy z zawodników powinien zapoznać się pobieżnie ze wszystkimi swoimi zadaniami oraz zdecydować, które z nich jest najłatwiejsze. Nie należy zaczynać któregoś zadania (nie czytając reszty) tylko dlatego, że wie się od razu jak je rozwiązać. Często, jak na złość, okazuje się, że zadanie, które odłożyło się na koniec, jest najprostsze. Ważną zasadą podczas zawodów bowiem jest, że należy rozwiązywać zadania w kolejności od najprostszych do najtrudniejszych.

Na samym początku konkursu trzeba również rozwiązać problem wykorzystywanych nagłówków, pliku `Makefile` oraz innych używanych skryptów. Wcześniej czy później będzie je trzeba przepisać na komputer, a ponieważ na samym początku zawodów i tak nikt nie implementuje żadnego zadania (najpierw trzeba zapoznać się z ich treściami), więc najlepiej jest wybrać jednego członka drużyny, odpowiedzialnego za przygotowanie odpowiedniego środowiska pracy, podczas gdy reszta drużyny analizuje zadania.

Ze względu na konieczność efektywnego wykorzystania komputera, ważne jest pisanie programów na kartce. Takie podejście pozwala na skrócenie czasu korzystania z komputera. Rozpoczynając pisanie programu na kartce trzeba mieć pewność co do poprawności stosowanego algorytmu. Nie można sobie również pozwolić na zastanawianie się przy komputerze. Przepisanie programu z kartki zajmuje znacznie mniej czasu oraz pozwala na dodatkową weryfikację jego poprawności. Odstępstwem od tej reguły jest sam początek zawodów. W każdym zestawie zadań znajduje się proste zadanie, które można zaimplementować bez zastanowienia. W takich sytuacjach, należy pominąć implementację na kartce.

Kilka kolejnych godzin zawodów mija w podobny sposób — zawodnicy na zmianę piszą kolejne zadania na komputerze, drukują programy, szukają i poprawiają błędy.

W końcu dochodzi do jednego z dwóch możliwych scenariuszy.

Jeden z nich opisany został już wcześniej (każdy z zawodników kontynuuje rozwiązywanie swoich zadań, lecz czas, który pozostał do końca zawodów nie pozwoli na rozwiązanie wszystkich rozpoczętych).

Drugi scenariusz polega na tym, że zostało jeszcze kilka zadań, ale nie wiadomo, jak je rozwiązać. W takich sytuacjach przydaje się doświadczenie z treningów. W zależności od typu zadań, należy odpowiednio powymieniać je między sobą, lub podjąć decyzję o odrzuceniu określonych zadań i skoncentrowaniu się w grupie nad jednym najprostszym. W sytuacjach, w których nie ma się żadnych pomysłów na rozwiązanie, pomocne może okazać się przejście po korytarzu i oderwanie się na chwilę od konkursu. Po powrocie do stanowiska świeże spojrzenie na problem może przynieść dobry rezultat.

Czasem zdażają się też sytuacje w których zawodnik jest przekonany o poprawności zastosowanego algorytmu, jednak zaimplementowane rozwiązanie nie działa poprawnie. Jeśli długotrwałe poszukiwanie błędów w programie kończy się niepowodzeniem, bardzo często pomocne jest przedstawienie koledze z drużyny sposobu działania programu. Najlepiej jest przeanalizować z nim wszystkie linijki programu. Nawet jeśli kolega w ogóle nie rozumie co opisywany program robi, to często znacznie łatwiej jest zauważyć drobne błędy, opowiadając działanie programu innej osobie.

9.3. Szablon

Jak już wspomnieliśmy na początku książki, implementowane podczas zawodów programy korzystają ze zbioru nagłówków. Są one wspólne dla wszystkich programów, zatem ich przepisanie na komputer jest wymagane tylko przed rozpoczęciem implementowania pierwszego zadania. Przystępując do rozwiązywania kolejnego zadania, wystarczy skopiować poprzedni program i po usunięciu niepotrzebnych fragmentów kodu, wykorzystać go jako nowy szablon. Rozwiązanie takie jednak nie jest najbezpieczniejsze. Zdarzało się niekiedy, że osoba rozpoczynająca implementowanie kolejnego programu, myliła literkę rozwiązywanego przez siebie zadania, nadpisując tym samym rozwiązanie innego, zaimplementowanego już zadania. Nie stanowi to wielkiego problemu, jeśli nadpisane zadanie zostało już zaakceptowane. Ale jeśli kolega z drużyny właśnie szuka błędu w tym programie na kartce? Oznaczało by to konieczność przepisania całego programu od nowa.

Ominięcie tego typu problemów jest możliwe poprzez zastosowanie trochę innej metodologii. Przygotowany na samym początku zawodów szablon można skopiować wielokrotnie, tworząc w ten sposób pliki o identycznej zawartości: `a.cpp`, `b.cpp`, ... Pliki te stanowią szablony, na podstawie których powstają rozwiązania odpowiednich zadań. Ponieważ każdy program pisany podczas zawodów w języku C++ musi zawierać funkcję `main`, zwracającą wartość 0, można ją również umieścić w szablonie. Skopiowanie szablonów dla poszczególnych zadań można w szybki i prosty sposób wykonać pod platformą Linux przy użyciu komendy:

```
for i in a b c ..; do cp szkielet.cpp $i.cpp; done
```

gdzie kolejne litery $a, b, c, \ldots$ reprezentują symbole poszczególnych zadań.

9.4. Makefile

Jak pokaże zawartość kolejnego rozdziału, dobrym pomysłem jest kompilowanie implementowanych programów z zestawem pewnych parametrów. Aby uniknąć konieczności wpisywania ich za każdym razem, można stworzyć plik `Makefile`, którego używać będą wszyscy członkowie zespołu.

Rozwiązanie takie pozwala nie tylko zaoszczędzić czas, ale również wprowadza porządek w środowisku pracy. Przykład pliku `Makefile` został przedstawiony na listingu 9.1.

Listing 9.1: Przykładowy plik `Makefile`

```
%: %.cpp
	g++ -o $@ $< -g -W -Wall -Wshadow
```

Kompilacja zadania z wykorzystaniem pliku `Makefile` jest prosta, wystarczy wykonać polecenie `make zadanie`, gdzie `zadanie` reprezentuje nazwę kompilowanego programu.

9.5. Parametry kompilacji programów

Bardzo pożytecznym pomocnikiem podczas zawodów okazuje się kompilator. Trzeba tylko umieć z niego odpowiednio korzystać. Kompilator języka C++ dostępny podczas praktycznie każdych zawodów — GCC — udostępnia wiele parametrów kompilacji, pozwalających na automatyczne wykrywanie niebezpiecznych instrukcji w programie, które mogą prowadzić do powstawania błędów. Inne kompilatory również posiadają podobne opcje, których nazwy nie muszą się niestety pokrywać z tymi z kompilatora GCC, w przypadku korzystania z takich kompilatorów, najlepiej zapoznać się z ich dokumentacją. Jednym z najczęściej stosowanych przełączników jest *-Wall*. Jego nazwa sugeruje, że włącza on wszystkie udostępniane przez kompilator ostrzeżenia, jednak nie jest to prawdą. Istnieje wiele przełączników, które służą do weryfikowania zgodności programu z różnymi standardami, jak np. *-Wabi* generujący ostrzeżenia o możliwej niezgodności kodu z *C++ ABI* (Application Binary Interface). W kolejnych kilku podrozdziałach przedstawione zostały różne najbardziej pożyteczne z punktu widzenia zawodów opcje kompilatora.

9.5.1. *-Weffc++*

Parametr *-Weffc++* włącza ostrzeżenia o wystąpieniach w kodzie niezgodnościach ze stylem pisania programów określonym przez Scotta Meyers'a w książce „Effective C++" [ECP]. W skład tych błędów wchodzą głównie nieprawidłowe typy zwracane przez funkcje i operatory.

Najciekawszym — z punktu widzenia konkursów — jest sprawdzanie, czy struktury zawierające zmienne wskaźnikowe, posiadają zaimplementowany konstruktor kopiujący. Brak takiego konstruktora powoduje stworzenie konstruktora domyślnego, który kopiuje wartość wskaźników występujących w strukturze, nie alokując na ich potrzeby nowej pamięci. W ten sposób, obiekty współdzielą część pamięci, co w wielu przypadkach jest błędem implementacyjnym. Przykład niepoprawnego programu został przedstawiony na listingu 9.2.

Listing 9.2: Nieprawidłowa implementacja struktury ze zmienną wskaźnikową, która nie posiada konstruktora kopiującego.

```
01 #include <iostream>
02 using namespace std;
03 struct BigNum {
04    int *vec;
05    BigNum() : vec(new int[1]) { }
06    ~BigNum() {
07       cout << "Zwalnianie pamięci pod adresem " << vec << endl;
08       delete[]vec;
09    }
10 };
```

Listing 9.2: (c.d. listingu z poprzedniej strony)

```
11 int main() {
12    BigNum a;
13    BigNum b = a;
14    BigNum c(a);
15    return 0;
16 }
```

Program przedstawiony na listingu 9.2 nie jest prawidłowy, gdyż podczas wykonywania operacji z linii 13 oraz 14, tworzone są nowe zmienne `b` i `c`, zawierające wskaźniki `vec`, wskazujące na ten sam obszar pamięci, co wskaźnik `vec` zmiennej `a`. Podczas wywoływania destruktora obiektów, pamięć wskazywana przez wskaźniki `vec` jest zwalniana wielokrotnie. Wynik wykonania tego programu został przedstawiony na listingu 9.3. Kompilacja programu bez dodatkowych parametrów nie wygeneruje żadnych ostrzeżeń i zaistniały w programie problem może zostać niezauważony. Wykorzystanie opcji *-Weffc++* powoduje, że proces kompilacji wyświetli ostrzeżenia przedstawione na listingu 9.4.

Listing 9.3: Wynik działania programu z listingu 9.2

```
Zwalnianie pamięci pod adresem 0x3d3b90
Zwalnianie pamięci pod adresem 0x3d3b90
Zwalnianie pamięci pod adresem 0x3d3b90
```

Listing 9.4: Ostrzeżenia wygenerowane podczas kompilacji z włączoną opcją *-Weffc++* programu z listingu 9.2

```
progs/effc0.cpp:3: warning: 'struct BigNum' has pointer data members
progs/effc0.cpp:3: warning:    but does not override 'BigNum(const BigNum&)'
progs/effc0.cpp:3: warning:    or 'operator=(const BigNum&)'
```

Ostrzeżenia wygenerowane przez kompilator podczas kompilacji programu z listingu 9.2 pozwalają na natychmiastowe poprawienie błędów. Nowa wersja programu została przedstawiona na listingu 9.5. Dla tego programu, podczas kompilacji nie generowane są już żadne ostrzeżenia, a wynik działania został zaprezentowany na listingu 9.6.

Listing 9.5: Poprawiona wersja programu z listingu 9.2

```
01 #include <iostream>
02 using namespace std;
03 struct BigNum {
04    int *vec;
05    BigNum() : vec(new int[1]) { }
06    BigNum(const BigNum & a) : vec(new int[1]) { }
07    BigNum & operator=(const BigNum & a) {
08       vec = new int[1];
09       vec[0] = a.vec[0];
10       return *this;
11    }
```

Listing 9.5: (c.d. listingu z poprzedniej strony)

```
12    ~BigNum() {
13       cout << "Zwalnianie pamięci pod adresem " << vec << endl;
14    }
15 };
16 int main() {
17    BigNum a;
18    BigNum b = a;
19    BigNum c(a);
20    return 0;
21 }
```

Listing 9.6: Wynik działania programu z listingu 9.5

```
Zwalnianie pamięci pod adresem 0x3d3c00
Zwalnianie pamięci pod adresem 0x3d3bf0
Zwalnianie pamięci pod adresem 0x3d3b90
```

9.5.2. *-Wformat*

Operator *-Wformat* włącza weryfikację parametrów przekazywanych do funkcji, takich jak `printf` czy `scanf`. Na podstawie pierwszego argumentu tych funkcji, sprawdzane są nie tylko prawidłowe typy innych parametrów, ale również weryfikowana jest poprawność wykonywanych konwersji. Użycie opcji *-Wformat* automatycznie włącza również flagę *-Wnonnull*.

Na listingu 9.7 przedstawiony jest program, którego kompilacja bez opcji *-Wformat* nie generuje żadnych ostrzeżeń. Jednak program ten jest daleki od ideału, ze względu na nieprawidłowe typy zmiennych, stanowiących parametry funkcji `scanf` oraz `printf`.

W przypadku, gdy kompilacja jest wykonana z flagą *-Wformat*, wykrywane są zaistniałe niezgodności typów. Ostrzeżenia wygenerowane przez kompilator zostały przedstawione na listingu 9.8. Po dokonaniu poprawek (nowa wersja programu została umieszczona na listingu 9.9), kompilator nie generuje już żadnych ostrzeżeń.

Listing 9.7: Program zawierający nieprawidłowe wywołania funkcji `scanf` i `printf`

```
01 #include <stdio.h>
02 int main() {
03    int a;
04    char b[100];
05    long long c;
06    scanf("%d\n", a);
07    scanf("%s\n", &b);
08    printf("%d\n", c);
09    printf("%c\n", b);
10    return 0;
11 }
```

Listing 9.8: Ostrzeżenia generowane przez kompilator przy włączonej opcji *-Wformat* dla programu z listingu 9.7

```
progs/format0.cpp: In function ‘int main()’:
progs/format0.cpp:6: warning: format argument is not a pointer (arg 2)
progs/format0.cpp:7: warning: char format, different type arg (arg 2)
progs/format0.cpp:8: warning: int format, different type arg (arg 2)
progs/format0.cpp:9: warning: int format, pointer arg (arg 2)
```

Listing 9.9: Poprawiona wersja programu z listingu 9.7

```
01 #include <stdio.h>
02 int main() {
03     int a;
04     char b[100];
05     long long c;
06     scanf("%d\n", &a);
07     scanf("%s\n", b);
08     printf("%lld\n", c);
09     printf("%c\n", b[0]);
10     return 0;
11 }
```

Opcja *-Wformat* okazuje się bardzo pożyteczna podczas pisania programów, które korzystają z biblioteki `stdio` zamiast `iostream`, (robi się tak zazwyczaj ze względów wydajnościowych — `stdio` jest istotnie szybsza od `iostream`). Błędy związane ze złymi typami przekazywanych parametrów, pojawiają się zazwyczaj nie bezpośrednio podczas pisania programu, lecz dopiero później, przy nanoszeniu poprawek. Przykładowo, jeśli zawodnik po napisaniu programu dochodzi do wniosku, że zmienna przechowująca wynik powinna być typu **long long**, a nie tak jak do tej pory **int**, to zmodyfikuje on typ tej zmiennej, ale może zapomnieć o odpowiednim poprawieniu wywołań funkcji `printf`.

9.5.3. *-Wshadow*

Kolejnym parametrem, który nie jest automatycznie włączany wraz z użyciem *-Wall*, jest *-Wshadow*. Parametr ten służy do wyszukiwania miejsc w programie, w których następuje przykrycie zmiennych globalnych przez zmienne lokalne. W przypadku wykrycia takich sytuacji, generowane są ostrzeżenia.

Kilkakrotnie podczas treningów, mój zespół spotkał się z sytuacją, w której zaimplementowany program nie działał poprawnie właśnie ze względu na ten problem. Wydawać by się mogło, że wykrycie takiego błędu jest dość proste, jednak w praktyce, podczas czytania kodu źródłowego na kartce, główną uwagę poświęca się na analizę złożonych wzorów oraz skomplikowanych operacji, nie zwracając uwagi na powtarzające się nazwy zmiennych.

Rozwiązanie kwestii związanej z wykorzystaniem w programie złej zmiennej o takiej samej nazwie, może polegać na zabronieniu wielokrotnego użycia zmiennych o tych samych nazwach. Wyjątek od reguły mogą stanowić często wykorzystywane zmienne tymczasowe, takie jak `tmp`, `x`, `y`, ... Ręczne sprawdzanie spełniania tego założenia jest podczas zawodów niewykonalne, na szczęście kompilator języka C++ przychodzi z pomocą, udostępniając parametr *-Wshadow*. Przedstawiony na listingu 9.10 program wyznacza kolejne wartości współczynnika

dwumianowego Newtona $\binom{n}{k}$ dla zadanej liczby n. Do swojego działania wykorzystuje funkcję `Binom`, która korzysta z globalnej zmiennej `n`. Funkcja ta deklaruje w linii 7 lokalną zmienną `n`, która jest używana zamiast zmiennej globalnej w linii 9. Kompilacja tego programu bez dodatkowych opcji nie generuje żadnych ostrzeżeń. Wykorzystanie flagi *-Wshadow* powoduje wygenerowanie ostrzeżeń dotyczących przysłaniania zmiennej globalnej przez zmienną lokalną (patrz listing 9.11).

Listing 9.10: Program prezentujący problem przykrywania zmiennej globalnej przez lokalną

```
01 #include <iostream>
02 using namespace std;
03 #define FOR(x,n,m) for(x=n;x<=m;x++)
04 int n;
05 int Binom(int k) {
06     int res = 1, x;
07     FOR(x, 1, n) res *= x;
08     int n;
09     FOR(n, 1, k) res /= n;
10     FOR(x, 1, n - k) res /= x;
11     return res;
12 }
13 int main() {
14     cin >> n;
15     for (int x = 0; x < n; x++)
16     cout << Binom(x) << endl;
17     return 0;
18 }
```

Listing 9.11: Ostrzeżenia wygenerowane przez proces kompilacji z włączoną flagą *-Wshadow* dla programu z listingu 9.10

```
progs/wshadow0.cpp: In function ‘int Binom(int)’:
progs/wshadow0.cpp:8: warning: declaration of ‘n’ shadows a global declaration
progs/wshadow0.cpp:4: warning: shadowed declaration is here
```

9.5.4. *-Wsequence-point*

Pisanie jak najkrótszych programów często wiąże się ze stosowaniem zagnieżdżonych instrukcji oraz operatorów. W takich sytuacjach, bardzo pożyteczny okazuje się również operator przecinka, pozwalający na łączenie kilku instrukcji w jedną. Podejście polegające na pisaniu jak najkrótszych programów, jest z jednej strony pożądane podczas konkursów informatycznych ze względu na możliwość skrócenia czasu implementacji, z drugiej zaś strony w istotny sposób zamazuje czytelność programu oraz utrudnia wyszukiwanie błędów. W ramach przykładu, przeanalizujmy rozwiązanie zadania „Krótki program”, które było pracą domową na zajęciach ze *Sztuki programowania* na Uniwersytecie Warszawskim.

Zadanie: Krótki program

Pochodzenie: Zajęcia ze Sztuki programowania na Uniwersytecie Warszawskim

Rozwiązanie: `short.cpp`

Zadanie polega na napisaniu jak najkrótszego program w C++, który szuka najkrótszej drogi w trójwymiarowym labiryncie.

Zadanie

Napisz program, który:

- wczyta opis labiryntu,
- wyznaczy odległość między punktem początkowym a końcowym,
- wypisze wynik.

Wejście

W pierwszym wierszu wejścia znajdują się trzy liczby całkowite w, s oraz d, oznaczające odpowiednio wysokość, szerokość oraz długość labiryntu. W kolejnych wierszach znajdują się opisy w poziomów labiryntu. Każdy z nich przedstawiony jest jako s wierszy po d znaków. Puste pola labiryntu oznaczone są przez spacje, ściany labiryntu to #, start zaznaczony jest literą S, natomiast meta przez literę M.

Wyjście

W pierwszym i jedynym wierszu wyjścia powinna znaleźć się jedna liczba — odległość między startem a metą.

Przykład

Dla następującego wejścia:

```
2 4 3
S  #
##
###
####
M ##
#
```

Poprawnym rozwiązaniem jest:

```
10
```

Listing 9.12: Przykład krótkiego programu

```
01 #include<stdio.h>
02 int h, w, l, s, x, v, k = 1;
03 char b[40];
04 main() {
05   gets(b);
06   sscanf(b, "%d%d%d", &h, &w, &l);
07   char t[s = (h + 2) * (w + 2) * (l + 2)];
08   int d[s], e[99], que[s],
09     dis[] = {-1, 1, -(v = w + 2), v, -v * (l + 2), v * (l + 2)};
10   for(x = 0; x < s; ++x) t[x] = 0, d[x] = -1;
```

Listing 9.12: (c.d. listingu z poprzedniej strony)

```
11    for(x = 0; x < h * w * l; ++x) (x % w ? 0 : gets(t + (v = (w + 2) *
12       (l + 3 + 2 * (x / (w * l)) + x / w) + 1))), e[t[v + (x % w)]] = v + (x % w);
13    for(d[que[0] = e['S']] = x = 0; x < 6 * k; ++x)
14       if(t[v = que[x / 6] + dis[x % 6]] != '#' && t[v] && d[v] == -1)
15          d[que[k++] = v] = d[que[x / 6]] + 1;
16    printf("%d\n", d[e['M']]);
17 }
```

Przykładowy krótki program rozwiązujący to zadanie został przedstawiony na listingu 9.12. Na jego przykładzie widać, jak bardzo można „skompresować" kod źródłowy. Dzięki temu jest on krótki, ale podejście takie istotnie pogorsza czytelność programu. Wykorzystywanie wielokrotnie zagnieżdżonych instrukcji niesie ze sobą dodatkowe zagrożenie polegające na złej kolejności wykonywania operacji.

Oczywistym jest, że prosta instrukcja postaci `a = b++;` spowoduje przypisanie zmiennej `a` wartości zmiennej `b`, a następnie zwiększenie o jeden wartości zmiennej `b`. W przypadku instrukcji `a = ++b;` kolejność wykonanych operacji zostanie odwrócona. Ale co będzie się działo w przypadku bardziej rozbudowanych instrukcji? Zapewne niewiele osób miało przyjemność świadomego użycia instrukcji postaci `n = n++;`, która nie powoduje zmiany wartości zmiennej `n`. Jej pojawienie się w programie zazwyczaj wiąże się z pomyleniem nazwy zmiennej, o co nietrudno w przypadku jednoliterowych nazw.

Kompilator GCC udostępnia opcję *-Wsequence-point*, która pozwala na wychwytywanie tego typu błędów. W aktualnej wersji kompilatora, opcja ta działa tylko dla programów implementowanych w języku C — wspomaganie dla języka C++ ma być dodane w przyszłości. Przykład działania tej flagi kompilatora dla programu z listingu 9.13 został przedstawiony na listingu 9.14.

Listing 9.13: Przykładowy program wykonujący operacje, których kolejność nie jest łatwa do wyznaczenia

```
01 #include <stdio.h>
02 int main() {
03    int a[10], b[10], n = 0, x;
04    for (x = 0; x < 10; x++)
05    a[x] = 0, b[x] = 1;
06    n = n++;
07    a[n] = b[n++];
08    a[n++] = n;
09    for (x = 0; x < 10; x++)
10    printf("%d ", a[x]);
11    return 0;
12 }
```

Listing 9.14: Ostrzeżenia wygenerowane przez proces kompilacji programu z listingu 9.13 z włączoną opcją *-Wsequence-point*

```
progs/sequence_point.c: In function 'main':
progs/sequence_point.c:5: warning: operation on 'n' may be undefined
progs/sequence_point.c:6: warning: operation on 'n' may be undefined
```

Listing 9.14: (c.d. listingu z poprzedniej strony)

```
progs/sequence_point.c:7: warning: operation on ‘n’ may be undefined
```

Opcja *-Wsequence-point* jest włączana automatycznie wraz z *-Wall*. Dla zainteresowanych, na listingu 9.15 przedstawiony jest wynik działania programu z przykładu 9.13

Listing 9.15: Wynik działania programu z listingu 9.13

```
0 1 2 0 0 0 0 0 0 0
```

Przy okazji omawiania krótkich programów, na listingu 9.16 zaprezentowany jest bardzo krótki program (zaledwie 48 znaków) napisany w języku C, wyznaczający ostatnią cyfrę liczby $n!$. Program ten jest autorstwa Tomasza Idziaszka. Długość tego kodu jest zawdzięczona między innymi brakowi jawnej deklaracji zmiennej `n` (w języku C nie ma konieczności deklarowania typu parametru funkcji `main`, w C++ takie podejście nie jest dozwolone) oraz brakowi typu zwracanego przez funkcję `main`. Bardzo ciekawy jest również sposób wczytywania i wypisywania wyniku. Instrukcja `gets(&n)` wczytuje do zmennej `n` liczbę k (program ma wyznaczyć ostanią cyfrę liczby k). Istotna jest w tym miejscu obserwacja, że wczytywana liczba k jest traktowana przez funkcję `gets` jako tekst, zatem jeśli $k = 0$, to $n = 48$, jeśli $k = 1$, to $n = 49$, ... jeśli $k > 4$, to $n > 52$. Wypisywanie wyniku dokonywane jest przy użyciu funkcji `putchar` — jeśli $n > 52$, to ostanią cyfrą liczby $k!$ jest 0, zatem wypisana jako wynik zostaje ostatnia cyfra napisu `112640`. Jeśli natomiast $n < 52$, to jako wynik zwracana jest odpowiednio inna cyfra (z pozycji $n - 48$).

Listing 9.16: Program wyznaczający ostatnią cyfrę liczb $n!$

```
1 main(n) {
2   gets(&n);
3   putchar("112640"[n > 52 ? 5 : n & 7]);
4 }
```

9.5.5. *-Wunused*

Stosunkowo częstym błędem, który trudno jest wykryć podczas analizy kodu programu na kartce, jest odwołanie się do złych zmiennych. Problem ten został poruszony już w poprzednim rozdziale. W niektórych przypadkach, wykorzystanie w pewnym miejscu złej zmiennej powoduje, że inna zainicjalizowana zmienna nie jest w ogóle używana (przykładowo, program mógł wyznaczyć pewien wynik pośredni i umieścić go w zmiennej, lecz w dalszej części programu wartość ta jest omyłkowo niewykorzystywana). Kompilator również i w tym przypadku umożliwia wykrywanie tego typu błędów. Należy w tym celu skorzystać z opcji *-Wunused*. Opcja ta zawiera w sobie kilka innych flag:

- *-Wunused-function* — generuje ostrzeżenie w przypadku występowania w programie niewykorzystywanych funkcji typu **static**. Niestety opcja ta nie wykrywa wszystkich niewykorzystywanych funkcji.
- *-Wunused-label* — generuje ostrzeżenie dla każdej zdefiniowanej, nieużytej etykiety.

- *-Wunused-parameter* — wykrywa brak wykorzystania parametrów funkcji.
- *-Wunused-variable* — wykrywa zadeklarowane, lecz niewykorzystane zmienne.
- *-Wunused-value* — wykrywa wyrażenia występujące w programie, których wartość nie jest wykorzystywana.

Niektóre z wyżej omówionych opcji można włączyć nie tylko przy użyciu flagi *-Wunused* ale również *-Wall*. Najlepszym sposobem na włączenie wszystkich tych opcji jest użycie dwóch flag — *-Wall* oraz *-W*. Na listingu 9.17 przedstawiony został przykładowy program, który deklaruje niewykorzystaną funkcję statyczną, zmienne, wyrażenie oraz etykietę. Kompilacja tego programu bez żadnych opcji nie generuje ostrzeżeń, dopiero włączenie opcji *-Wall* oraz *-W* pozwala na wykrycie potencjalnych błędów. Ostrzeżenia generowane przez proces kompilacji dla tego programu przedstawione zostały na listingu 9.18.

Listing 9.17: Program zawierający niewykorzystywane zmienne

```
1 static int foo(int a) {
2     return 10;
3 }
4 int main() {
5     int a = 10, b = 5;
6     a *a;
7     label:
8     return 0;
9 }
```

Listing 9.18: Wynik kompilacji

```
progs/unused.cpp: In function ‘int foo(int)’:
progs/unused.cpp:1: warning: unused parameter ‘int a’
progs/unused.cpp: In function ‘int main()’:
progs/unused.cpp:5: warning: unused variable ‘int b’
progs/unused.cpp:7: warning: label ‘label’ defined but not used
progs/unused.cpp:6: warning: statement with no effect
progs/unused.cpp: At top level:
progs/unused.cpp:1: warning: ‘int foo(int)’ defined but not used
```

9.5.6. *-Wuninitialized*

Kolejną przydatną opcją kompilatora jest flaga, pozwalająca na wykrywanie zmiennych, które nie zostały zainicjalizowane przed pierwszym użyciem. Jest do dość pożyteczna opcja, gdyż często pisząc program zakłada się niesłusznie, iż wartość nowotworzonych zmiennych ustawiana jest na 0. Nie jest to jednak zawsze prawdą. W niektórych przypadkach tak faktycznie będzie, gdyż system operacyjny czyści przydzielaną programom pamięć ze względów bezpieczeństwa. Okazać się może, że kompilacja programu z optymalizacjami wygeneruje program, który będzie wykorzystywał ten sam obszar pamięci dla różnych zmiennych. Program, który zakłada, że nowotworzone zmienne mają wartość 0, może działać podczas testowania przez zawodnika, a dopiero na komputerach jury ujawnią się błędy. Użycie opcji

-*Wuninitialized* pozwala na wykrycie tego typu problemów. Dla programu przedstawionego na listingu 9.19, kompilacja bez żadnych dodatkowych opcji kompilatora kończy się bez komunikatów. Dopiero użycie -*Wuninitialized* generuje ostrzeżenia — są one przedstawione na listingu 9.20.

Listing 9.19: Program zawierający niezadeklarowane zmienne

```
01 #include <stdio.h>
02 int foo() {
03     int res;
04     return res;
05 }
06 int main() {
07     int a, b;
08     b = a * a;
09     printf("%d\n", b);
10     return 0;
11 }
```

Listing 9.20: Wynik kompilacji

```
progs/uninitialized.cpp: In function 'int foo()':
progs/uninitialized.cpp:3: warning: 'int res' might be used uninitialized in
    this function
progs/uninitialized.cpp: In function 'int main()':
progs/uninitialized.cpp:7: warning: 'int a' might be used uninitialized in this
    function
```

9.5.7. -*Wfloat-equal*

W przypadku rozwiązywania zadań geometrycznych często pojawiają się dodatkowe problemy, związane z wykonywanymi zaokrągleniami wartości zmiennych. Powoduje to, że nie tylko wartość wyliczonego wyniku może być niedokładna, ale cały program może działać nieprawidłowo. Istnieje niebezpieczeństwo, że wykonane porównanie dwóch zmiennych typu zmiennoprzecinkowego postaci `a < b` powinno teoretycznie zwrócić prawdę, jednak wykonane przez program zaokrąglenia spowodowały, że zachodzi nierówność odwrotna — `a > b`. Porównywanie wartości zmiennoprzecinkowych jest szczególnie niebezpieczne w przypadku operacji postaci `a == b`. W takich sytuacjach, nawet wyliczone w identyczny sposób zmienne `a` i `b` mogą mieć inną wartość (przyczyną mogą być m.in. realizowane przez kompilator optymalizacje kodu), zatem porównanie `a == b` praktycznie zawsze zwraca fałsz. Dlatego też, wykonywanie porównań wartości zmiennoprzecinkowych postaci `a == b` jest bardzo niebezpieczne.

W obliczu konieczności wykonania takiego porównania, należy brać pod uwagę pewną granicę błędu — problem ten jest dokładniej omówiony w rozdziale dotyczącym geometrii obliczeniowej. Dobrym pomysłem jest dodatkowe zabezpieczenie się przed możliwością popełnienia tego typu błędu. Opcja -*Wfloat-equal* kompilatora pozwala na weryfikację poprawności programu pod względem braku porównań zmiennoprzecinkowych postaci `a == b`. Na listingu 9.21 przedstawiony jest nieprawidłowy program, którego kompilacja kończy się wygen-

erowaniem ostrzeżeń umieszczonych na listingu 9.22.

Listing 9.21: Program wykonujący niebezpieczne porównanie

```
1 int main() {
2     float a = 10.0, b = 8.0;
3     bool equal = (a == b);
4     return 0;
5 }
```

Listing 9.22: Wynik kompilacji programu z listingu 9.21 z użyciem flagi *-Wfloat-equal*

```
progs/float.cpp: In function ‘int main()’:
progs/float.cpp:3: warning: comparing floating point with == or != is unsafe
```

9.6. Nieustanny time-limit

Zdarzają się sytuacje, w których kolejne rozwiązania zadania ciągle otrzymują wynik przekroczenia czasu wykonania. Jednym z powodów takiej sytuacji jest zastosowanie nieefektywnego algorytmu. W takim przypadku, zazwyczaj należy wybrać szybszy algorytm i zmienić implementację całego programu. Jedynym odstępstwem od tej reguły jest, jeśli oczekiwana złożoność programu jest lepsza od aktualnej o czynnik logarytmiczny. W takich sytuacjach można zaryzykować i nie dokonywać zmiany wykorzystywanego algorytmu, lecz skupić się nad innym sposobem rozwiązania zaistniałego problemu.

Sposobem radzenia sobie z przekroczeniem czasu, nie wymagającym zmiany stosowanego algorytmu, jest przeprowadzanie optymalizacji, pozwalających na kilkukrotne przyspieszenie programu. Tego typu optymalizacje są szczególnie istotne w przypadku konkursów z rodziny *Olimpiady Informatycznej*, w których punktacja zadań jest płynna, a wyniki są ogłaszane dopiero po zakończeniu zawodów.

Istnieje kilka prostych sposobów, pozwalających na przyspieszenie działania programu:

- eliminacja dzielenia oraz obliczania reszty z dzielenia,
- stosowanie operatora **inline**,
- zmiana funkcji służących do wczytywania danych,
- korzystanie ze wstawek assemblerowych,
- kompilacja programu z optymalizacjami,
- modyfikacja programu, mająca na celu lepsze wykorzystanie pamięci podręcznej.

Oprócz wyżej wymienionych, istnieje jeszcze jedna metoda przyspieszania programów zwana preprocessingiem, jednak jest ona zależna od stosowanego algorytmu. Zasadniczo polega ona na wyliczaniu pewnych wartości wykorzystywanych przez algorytm przed lub podczas kompilacji programu, co pozwala na zaoszczędzenie czasu podczas samego wykonania programu.

Przeanalizujmy po kolei przedstawione techniki optymalizacji.

9.6.1. Eliminacja dzielenia

Pierwsza z optymalizacji — eliminacja operacji dzielenia, w przypadku zadań wykonujących liczne operacje arytmetyczne, może przynieść ogromne korzyści. Operacja dzielenia jest bardzo kosztowna, przez co jej wyeliminowanie z programu może przynieść nawet kilkukrotne przyspieszenie. Obchodzenie się bez operacji dzielenia nie jest rzeczą łatwą, jednak w wielu przypadkach można sobie poradzić przy użyciu operacji przesunięcia bitowego (`>>`). Przykładowo, w celu podzielenia wartości zmiennej `a` przez `4`, zamiast pisać `a = a / 4;`, można wykonać operację `a = a >> 2;`. Operacja taka (w zależności od architektury komputera) działa około 5 razy szybciej. Jeśli program jest kompilowany z optymalizacjami, to kompilator jest w stanie w wielu przypadkach sam zastąpić dzielenie oraz wyznaczanie reszty z dzielenia innymi operacjami.

9.6.2. Wczytywanie wejścia

Czas dostępu do informacji zapisanych na dysku twardym jest istotnie dłuższy od odwoływania się do pamięci operacyjnej. W przypadku pisania programów konkursowych sytuacja jest podobna. Czas wczytywania i wypisywania danych stanowi w wielu sytuacjach znaczną część czasu wykonania programu. Na efektywność tych czynności w dużym stopniu ma wpływ sposób realizacji operacji wejścia/wyjścia.

W języku C++ szeroko wykorzystywane są dwie metody. Jedną z nich jest użycie znanych jeszcze z języka C, funkcji `scanf` oraz `printf`. Są one stosunkowo szybkie, jednak posługiwanie się nimi jest niewygodne ze względu na konieczność podawania formatu danych, na których wykonywane są operacje.

Drugą metodą jest stosowanie strumieni wprowadzonych w języku C++. Ich użycie jest bardzo proste. Strumienie „same" określają format danych, na podstawie typów zmiennych, biorących udział w realizowanych operacjach. Wygoda jednak niesie ze sobą również wielkie koszty — strumienie są istotnie wolniejsze od funkcji `scanf` oraz `printf`. Różnice w ich czasie działania wahają się wraz z różnymi wersjami kompilatorów, jednak należy zakładać, że strumienie są średnio pięciokrotnie wolniejsze. Na listingach 9.23 oraz 9.24 przedstawione zostały dwa programy wczytujące i wypisujące 1 000 000 liter. Czas działania pierwszego z nich na komputerze testowym to 6.5 sekund, podczas gdy drugi program potrzebuje niecałe półtorej sekundy.

Listing 9.23: Program wczytujący i wypisujący 1 000 000 liczb, wykorzystujący do tego celu strumienie

```
01 #include <iostream>
02 using namespace std;
03 int main() {
04     char w;
05     for (int x = 0; x < 1000000; x++) {
06         cin >> w;
07         cout << w;
08     }
09     return 0;
10 }
```

Listing 9.24: Program wczytujący i wypisujący 1 000 000 liczb, wykorzystujący do tego celu funkcje `scanf` oraz `printf`

```
1 #include <stdio.h>
2 int main() {
3    char w;
4    for (int x = 0; x < 1000000; x++) {
5       scanf("%c", &w);
6       printf("%c", w);
7    }
8    return 0;
9 }
```

Przedstawiona analiza świadczy o tym, że podczas rozwiązywania zadań konkursowych nie należy korzystać ze strumieni, chyba że liczba wczytywanych i wypisywanych danych jest niewielka.

Podczas jednego z konkursów, w których zespół Warsaw Predators brał udział, oczekiwana złożoność rozwiązania jednego z zadań o dużym wejściu była liniowa. Jednak dzięki zoptymalizowaniu operacji wczytywania danych, udało się uzyskać akceptację rozwiązania działającego w czasie $O(n * log(n))$. Wykorzystanie funkcji `scanf` było jednak niewystarczające, co zmusiło nas do wykorzystania funkcji `getchar` i `putchar`. Przykładowy program został przedstawiony na listingu 9.25. Czas jego wykonania na tym samym komputerze co poprzednie programy to $0,42$ sekundy.

Listing 9.25: Program wczytujący i wypisujący 1 000 000 liter, wykorzystujący do tego celu funkcje `getchar` oraz `putchar`

```
1 #include <stdio.h>
2 int main() {
3    char w;
4    for (int x = 0; x < 1000000; x++) {
5       w = getchar();
6       putchar(w);
7    }
8    return 0;
9 }
```

9.6.3. Kompilacja z optymalizacjami i wstawki assemblerowe

Kolejną techniką, pozwalającą na zwiększenie wydajności programu, jest korzystanie ze wstawek assemblerowych. W języku C++ wstawki można dodawać do programu przy użyciu operatora `asm`. Podejście tego typu pozwala na najdokładniejszą kontrolę tego, co jest wykonywane przez implementowany program. W przypadku korzystania z instrukcji języka wysokiego poziomu, jakim jest C++, nie jest wiadomo, jaki kod maszynowy zostanie wygenerowany dla określonej sekwencji. W wielu przypadkach człowiek jest w stanie napisać znacznie szybszy kod niskopoziomowy. Podejście wykorzystujące wstawki assemblerowe było dość często stosowaną techniką przyspieszania programów, jednak obecnie jest ona coraz rzadziej spotykana, ze względu na czasochłonność i łatwość popełniania błędów. Jakość dzisiejszych kom-

pilatorów jest na tyle wysoka, że ręczne pisanie kodu maszynowego nie daje tak wielkich różnic jak kiedyś.

Skoro jakość generowanego kodu maszynowego podczas zawodów powierza się w całości kompilatorowi, to należy upewnić się, że zadanie to jest wykonane jak najlepiej. Jedną z możliwości wpływania na ten proces jest poprzedzanie funkcji słowem kluczowym **inline**. Stanowi ono podpowiedź dla kompilatora, że treść funkcji powinna być wstawiana w miejsce jej wywołania, zamiast wykonywania skoku do miejsca, w którym się ona znajduje. W przypadku prostych funkcji, takich jak wyznaczanie maksimum dwóch liczb, podejście takie w istotny sposób zwiększa efektywność programu. Stosowanie operatora **inline** pozwala na zaoszczędzenie czasu potrzebnego na odkładanie argumentów na stosie oraz wykonywanie operacji związanych z wywołaniem funkcji. W przypadku kompilacji programu z optymalizacjami, rozwijanie wywołań funkcji wykonywane jest automatycznie, jednak operator **inline** może pomóc kompilatorowi w podejmowaniu decyzji, które funkcje należy rozwijać.

Listing 9.26: Program wykonujący 100 000 000 odwołań do tablicy liczb

```
1 using namespace std;
2 int main() {
3     int a[16];
4     for (int x = 0; x < 100000000; x++)
5     a[x & 15] = x;
6     return 0;
7 }
```

Listing 9.27: Program wykonujący 100 000 000 odwołań do wektora liczb

```
1 #include <vector>
2 using namespace std;
3 int main() {
4     vector<int> a(16, 0);
5     for (int x = 0; x < 100000000; x++)
6     a[x & 15] = x;
7     return 0;
8 }
```

Drugą metodą generowania kodu maszynowego wysokiej jakości jest kompilacja programu z optymalizacjami (przełączniki kompilatora *-O1*, *-O2*, *-O3*, ...). Optymalizacje są szczególnie istotne w przypadku wykorzystywania zaawansowanych konstrukcji języka C++, które znajdują szerokie zastosowanie w bibliotece STL.

Wyniki są widoczne na przykładzie programów z listingów 9.26 oraz 9.27. Pierwszy z nich wykonuje 100 000 000 zapisów do tablicy liczb, podczas gdy drugi wykorzystuje do tego samego celu wektor z biblioteki STL. Czasy wykonania tych programów kompilowanych z optymalizacjami na poziomie *-O2*, to odpowiednio 0,867 i 1,154 sekundy, zatem są one porównywalne. Jeśli te same programy zostaną skompilowane bez optymalizacji, to czasy wykonania rosną odpowiednio do 4 i 30 sekund. W przypadku brania udziału w konkursach, w których jury kompiluje programy bez optymalizacji, programy wykorzystujące bibliotekę STL mają bardzo utrudnione zadanie. Istnieje jednak metoda na obejście tego problemu, a polega ona na wstępnej kompilacji programu do kodu maszynowego, zanim odda się program do oceny.

Zastosowanie tej techniki nie wymaga żadnych zmian w sposobie pisania programu. Realizacja tego procesu wygląda następująco:

- Implementacja programu (nazwijmy go `a.cpp`).
- Kompilacja programu do kodu maszynowego przy użyciu polecenia

  ```
  gcc -S -Ox a.cpp
  ```

  gdzie x oznacza wykorzystywany poziom optymalizacji. Zazwyczaj stosuje się poziom 2, gdyż daje bardzo dobre efekty i nie niesie ze sobą ryzyka zmiany działania programu. Wyższe poziomy optymalizacji mogą okazać się niebezpieczne i mieć wpływ na sposób działania programu. Pomyślnie zakończony proces kompilacji powinien spowodować powstanie pliku `a.s`, zawierającego kod maszynowy programu.
- Ponieważ jury oczekuje, że rozwiązania zadań będą nadsyłane w języku C++, a nie Assemblerze, zatem należy dokonać modyfikacji kodu maszynowego w taki sposób, aby można było dokonać kompilacji przy pomocy kompilatora języka C++. Proces ten wymaga umieszczenia na początku pliku `a.s` tekstu „`asm(`” oraz na jego końcu „`);`”. Poza tym, należy: zamienić występujące w programie znaki ” na \”, umieścić na początku każdego wiersza \” oraz na końcu każdego wiersza `\n"`. Można tego dokonać przy użyciu edytora tekstu, lub wykonując następującą komendę:

  ```
  perl -pi~ -e 's/\"/\\\"/g; s/(.*)/\"$1\\\n\"/ig' a.s
  ```
- Zawartość plku `a.s` można skompilować jak zwykły kod źródłowy napisany w języku C++.

9.6.4. Lepsze wykorzystanie pamięci podręcznej

Wielki wpływ na czas wykonania programu może również mieć kolejność, w jakiej program odwołuje się do pamięci. Różnice zależą w istotny sposób od architektury procesora, a dokładniej od rodzaju oraz wielkości jego pamięci podręcznej. Szybkość działania pamięci RAM w komputerze jest około dziesięciokrotnie wolniejsza od prędkości procesora. W przypadku, gdy istnieje konieczność odwołania się do pewnej komórki pamięci, procesor musi czekać na sprowadzenie potrzebnych danych z pamięci RAM. W celu poprawienia tej sytuacji, dzisiejsze procesory wyposażone są w pamięć podręczną (ang. cache), której prędkość jest istotnie większa od pamięci RAM. Zadaniem pamięci podręcznej jest przechowywanie ostatnio wykorzystywanych adresów pamięci, co pozwala zaoszczędzić czas tracony w oczekiwaniu na sprowadzenie odpowiednich informacji z RAM-u.

W przypadku zadań konkursowych, najwięcej czasu przy wykonywaniu programu można zaoszczędzić poprzez zmianę kolejności inicjalizacji tablic. Sposób wykonywania tego procesu jest szczególnie ważny podczas rozwiązywania zadań na programowanie dynamiczne, które często wielokrotnie muszą aktualizować zawartość wielowymiarowych tablic. Na listingach 9.28 oraz 9.29 przedstawione są dwa programy, które wykonują identyczne zadanie — wypełniają 100 megabajtową tablicę różnymi wartościami. Jedyna różnica polega na kolejności przetwarzania pól tablicy. Pierwszy z tych programów wypełnia tablicę „wierszami”, podczas gdy drugi robi to „kolumnami”. W przypadku pierwszego programu, odwołania do

pamięci realizowane są lokalnie — kolejno modyfikowane bloki pamięci są spójne. Drugi program natomiast wykonuje liczne skoki po pamięci (odległość między kolejnymi zapisywanymi blokami pamięci wynosi 10000 bajtów), co uniemożliwia efektywne wykorzystanie pamięci podręcznej procesora. Odbija się to znacząco na czasie wykonania obu programów. Pierwszy z nich działa na testowej maszynie 1.2 sekundy, podczas gdy drugi ponad 15 sekund.

Listing 9.28: Program zapisujący tablicę „wierszami”

```
1 char tab[10000][10000];
2 int main() {
3     for (int x = 0; x < 10000; x++)
4     for (int y = 0; y < 10000; y++)
5     tab[x][y] = x + y;
6     return 0;
7 }
```

Listing 9.29: Program zapisujący tablicę „kolumnami”

```
1 char tab[10000][10000];
2 int main() {
3     for (int y = 0; y < 10000; y++)
4     for (int x = 0; x < 10000; x++)
5     tab[x][y] = x + y;
6     return 0;
7 }
```

9.6.5. Preprocessing

Kolejną metodą, pozwalającą na przyspieszenie wykonania programów, jest technika zwana preprocessingiem. Odmiennie od technik omówionych uprzednio, jest ona zależna od stosowanego algorytmu i polega na wyliczaniu pewnych danych, których wartość nie jest uwarunkowana od danych wejściowych programu, przed lub podczas kompilacji. W ten sposób program nie musi wyznaczać potrzebnych do swojego działania informacji podczas wykonywania operacji — wystarczy, że skorzysta ze stablicowanych wyników wyliczonych wcześniej. Przy zastosowaniu powyższej metody, w przypadku niektórych zadań, można sobie pozwolić na implementację znacznie mniej efektywnego algorytmu. Wyznaczanie danych dla programu można bowiem wykonywać podczas rozwiązywania innych zadań. Przykładem tego typu zadania są *Liczby antypierwsze* z pierwszego etapu *VIII Olimpiady Informatycznej*.

Zadanie: Liczby antypierwsze

Pochodzenie: VIII Olimpiada Informatyczna

Rozwiązanie: `ant.cpp`

Dodatnią liczbę całkowitą nazywamy antypierwszą, gdy ma ona więcej dzielników niż każda dodatnia liczba całkowita mniejsza od niej. Przykładowymi liczbami antypierwszymi są: 1, 2, 4, 6, 12 i 24.

Zadanie

Napisz program, który:

- wczyta dodatnią liczbę całkowitą n,
- wyznaczy największą liczbę antypierwszą nieprzekraczającą n,
- wypisze wyznaczoną liczbę.

Wejście

W jedynym wierszu wejścia znajduje się jedna liczba całkowita n, $1 < n < 2\,000\,000\,000$.

Wyjście

W jedynym wierszu wyjścia program powinien zapisać dokładnie jedną liczbę całkowitą — największą liczbę antypierwszą nieprzekraczającą n. Efektywne rozwiązanie tego zadania,

Przykład

Dla następującego wejścia:	Poprawnym rozwiązaniem jest:
1000	840

wymaga zastosowania pewnych własności rozkładu liczb na liczby pierwsze. Okazuje się jednak, że liczb antypierwszych w przedziale $\{1, 2\,000\,000\,000\}$ jest niewiele — zaledwie 68. Można zatem je wszystkie wyznaczyć wcześniej, a następnie umieścić ich listę w programie. Ze względu na to, że zadanie to pojawiło się na pierwszym etapie Olimpiady i na jego rozwiązanie zawodnicy mieli około miesiąca, zatem można było zastosować w zasadzie dowolną metodę wyznaczania liczb antypierwszych. Program prezentujący zastosowanie techniki preprocessingu w praktyce został przedstawiony na listingu 9.30

Listing 9.30: Rozwiązanie zadania *Liczby antypierwsze*, wykorzystujące technikę preprocessingu

```
// Tablica wszystkich liczb antypierwszych na przedziale [1..2*10^9]
int vals[] = { 1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, 1260,
   1680, 2520, 5040, 7560, 10080, 15120, 20160, 25200, 27720, 45360, 50400,
   55440, 83160, 110880, 166320, 221760, 277200, 332640, 498960, 554400, 665280,
   720720, 1081080, 1441440, 2162160, 2882880, 3603600, 4324320, 6486480,
   7207200, 8648640, 10810800, 14414400, 17297280, 21621600, 32432400, 36756720,
   43243200, 61261200, 73513440, 110270160, 122522400, 147026880, 183783600,
   245044800, 294053760, 367567200, 551350800, 698377680, 735134400, 1102701600,
   1396755360 };

int main() {
   int nr, pos = 67;
   cin >> nr;
   while(vals[pos] > nr) pos--;
   cout << vals[pos] << endl;
   return 0;
}
```

Oprócz przedstawionych w tym rozdziale technik pozwalających na zwiększanie efektywności programów, istnieją inne zaawansowane metody, z których wiele wymaga dobrej znajomości

architektury komputerów. Optymalizacje te są związane między innymi z kolejnością wykonywania operacji, czy sposobem odwoływania się do pamięci. Bardzo dobrym źródłem wiedzy na ten temat, przydatnej nie tylko podczas brania udziału w konkursach, ale również w codziennym życiu programisty, jest książka „Optymalizacja Kodu. Efektywne wykorzystanie pamięci - programowanie” ([OPK]).

Rozdział 10

Rozwiązania zadań

W rozdziale tym znajdują się wskazówki dotyczące rozwiązań zadań z poszczególnych rozdziałów książki. Zadania mogą być trudne z dwóch względów. Pierwszym z nich jest skomplikowana implementacja algorytmu. W takiej sytuacji, przydatna może się okazać analiza kodów źródłowych programów znajdujących się na płycie dołączonej do książki. Znajdują się na niej bowiem programy, stanowiące rozwiązania wszystkich zadań zawartych w tej książce, zaimplementowane przy użyciu przedstawionej biblioteczki algorytmicznej. Kolejnym źródłem problemów może być samo wymyślenie odpowiedniego algorytmu. W takiej sytuacji, przydatna może się okazać zawartość niniejszego rozdziału, w obrębie którego można znaleźć podpowiedzi do poszczególnych zadań. Do każdego z nich dostępnych jest kilka podpowiedzi — każda kolejna wskazówka konkretyzuje pomysł naszkicowany w poprzedniej. Dzięki takiemu podejściu, jeśli początkowe wskazówki okazują się niewystarczające do rozwiązania zadania, można pokusić się o przeczytanie następnych. Z uwagi na to, że każde zadanie można rozwiązać na wiele sposobów, natomiast wskazówki nakierowują na jedno konkretne rozwiązanie, nie należy zatem przejmować się zbytnio, jeśli koncepcja rozwiązania przedstawiona w podpowiedziach nie pokrywa się z podejściem czytelnika.

10.1. Algorytmy grafowe

10.1.1. Mrówki i biedronka

- Złożoność asymptotyczna proponowanego rozwiązania zadania wynosi *O(n*l)*
- W momencie lądowania biedronki wszystkie mrówki poruszają się w jej kierunku. Dobrym pomysłem jest wyznaczenie dla każdego liścia kierunku, w którym należy się z niego poruszać. Ponieważ rozpatrywana struktura jest drzewem, zatem do wyznaczenia tych kierunków można zastosować algorytm BFS.
- Dla każdego kolejnego lądowania biedronki można wyznaczyć drzewo BFS, o którym mowa w powyższej wskazówce, a następnie symulować ruchy mrówek do momentu, w którym jedna z nich dotrze do biedronki. W jaki sposób można wykonać symulację, aby czas jej wykonania był liniowy ze względu na liczbę liści w drzewie?
- Problemem w efektywnej realizacji symulacji jest stwierdzenie, czy przetwarzana mrówka powinna wykonać ruch. W celu zweryfikowania tego należy sprawdzić, czy na ścieżce między nią a biedronką nie ma żadnej innej mrówki. Można tego szybko dokonać poprzez zaznaczanie liści w drzewie, na które nie można już wchodzić (gdyż na ścieżce do

miejsca lądowania biedronki znajduje się mrówka). Zaznaczanie liści można wykonywać po każdym pojedynczym ruchu mrówki; wszystkie wierzchołki z poddrzewa BFS tego wierzchołka należy zaznaczyć jako zablokowane. Sumaryczny czas wykonania pojedynczej symulacji jest liniowy, ponieważ każdy wierzchołek zaznaczany jest jako zablokowany dokładnie raz.

10.1.2. Komiwojażer Bajtazar

- Istnieje możliwość rozwiązania tego zadania w czasie $O(n+m)$, lecz algorytm jest dość skomplikowany. Sugerowana złożoność czasowa to $O((n+m)*log(n))$

- Podróże Bajtazara pomiędzy kolejnymi parami miast są niezależne, zatem zadanie można rozwiązać wyliczając wszystkie częściowe odległości, a następnie zwrócić sumę wyznaczonych wartości. Sieć dróg nigdy nie tworzy cykli oraz z każdego miasta można dotrzeć do każdego innego, zatem sieć ta tworzy drzewo i dla każdej pary miast istnieje dokładnie jedna ścieżka je łącząca. Rozwiązanie zadania sprowadza się zatem do zaprojektowania algorytmu wyznaczającego odległość między dowolnymi dwoma wierzchołkami w drzewie. Jak tego dokonać?

- W celu wyznaczania odległości między parami wierzchołków w drzewie można na wstępie ukorzenić analizowane drzewo w pewnym (dowolnym) wierzchołku r. Odległość między dwoma wierzchołkami v oraz w (którą oznaczać będziemy przez $d(v,w)$) w drzewie o korzeniu r można wyznaczyć, korzystając z pojęcia najniższego wspólnego przodka dwóch wierzchołków (ang. least common ancestor) — $p = LCA(v,w)$. Jest to najbardziej oddalony od korzenia drzewa wierzchołek, który leży na ścieżkach $r \rightsquigarrow v$ oraz $r \rightsquigarrow w$. Poszukiwaną wartość $d(v,w)$ można wyrazić wtedy jako $d(p,v)+d(p,w)$. W jaki sposób można wyznaczać w czasie logarytmicznym ($O(log(n))$ najniższego wspólnego przodka dwóch wierzchołków v i w w drzewie?

- Wyznaczanie najniższego wspólnego przodka dwóch wierzchołków można dokonać w czasie stałym. Opis takiego algorytmu można znaleźć w pracy [LCA]. Algorytm ten jednakże jest stosunkowo skomplikowany w implementacji. W rozwiązaniu zadania umieszczonego na płycie zastosowany został nieco prostszy algorytm. Zauważmy na wstępie, że wykonując na drzewie algorytm DFS, zaczynając od korzenia r, jesteśmy w stanie w czasie stałym odpowiadać na pytanie, czy wierzchołek p jest przodkiem wierzchołka v. Aby tak było musi zachodzić:

$$d(p) < d(v), f(p) > f(v)$$

  Znając listę wierzchołków występujących na ścieżce od wierzchołka v do korzenia r można wyznaczać wartość $LCA(v,w)$, stosując wyszukiwanie binarne. Podejście takie wymaga jednak skonstruowania pewnej reprezentacji ścieżek, której wielkość nie przekracza $O(n*log(n))$ oraz która pozwala efektywnie przeszukiwać ścieżkę. Jedną z możliwości jest wyznaczenie dla każdego węzła drzewa v listy wierzchołków leżących na ścieżce do korzenia oddalonych od v o $1, 2, 4, \ldots$.

10.1.3. Drogi

- Zadanie można rozwiązać w czasie $O(n+m)$.

- Sieć dróg w Bajtocji można przedstawić w postaci grafu skierowanego G. Problem postawiony w zadaniu jest równoważny wyznaczeniu minimalnej liczby krawędzi, jakie należy dodać do grafu G, aby z każdego wierzchołka dało się dojść do każdego innego.

- Jeżeli rozpatrzymy graf H silnie spójnych składowych dla grafu G, to okaże się, że nie trzeba dokładać żadnych dodatkowych dróg, jeśli graf H składa się z dokładnie jednego wierzchołka (wszystkie wierzchołki grafu G należą do tej samej silnie spójnej składowej). Aby zatem rozwiązać zadanie, należy wyznaczyć liczbę krawędzi, jaką należy dodać do grafu G, tak aby odpowiadający graf silnie spójnych składowych składał się z jednego wierzchołka. Zamiast operować na oryginalnym grafie, można również dodawać krawędzie do grafu H (dodana krawędź w grafie H reprezentuje krawędź łączącą dowolne wierzchołki z odpowiednich silnie spójnych składowych grafu G). Sprowadzenie grafu H do pożądanej postaci jest łatwiejsze niż grafu G, gdyż jest on acykliczny.

- Niech liczby a i b będą odpowiednio równe liczbie wierzchołków o stopniu wyjściowym 0 oraz o stopniu wejściowym 0 w grafie H. Poszukiwana liczba krawędzi, jaką należy dodać do grafu G jest równa $max(a, b)$.

10.1.4. Spokojna komisja

- Zadanie można rozwiązać w czasie $O(n + m)$.

- Zauważmy, że jeżeli posłowie v oraz w nawzajem się nie lubią, to wybranie do komisji posła v wymusza również wybranie posła z, który jest z posłem w razem w partii. W takim przypadku mówimy, że poseł z jest zależny od posła v. Na podstawie listy par posłów, którzy się nawzajem nie lubią, można skonstruować graf zależności G, a następnie, przy jego użyciu, wyznaczyć skład komisji.

- Jeśli do komisji zostanie wybrany poseł v, to muszą również zostać wybrani wszyscy posłowie, którzy są osiągalni z wierzchołka v w grafie zależności G. Zatem dla każdej silnie spójnej składowej grafu G, albo wszyscy posłowie z tej składowej należą do komisji, albo żaden z nich. Dodatkowo, jeśli pewien poseł z silnie spójnej składowej x został wybrany do komisji, to wszyscy posłowie ze składowych osiągalnych z x również muszą być w komisji. Jeśli w grafie G z wierzchołka reprezentującego posła v osiągalne są wierzchołki reprezentujące dwóch posłów z tej samej partii, to v nie może zostać wybrany do komisji.

- Rozpoczynając od pustego zbioru członków spokojnej komisji oraz rozpatrując wierzchołki silnie spójnych składowych grafu G w porządku odwrotnym do topologicznego, można zastosować metodę zachłanną konstruowania wyniku. Jeśli aktualnie przetwarzana składowa nie zawiera posłów, których dodanie do konstruowanego wyniku powodowałoby wystąpienie nielubiącej się pary posłów w komisji, to grupę taką można dodać do wyniku. Po przetworzeniu całego grafu, jeśli skonstruowany wynik spełnia wymagania zadania, to należy zwrócić go jako odpowiedź, w przeciwnym razie komisja nie może zostać ustanowiona.

10.1.5. Wirusy

- Zadanie można rozwiązać w czasie liniowym ze względu na sumaryczną długość wszystkich kodów wirusów.

- W rozwiązywaniu tego zadania pożyteczna może okazać się lektura rozdziału *7.3*, dotyczącego algorytmu Aho-Corasick do wyszukiwania wielu wzorców w tekście.
- Zauważmy, że jeżeli istnieje nieskończony, bezpieczny ciąg zer i jedynek t, to nie zawiera on żadnego z wirusów jako swojego podciągu. W takiej sytuacji, wykonując algorytm Aho-Corasick dla wzorców będących sekwencjami wirusów, algorytm nigdy nie trafi do wierzchołka w drzewie prefiksów wzorców, który reprezentuje cały wzorzec. Czy można wykorzystać drzewo prefiksów wzorców do wyznaczenia bezpiecznego nieskończonego ciągu zer i jedynek?
- Aby istniał nieskończony ciąg zer i jedynek, nie zawierający żadnego wirusa, to algorytm Aho-Corasick musi poruszać się po takim cyklu, w obrębie drzewa prefiksów, który nie zawiera żadnego węzła reprezentującego kod wirusa. Jeśli taki cykl nie istnieje, to ze względu na ograniczoną wielkość drzewa, algorytm w końcu trafi do węzła reprezentującego pewien wirus.
- Aby rozwiązać zadanie, należy skonstruować strukturę `mkmp` dla podanych kodów wirusów, a następnie sprawdzić, czy graf reprezentujący wzorce zawiera cykl (przy wyszukiwaniu cyklu należy traktować wartości funkcji prefiksowej jako krawędzie grafu).

10.1.6. Linie autobusowe

- Zadanie można rozwiązać w czasie $O(n+m)$.
- Oznaczmy poszukiwaną liczbę linii autobusowych przez k. Zastanówmy się najpierw, kiedy istnieje możliwość zagwarantowania transportu wszystkim mieszkańcom przy użyciu dokładnie jednej linii autobusowej. Ze względu na fakt, iż graf dróg jest spójny (z każdego miasta da się dojechać do każdego innego), trasę autobusu przebiegającą przez wszystkie drogi można utożsamić ze ścieżką Eulera. Jeśli ścieżka Eulera istnieje, to można rozwiązać zadanie wykorzystując dokładnie jedną linię autobusową.
- Ścieżkę Eulera można wyznaczyć w spójnym grafie wtedy, gdy każdy wierzchołek ma stopień parzysty lub gdy liczba wierzchołków o stopniu nieparzystym (l) jest równa 2. Ile linii autobusowych jest potrzebnych, jeśli liczba $l > 2$? Z własności ścieżek Eulera wiemy, że $k \geqslant \frac{l}{2}$. Czy $\frac{l}{2}$ jest wystarczającą liczbą linii autobusowych?
- Faktycznie, $k = \frac{l}{2}$. Jeśli bowiem do grafu dodamy specjalny wierzchołek v, który połączymy krawędzią z każdym wierzchołkiem o nieparzystym stopniu, to uzyskany graf będzie posiadał cykl Eulera. Po jego wyznaczeniu, możemy rozdzielić go na zbiór ścieżek poprzez usunięcie wszystkich wystąpień wierzchołka v. Spowoduje to powstanie $\frac{l}{2}$ ścieżek, które przechodzą przez wszystkie krawędzie oryginalnego grafu, a co za tym idzie, stanowią poszukiwany zbiór linii autobusowych.

10.1.7. Przemytnicy

- Sugerowana złożoność czasowa rozwiązania to $O(m * log(n))$.
- Przemytnicy muszą przewieźć przez granicę złoto pod postacią pewnego metalu. Aby wyznaczyć jaki wybór jest najbardziej opłacalny, dla każdego z metali należy wyznaczyć sumę: kosztu zamiany jednego kilograma złota w dany metal, cła dla jednego kilograma tego kruszcu oraz kosztu zamiany z powrotem w złoto. Po wyznaczeniu wszystkich tych wartości, wystarczy wybrać najmniejszą z nich.

- Aby wyznaczyć koszt zamiany 1 kg złota w określony metal, można skonstruować graf reprezentujący koszty zamiany metali, w którym wierzchołkami są metale, a krawędzie reprezentują proces alchemii (każda krawędź ma przypisaną wagę, równą kosztowi przeprowadzenia zamiany 1 kg jednego metalu na drugi). Na tak skonstruowanym grafie wystarczy wykonać algorytm Dijkstry, rozpoczynając z wierzchołka reprezentującego złoto; odległości poszczególnych wierzchołków od źródła reprezentują koszty uzyskania tych metali ze złota. Przy użyciu jednego wywołania algorytmu Dijkstry, uzyskujemy koszty otrzymania wszystkich metali ze złota.
- Aby wyznaczyć koszty uzyskania złota z poszczególnych metali, można zastosować podobną metodę jak w poprzednim punkcie, lecz tym razem algorytm Dijkstry należy wykonać na grafie transponowanym (w którym krawędzie reprezentujące poszczególne przemiany mają przeciwny zwrot niż w grafie oryginalnym).

10.1.8. Skoczki

- Sugerowana złożoność czasowa rozwiązania to $O(n^3)$.
- Szachownica składa się z dwóch rodzajów pól — białych oraz czarnych. Skoczek wykonując ruch zawsze przemieszcza się między polami o różnych kolorach.
- Graf, którego wierzchołki reprezentują pola szachownicy, a krawędzie dozwolone ruchy skoczków, jest dwudzielny. Problem wyznaczenia maksymalnego, nie bijącego się zbioru skoczków jest równoważny wyznaczeniu maksymalnego zbioru wierzchołków niepołączonych krawędziami — tzw. maksymalnego zbioru niezależnych wierzchołków.
- W przypadku grafów dwudzielnych, liczność maksymalnego zbioru niezależnych wierzchołków jest równa liczbie wierzchołków w tym grafie, pomniejszonej o wielkość maksymalnego skojarzenia. Rozwiązanie zadania jest już proste — można zastosować do tego algorytm Hopcrofta-Karpa.

10.2. Geometria obliczeniowa na płaszczyźnie

10.2.1. Akcja komandosów

- Sugerowana złożoność czasowa rozwiązania to $O(n^2)$.
- Każdemu komandosowi przypisane jest pewne koło; jeśli bomba znajduje się w obrębie tego obszaru, to podczas jej wybuchu komandos zginie. Dla danego zbioru komandosów istnieje możliwość takiego umieszczenia bomby, aby żaden z komandosów nie przeżył jej wybuchu, o ile przecięcie wszystkich odpowiadających kół jest niepuste. Zatem rozwiązanie zadania polega na dodawaniu kolejnych kół reprezentujących obszary rażenia dla komandosów tak długo, dopóki ich część wspólna jest niepusta. W jaki sposób można sprawdzić, czy część wspólna k kół jest niepusta w czasie $O(k)$?
- Brzeg obszaru stanowiącego część wspólną k kół jest wyznaczony przez obwody tych kół. Czy można wyznaczyć pewien punkt charakterystyczny c, należący do obwodu jednego z kół, który należy do części wspólnej wszystkich kół o ile jest ona niepusta?
- Dla danych k kół, z których każde dwa mają niepustą część wspólną, niech X będzie zbiorem $\frac{k*(k-1)}{2}$ punktów stanowiących skrajnie-prawe punkty części wspólnych tych kół. Przecięcie wszystkich k kół jest niepuste wtw gdy do wszystkich k kół należy

najbardziej lewy punkt zbioru X — poszukiwany punkt c. Przy dowodzeniu tego faktu pomocne może okazać się narysowanie przykładowego układu kół. Teraz już łatwo jest napisać program, który dokłada kolejne koła oraz aktualizuje odpowiednio punkt c.

10.2.2. Pomniki

- Sugerowana złożoność czasowa rozwiązania to $O((n+m)*log(n))$.
- Warunek, aby wybudowany pomnik nie zakłócał komunikacji między serwerami jest równoważny temu, że jego lokalizacja nie znajduje się wewnątrz wypukłej otoczki wyznaczonej przez lokalizacje Bito-serwerów.
- Sprawdzenie, czy dany punkt należy do wnętrza wielokąta wypukłego można wykonać w czasie logarytmicznym, przy użyciu funkcji `PointInsideConvexPol`.
- Rozwiązanie zadania sprowadza się do wyznaczenia wypukłej otoczki dla zbioru lokalizacji Bito-serwerów, a następnie dla każdej proponowanej lokalizacji pomnika sprawdzenie, czy leży ona wewnątrz wyznaczonej wypukłej otoczki.

10.2.3. Ołtarze

- Istnieje możliwość rozwiązania tego zadania w złożoności $O(n^2)$, jednak rozwiązanie takie jest dość skomplikowane. Sugerowana złożoność programu to $O(n^2 * log(n))$.
- Sprawdzanie, czy dana świątynia może zostać zhańbiona, można wykonywać niezależnie dla każdej z nich. W ten sposób należy przeprowadzić n testów, z których każdy należy zrealizować w złożoności $O(n*log(n))$.
- Aby sprawdzić, czy dana świątynia s może zostać zhańbiona, można zastosować metodę zamiatania kątowego (należy posortować świątynie widziane przez wejście do świątyni s po kącie odchylenia względem pozycji ołtarza świątyni s, a następnie sprawdzić, czy „patrząc" z pozycji ołtarza istnieje prześwit, przez który mógłby się przemknąć duch).
- Podczas sprawdzania czy istnieje prześwit, każdą świątynię można reprezentować przy pomocy pary jej przekątnych. Lewe końce wszystkich odcinków, widocznych przez wejście do świątyni s z ołtarza, można umieścić w jednej kolejce, a prawe końce w drugiej. Po wykonaniu sortowania kątowego należy przetworzyć obie kolejki, za każdym razem wybierając z jednej z nich punkt, który znajduje się bardziej na lewo. Jeśli w pewnym momencie liczba przetworzonych lewych końców jest równa liczbie przetworzonych prawych końców, to oznacza to, że znaleziony został prześwit.

10.3. Kombinatoryka

10.3.1. Liczby permutacyjnie-pierwsze

- Sprawdzenie, czy dana liczba jest permutacyjnie-pierwsza, nie jest zadaniem prostym. Wymaga ono wygenerowania wszystkich permutacji cyfr danej liczby, a następnie sprawdzenia, czy któraś z nich zachowuje własność podaną w zadaniu.
- Aby rozwiązać jeden podproblem, należy sprawdzić dla każdej liczby z zadanego przedziału, czy jest ona permutacyjnie-pierwsza. W tym celu potrzebna jest również możliwość szybkiego sprawdzania, czy dana liczba jest pierwsza. Można tego dokonać poprzez

wygenerowanie listy liczb pierwszych za pomocą funkcji `bitvector Sieve()`. Sumaryczny czas działania programu to $O(s * t)$, gdzie s to liczba przeprowadzonych testów $(q - p)$, natomiast t określa złożoność wykonania pojedynczego testu. W jakim czasie można wykonać pojedynczy test?

- Stosując generator permutacji z użyciem minimalnej liczby transpozycji oraz wykorzystując tablicę liczb pierwszych, pojedynczy test można wykonać w czasie $O(k!)$, gdzie k jest liczbą cyfr testowanej liczby. Po wykonaniu zamiany miejscami dwóch cyfr liczby w łatwy sposób można zaktualizować jej wartość, a następnie sprawdzić, czy różnica między nią a początkową liczbą jest postaci $9 * p$.

10.4. Teoria liczb

10.4.1. Bilard

- Sugerowana złożoność rozwiązania to $O(log(n))$, gdzie n to ograniczenie na wielkość współrzędnych z zadania.

- Przy rozwiązaniu zadania pomocne jest rozważenie „odbicia" kuli od boku stołu bilardowego jak w lustrze (po odbiciu obserwowany jest obraz kuli w lustrze). W ten sposób kula nie odbija się od ścian stołu, lecz porusza się cały czas wzdłuż linii prostej. Stół można utożsamić z siatką, którą kula przecina podczas odbijania się od ścian stołu. Kula w oryginalnym problemie wpadnie do łuzy, jeśli w problemie zmodyfikowanym znajdzie się w punkcie kratowym siatki. W jaki sposób w zmodyfikowanym zadaniu wyznaczyć czy i do której łuzy wpadnie kula?

- Wyznaczenia pierwszego punktu kratowego, przez który przetoczy się kula, można dokonać obliczając najpierw współrzędne kuli w chwili pierwszego przecięcia pionowej linii siatki, a następnie przeskalowując te współrzędne w taki sposób, aby współrzędna y-kowa również znajdowała się na poziomej linii siatki. Aby stwierdzić, do której łuzy wpadła kula, wystarczy przeanalizować liczbę przecięć pionowych oraz poziomych linii siatki.

10.4.2. Wyliczanka

- Zadanie można w prosty sposób rozwiązać w złożoności $O(n^2)$.

- Jeśli w pewnym momencie w kółku stoi k dzieci oraz między dziećmi, które kolejno odpadają z kółka, znajduje się $m - 1$ innych dzieci, to długość wyliczanki musi być postaci $l * k + m$, $l \in N$ (podczas wyliczania m dzieci wypowie o jedną sylabę więcej niż reszta $k - m$ dzieci stojących w kółku).

- Rozpatrując kolejne odchodzące z kółka dzieci, można skonstruować układ n równań modularnych postaci $n \equiv m_l \pmod{k_l}$. Jeśli układ ten nie jest sprzeczny, to najmniejsze jego rozwiązanie stanowi poszukiwaną długość wyliczanki.

- Wyznaczenie rozwiązania układu równań modularnych jest możliwe przy użyciu funkcji `congr`. Równania można rozwiązywać parami, w każdym kroku eliminując jeden element układu kongruencji. Ostatni uzyskany wynik (jeśli istnieje) stanowi długość najkrótszej możliwej wyliczanki.

10.4.3. Łańcuch

- Istnieje możliwość rozwiązania tego zadania w czasie $O(n * log(n) * log(log(n)))$, jednak sugerowana złożoność rozwiązania to $O(n^2)$.

- Zauważmy, że jeśli sekwencję ruchów l prowadzącą do zdjęcia łańcucha z pręta wykonamy w odwróconej kolejności, rozpoczynając od łańcucha zdjętego z pręta, to uzyskamy w ten sposób sytuację początkową. Zatem zamiast rozpatrywać problem zdejmowania łańcucha z pręta, możemy przeanalizować zakładanie go w taki sposób, aby uzyskać zadaną konfigurację końcową.

- Zgodnie z opisem dozwolonych ruchów, minimalna liczba kroków, które trzeba wykonać, aby założyć k-te ogniwo łańcucha na pręt jest niezależna od konfiguracji ogniw $k + 1, k + 2, \ldots$. Dodatkowo, w chwili gdy wykonywany jest ruch nakładania k-tego ogniwa na pręt, konfiguracja ogniw $k - 1, k - 2, \ldots$ jest zdefiniowana jednoznacznie — ogniwo $k - 1$ musi być założone na pręt, a ogniwa $k - 2, k - 3, \ldots$ — zdjęte. Ile ruchów należy wykonać, aby założyć k-te ogniwo przy założeniu, że wszystkie wcześniejsze są zdjęte z pręta?

- Założenie k-tego ogniwa wymaga 2^{k-1} ruchów. Ze względu na opisane wcześniej własności ruchów związanych z zakładaniem i zdejmowaniem ogniw z pręta, wyznaczenie liczby potrzebnych ruchów do zdjęcia całego łańcucha można dokonać, sumując liczby ruchów potrzebnych do zdjęcia kolejnych ogniw (zaczynając od ogniwa numer n). Zdejmując ogniwo na pozycji k, należy pamiętać o zmianie stanu ogniwa $k - 1$. Do przechowywania wyniku należy wykorzystać arytmetykę wielkich liczb.

10.5. Struktury danych

10.5.1. Małpki

- Sugerowana złożoność rozwiązania zadania to $O(n * log^*(n))$.

- Sposób trzymania się małpek między sobą można reprezentować przy użyciu grafu, w którym wierzchołkami są małpki, natomiast krawędź łącząca dwa wierzchołki v oraz w reprezentuje trzymanie się odpowiednich małpek. Puszczenie małpki w przez małpkę v można reprezentować przez usunięcie krawędzi (v, w) z grafu. Takie podejście po usunięciu każdej krawędzi wymaga zweryfikowania, które małpki spadły na ziemię. Proste rozwiązanie tego problemu polega na wyszukiwaniu wszystkich wierzchołków ze spójnej składowej wierzchołka 1 i daje w konsekwencji algorytm o złożoności $O(n^2)$. W jaki sposób można podejście to przyspieszyć?

- Zamiast symulować zdarzenia w kolejności ich wystąpienia, można wykonywać symulację w odwróconej kolejności — w ten sposób operacja usuwania krawędzi z grafu zostaje zastąpiona przez operację wstawiania krawędzi. Zmiana podejścia w istotny sposób ułatwia proces sprawdzania, które wierzchołki zostały dodane do spójnej składowej wierzchołka 1.

- Wykorzystując strukturę `FAU` można w prosty sposób reprezentować proces łączenia grup małpek. Cała symulacja wymaga $O(n)$ operacji `Find` oraz operacji `Union`, co w konsekwencji daje poszukiwaną złożoność algorytmu.

10.5.2. Kodowanie permutacji

- Oczekiwana złożoność czasowa rozwiązania to $O(n * log(n))$.
- Liczba b_k reprezentuje liczbę elementów permutacji ze zbioru $\{a_1, a_2, \ldots, a_{k-1}\}$ większych od elementu a_k. Sekwencja nie jest kodem żadnej permutacji, jeśli zachodzi $b_k \geqslant k, k \in \{1, 2, \ldots, n\}$. Jeśli sytuacja taka nie zachodzi, to kod permutacji jednoznacznie reprezentuje pewną permutację. W jaki sposób można wyznaczyć jej elementy?
- Dla danego kodu B permutacji A o liczbie elementów n istnieje możliwość wyznaczenia n-tego elementu permutacji — jest on po prostu równy $n - b_n$. Stosując tę technikę oraz uwzględniając już wykorzystane elementy permutacji, można wyznaczać kolejne elementy — $a_{n-1}, a_{n-2}, \ldots, a_1$. W jaki sposób można wyszukiwać k-ty najmniejszy element w danym zbiorze niewykorzystanych elementów?
- Korzystając ze struktury danych `PosTree`, wyznaczanie kolejnych elementów permutacji oraz aktualizowanie struktury danych można wykonać w czasie logarytmicznym, co w sumie daje algorytm o złożoności $O(n * log(n))$.

10.5.3. Marsjańskie mapy

- Oczekiwana złożoność czasowa rozwiązania to $O(n * log(n))$.
- Wyliczenie sumarycznego obszaru pokrytego przez wszystkie mapy można wykonać stosując technikę zamiatania. Mapy można utożsamić z prostokątami o bokach równoległych do osi układu współrzędnych. Jedna z metod rozwiązania zadania przetwarza pionowe krańce (początki oraz końce) kolejnych map w kolejności niemalejących odciętych. W każdym kroku należy aktualizować długość przekroju w płaszczyźnie y, która jest pokryta przez aktywne mapy (takie których lewe końce zostały już przetworzone, ale prawe jeszcze nie) oraz zwiększać odpowiednio wartość wyniku. W jaki sposób należy przechowywać informacje o aktywnych mapach, aby można było szybko uzyskiwać informację o wielkości pokrytego przekroju?
- Strukturą danych, którą można wykorzystać jest `CoverBTree`. Pozwala ona w czasie logarytmicznym na dodawanie oraz usuwanie kolejnych pionowych krańców map oraz wyznaczanie wielkości przekroju.

10.5.4. Puste prostopadłościany

- Oczekiwana złożoność czasowa rozwiązania to $O(n * log(n))$.
- Poszukiwany w zadaniu prostopadłościan musi mieć największą możliwą objętość. Pociąga to za sobą fakt, iż każda z jego trzech ścian, których pozycją można manewrować, muszą albo być umieszczone w odległości 10^6 od punktu $(0, 0, 0)$, albo muszą zawierać jeden z punktów ze zbioru A. Wyszukiwanie takiego prostopadłościanu można usystematyzować, zakładając przykładowo, że jego górna ściana (równoległa do płaszczyzny OXY) zawiera kolejne punkty ze zbioru A. Dla każdego przetwarzanego punktu należy maksymalizować powierzchnię podstawy prostopadłościanu tak, aby nie zawierał on w swym wnętrzu żadnego punktu ze zbioru A.
- Jeśli wysokość prostopadłościanu jest ustalona, to wyznaczenie maksymalnej powierzchni podstawy można wykonać w podobny sposób — poprzez wybieranie kolejnych punktów leżących w ścianie prostopadłościanu równoległej do osi OYZ oraz dobieranie

odpowiedniego punktu dla ściany równoległej do OXZ. Takie rozwiązanie jednak ma złożoność $O(n^3)$.

Na szczęście istnieje możliwość przyspieszenia sposobu wyznaczania ograniczających punktów. Zauważmy, że przy ustalonej wysokości prostopadłościanu interesujące są tylko te punkty, które mają mniejszą wysokość (współrzędną z) od wysokości prostopadłościanu. Wszystkie takie punkty można zrzutować na płaszczyznę OXY, a następnie wyznaczać na tej płaszczyźnie podstawę prostopadłościanu, nie zawierającą żadnego ze zrzutowanych punktów. Punkty interesujące podczas wyznaczania podstawy reprezentują „schodki" prowadzące od punktu na dodatniej półosi OY $(0, 1\,000\,000)$ do punktu na dodatniej półosi OX $(1\,000\,000, 0)$. Powierzchnia największej podstawy prostopadłościanu jest równa największemu iloczynowi współrzędnej y pewnego punktu ze współrzędną x kolejnego punktu wyznaczających „schodki". Rozwiązanie takie daje algorytm o złożoności czasowej $O(n^2)$.

- Punkty ze zbioru A można przetwarzać w kolejności niemalejących współrzędnych z. Dla kolejnego punktu p należy ustalić wysokość prostopadłościanu oraz wybrać największą powierzchnię podstawy bazując na aktualnej zawartości „schodków". Następnie należy zaktualizować postać schodków poprzez dodanie punktu p. Do reprezentacji schodów można wykorzystać odpowiednio wzbogaconą strukturę drzewa BST, która będzie pozwalała na szybkie wyznaczanie powierzchni podstawy, co daje w konsekwencji algorytm o złożoności $O(n * log(n))$.

- Istnieje możliwość rozwiązania tego zadania w złożoności $O(n * log(n))$ bez wykorzystywania wzbogaconych drzew `BST` — wystarczające okazują się struktury danych znajdujące się w bibliotece STL. Algorytm taki jest zaprezentowany w programie stanowiącym rozwiązanie zadania, znajdującym się na dołączonej płycie.

10.6. Algorytmy tekstowe

10.6.1. Szablon

- Rozwiązanie zadania można zaimplementować w złożoności czasowej $O(n)$, jednak wystarczająca złożoność to $O(n*log(n))$. Na temat liniowej implementacji rozwiązania można przeczytać w [OI12].

- Szablon tekstu musi pokrywać w całości napis, dla którego jest wyznaczany. W szczególności, musi on pasować do początku, jak i do końca tekstu, co oznacza, że jest on zarówno prefiksem, jak i sufiksem tekstu. W jaki sposób można efektywnie wyznaczyć wszystkie prefikso-sufiksy danego tekstu w, celu zweryfikowania czy są one szablonami oraz wybrania najkrótszego z nich?

- Doskonała do wyznaczania wszystkich prefikso-sufiksów tekstu jest tablica prefiksowa wykorzystywana w algorytmie KMP. Dla każdego wyznaczonego przy jej użyciu prefikso-sufiksu (ich wyznaczenie zajmuje czas liniowy), można sprawdzić choćby przy użyciu algorytmu KMP — które z prefikso-sufiksów są szablonami tekstu, a następnie wybrać najkrótszy. Takie rozwiązanie jednak ma pesymistyczną złożoność $O(n^2)$ — dla tekstu postaci a^n istnieje $O(n)$ prefikso-sufiksów, a sprawdzenie każdego z nich zajmuje czas liniowy. Czy można w jakiś sposób zredukować liczbę sprawdzanych prefikso-sufiksów, aby poprawić złożoność rozwiązania?

- Jeśli dla tekstu t istnieją dwa prefikso-sufiksy a oraz b, dla których zachodzi $|a| \leqslant 2*|b|$, to jeśli a jest szablonem dla tekstu t, to b również nim jest. Jest tak dlatego, że b stanowi wówczas szablon a. Korzystając z tej własności, podczas sprawdzania kolejnych prefiksosufiksów tekstu t, można wiele z nich pominąć — każdy kolejny sprawdzany tekst będzie przynajmniej dwa razy krótszy od poprzedniego, co daje najwyżej $O(log(n))$ testów oraz złożoność końcową $O(n * log(n))$.

10.6.2. MegaCube

- Zadanie można rozwiązać w czasie $O(n)$, gdzie n reprezentuje powierzchnię labiryntu.

- Jeśli zwiedzony obszar labiryntu jest prostokątem o jednym z boków długości 1, to rozwiązanie zadania można zrealizować poprzez wyznaczenie wszystkich wystąpień wzorca reprezentującego zwiedzony obszar w tekście opisującym labirynt. Czy zawsze można dokonać redukcji problemu z zadania do takiej postaci?

- Sposób redukcji do problemu wyszukiwania wzorca w tekście może polegać na przyporządkowaniu każdej kolumnie wzorca pewnego identyfikatora oraz skonstruowaniu nowej reprezentacji labiryntu, w której wystąpienia kolumn wzorca zostały zastąpione ich identyfikatorami. W ten sposób, wyszukiwanie 2-wymiarowego wzorca w danym dwuwymiarowym tekście, zostało zredukowane do wyszukiwania zwykłego wzorca (składającego się z identyfikatorów kolumn), w zmodyfikowanym tekście. Jaką metodę zamiany reprezentacji należy zastosować, aby uzyskać algorytm liniowy ze względu na wielkość danych wejściowych?

- Wykorzystując algorytm wyszukiwania wielu wzorców w tekście (struktura `MKMP`), fazę redukcji można wykonać w czasie liniowym. Kolejne kolumny wzorca stanowią wyszukiwane wzorce w tekście reprezentowanym przez labirynt.

10.6.3. Palindromy

- Zadanie można rozwiązać w złożoności $O(n^2)$, gdzie n reprezentuje długość przetwarzanego tekstu.

- Zadanie można rozwiązać, sprowadzając je do problemu grafowego. Każdy prefiks tekstu t można reprezentować poprzez wierzchołek w grafie. Krawędź między dwoma wierzchołkami t_l oraz t_k istnieje wtedy, gdy podsłowo $[t_{l+1} \dots t_k]$ jest palindromem parzystym. Krawędzie w grafie można wyznaczać, stosując algorytm do wyliczania promieni palindromów w tekście (ze względu na końcową złożoność całego algorytmu, nie musi to być algorytm działający w czasie liniowym). Każdy rozkład słowa t na palindromy parzyste ma swój odpowiednik w grafie jako ścieżka zaczynająca się w wierzchołku $t_0 = \epsilon$ i kończąca się w $t_n = t$.

- Rozkład na minimalną liczbę palindromów odpowiada najkrótszej ścieżce między wierzchołkiem t_0 a t_n, natomiast maksymalny rozkład na palindromy — najdłuższej. W celu wyznaczenia tych ścieżek nie jest konieczne jawne konstruowanie grafu — wystarczy zastosować programowanie dynamiczne, polegające na wyznaczaniu dla kolejnych prefiksów t najkrótszej oraz najdłuższej ścieżki o początku w t_0.

10.6.4. Punkty

- Zadanie można rozwiązać w złożoności $O(n * k * log(k))$.

- Na wstępnie zakładamy, że porównywane zbiory punktów są tej samej mocy — jest to bowiem warunek konieczny, aby zbiory punktów były podobne. Zgodnie z treścią zadania, dwa zbiory punktów są do siebie podobne, jeśli można uzyskać jeden z drugiego poprzez wykonanie serii operacji: obrotów, przesunięć, symetrii osiowych i jednokładności. Sprawdzenie podobieństwa dwóch zbiorów punktów może polegać na próbie wykonania serii przekształceń jednego ze zbiorów, mających na celu uzyskanie drugiego zbioru. W jaki sposób można zrealizować to zadanie?

- Operacje, jakie można wykonywać na zbiorze punktów są przemienne w tym sensie, że jeśli zbiór A możemy przekształcić na zbiór B, przy użyciu sekwencji przekształceń $l_1, l_2, l_3, \ldots$, to istnieje również sekwencja przekształceń $k_1, k_2, k_3, \ldots$, takich, że na początku wykonywane są tylko obroty, następnie przesunięcia, symetrie osiowe a na końcu jednokładności. W jaki sposób można skonstruować (o ile istnieje) ciąg takich przekształceń?

- Wyeliminowanie operacji przesunięcia można dokonać poprzez wyznaczenie dla obu zbiorów ich środków ciężkości, a następnie zrealizować takie przesunięcie, aby środki te znalazły się w punkcie $(0, 0)$. Takie podejście powoduje, że nie zachodzi potrzeba wykonywania innych symetrii osiowych niż o środku w punkcie $(0, 0)$. Późniejsze wykonywanie obrotów, jednokładności oraz symetrii osiowych o środku w punkcie $(0, 0)$ nie zmienia położenia środka ciężkości zbioru punktów, zatem znika potrzeba dalszego wykonywania operacji przesunięć. W jaki sposób wyeliminować kolejne operacje?

- Eliminacja jednokładności może zostać wykonana poprzez odpowiednie przeskalowanie obu zbiorów punktów względem punktu $(0, 0)$ tak, aby ich wielkości były równe (wielkość możemy rozumieć jako odległość najdalszego punktu od $(0, 0)$). Po wykonaniu takiego przeskalowania, operacje obrotów oraz symetrii osiowych nie powodują zmiany wielkości zbioru.

- Przy założeniu, że nie musimy wykonywać symetrii osiowej, sprawdzenie, czy dwa zbiory punktów są identyczne z dokładnością do obrotu, można wykonać sortując oba zbiory punktów po kątach oraz wyznaczając dla posortowanych sekwencji punktów ich reprezentacje tekstowe (przykładowo poprzed zastąpienie każdego punktu parą liczb odległość od punktu $(0, 0)$ oraz odległość od poprzedniego punktu w sekwencji). Dwa zbiory punktów są identyczne, jeśli wyznaczone obwódki są równoważne cyklicznie. Problem symetrii osiowej można łatwo wyeliminować, zamieniając kolejność jednej z obwódek oraz sprawdzając po raz kolejny równoważność cykliczną.

10.7. Algebra liniowa

10.7.1. Taniec

- Zadanie można rozwiązać w czasie $O((x * y)^3)$.

- Nastąpienie na każde pole szachownicy powoduje zmianę stanu świecenia tego pola oraz pól sąsiednich, dwukrotne nastąpienie na to samo pole przywraca poprzedni stan podświetlenia, zatem na każde pole szachownicy opłaca się nastąpić co najwyżej raz.

Oznacza to, że rozwiązanie zadania sprowadza się do wyznaczenia, dla każdego pola, czy należy na nie nastąpić.

- Stan podświetlenia każdego pola jest również „binarny" — zapalone bądź nie. Można się zatem pokusić o sprowadzenie zadania do rozwiązania odpowiedniego układu równań w ciele Z_2. Jak powinien wyglądać taki układ równań?

- Z każdym polem szachownicy należy związać niewiadomą, która przyjmuje wartość 0, jeśli na pole nie należy nastąpić oraz wartość 1 w przeciwnym razie. Takie podejście daje $x * y$ niewiadomych. Podobnie, dla każdego pola, należy stworzyć równanie postaci $x + x_1 + \ldots + x_k = c$, gdzie $x_1, x_2, \ldots, x_k$ są niewiadomymi reprezentującymi sąsiednie pola x, natomiast c równa jest 1, jeśli pole x nie jest na początku zapalone, a 0 w przeciwnym przypadku. Powstaje w ten sposób układ $x*y$ równań z $x*y$ niewiadomymi, który można rozwiązać przy pomocy funkcji `GaussZ2`.

10.7.2. Sejf

- Zadanie można rozwiązać w czasie $O(n^3)$.

- Idea rozwiązania tego zadania jest bardzo zbliżona do poprzedniego zadania *Taniec*. Jedyna różnica polega na tym, że wartości wszystkich współczynników oraz niewiadomych są z przedziału $\{0, 1, p-1\}$, a nie jak poprzednio $\{0, 1\}$.

10.7.3. Szalony malarz

- Zadanie można rozwiązać przy użyciu programowania liniowego.

- Każdemu możliwemu krokowi można przypisać niewiadomą x_i, $i \in \{1, 2, \ldots, k\}$, która będzie przyjmowała wartość 1, jeśli krok ten jest wykonywany oraz wartość 0, w przeciwnym przypadku. Każdej niewiadomej x_i można przypisać wagę a_i, równą czasowi wykonania odpowiedniego ruchu malowania obrazu. Wyznaczenie listy ruchów pozwalających na pomalowanie obrazu w najkrótszym możliwym czasie, jest równoważne zminimalizowaniu wartości funkcji $a_1 * x_1 + a_2 * x_2 + \ldots + a_k * x_k$. W jaki sposób można wyznaczyć możliwe ruchy malowania oraz jak zapewnić, że każdy obszar obrazu zostanie pomalowany?

- Jeśli w optymalnym malowaniu obrazu wykonywany jest ruch pionowy bądź poziomy, który nie pokrywa wszystkich sąsiadujących pól obrazu o tym samym kolorze, to ruch taki można wydłużyć nie zmieniając jednocześnie czasu malowania obrazu. Zatem liczba różnych ruchów które mogą występować w optymalnym malowaniu obrazu jest ograniczona przez $3*m*n$. Wszystkie te ruchy należy uwzględnić w warunkach dla programowania liniowego.

- W celu zapewnienia aby każdy cal kwadratowy obrazu został pomalowany, należy dla każdego z nich sformułować warunek mówiący, że suma zmiennych reprezentujących ruchy malowania obrazu, które zawierają ten obszar, jest nie mniejsza od 1. Ze względu na charakter wszystkich skonstruowanych w ten sposób warunków, wartości wyznaczonych zmiennych przez programowanie liniowe będą należeć do zbioru $\{0, 1\}$. O innych sposobach stosowania programowania liniowego można przeczytać w [PCC]. Po analizie tej lektury, wykazanie poprawności przedstawionego rozwiązania powinno być już bardzo proste.

Dodatek A

Nagłówki Eryka Kopczyńskiego

Listing A.1: Nagłówki Eryka Kopczyńskiego na konkurs TopCoder

```
001 #include <algorithm>
002 #include <string>
003 #include <vector>
004 #include <ctype.h>
005 #include <math.h>
006 #include <iostream>
007 #include <set>
008 #include <map>
009 #include <complex>
010 #include <sstream>
011
012 using namespace std;
013
014 typedef long long ll;
015 typedef long double ld;
016 typedef vector<int> vi;
017 typedef vector<int> vll;
018 typedef vector<string> vs;
019 typedef complex<ll> cll;
020 typedef complex<ld> cld;
021 #define Size(x) (int(x.size()))
022 #define LET(k,val) typeof(val) k = (val)
023 #define CLC(act,val) (*({act; static typeof(val) CLCR; CLCR = (val); &CLCR;}))
024 #define FOR(k,a,b) for(typeof(a) k=(a); k < (b); ++k)
025 #define FORREV(k,a,b) for(typeof(b) k=(b); (a) <= (--k);)
026 #define FIRST(k,a,b,cond) CLC(LET(k, a); for(; k < (b); ++k) if(cond) break, k)
027 #define LAST(k,a,b,con) CLC(LET(k, b); while((a) <= (--k)) if(con) break, k)
028 #define EXISTS(k,a,b,cond) (FIRST(k,a,b,cond) < (b))
029 #define FORALL(k,a,b,cond) (!EXISTS(k,a,b,!(cond)))
030 #define FOLD0(k,a,b,init,act) CLC(LET(k, a); LET(R##k, init); \
031     for(; k < (b); ++k) {act;}, R##k)
032 #define SUMTO(k,a,b,init,x) FOLD0(k,a,b,init,R##k += (x))
033 #define SUM(k,a,b,x) SUMTO(k,a,b,(typeof(x)) (0), x)
034 #define PRODTO(k,a,b,init,x) FOLD0(k,a,b,init,R##k *= (x))
```

Listing A.1: (c.d. listingu z poprzedniej strony)

```
035 #define PROD(k,a,b,x) PRODTO(k,a,b,(typeof(x)) (1), x)
036 #define MAXTO(k,a,b,init,x) FOLD0(k,a,b,init,R##k >?= (x))
037 #define MINTO(k,a,b,init,x) FOLD0(k,a,b,init,R##k <?= (x))
038 #define QXOR(k,a,b,x) FOLD0(k,a,b,(typeof(x)) (0), R##k ^= x)
039 #define QAND(k,a,b,x) FOLD0(k,a,b,(typeof(x)) (-1), R##k &= x)
040 #define QOR(k,a,b,x) FOLD0(k,a,b,(typeof(x)) (-1), R##k |= x)
041 #define FOLD1(k,a,b,init,act) CLC(LET(k, a); LET(R##k, init); \
042     for(++k; k < (b); ++k) act, R##k)
043 #define MAX(k,a,b,x) FOLD1(k,a,b,x, R##k >?= (x))
044 #define MIN(k,a,b,x) FOLD1(k,a,b,x, R##k <?= (x))
045 #define FIRSTMIN(k,a,b,x) (MIN(k,a,b,make_pair(x,k)).second)
046 #define INF 1000000000
047
048 int tcize(int n) {return n<INF ? n : -1;}
049
050 vi parsevi(string s) {
051     s = s + " ";
052     int q = 0;
053     vi res;
054     FOR(l,0, Size(s)) {
055         if(s[l] == ' ') { res.push_back(q); q = 0;}
056         else { q = q * 10 + s[l] - '0'; }
057     }
058     return res;
059 }
060
061 vs parsevs(string s) {
062     s = s + " ";
063     string q = "";
064     vs res;
065     FOR(l,0, Size(s)) {
066         if(s[l] == ' ') { res.push_back(q); q = "";}
067         else { q += s[l]; }
068     }
069     return res;
070 }
071
072 #define MKey(x) \
073     typedef typeof(memo) tmemo; tmemo::key_type key = (x); \
074     if(memo.find(key) != memo.end()) return memo[key]
075
076 #define MRet(y) return (memo[key] = y)
077
078 template <class T> T operator | (complex<T> x, complex<T> y) {
079     return (x*conj(y)).real();
080 }
081
082 template <class T> T operator ^ (complex<T> x, complex<T> y) {
```

Listing A.1: (c.d. listingu z poprzedniej strony)

```
083     return (x*conj(y)).imag();
084 }
085
086 int bitc(ll r) {return r == 0 ? 0 : (bitc(r>>1) + (r&1));}
087
088 ll gcd(ll x, ll y) {return x ? gcd(y%x,x) : y;}
089
090 template<class T> T& operator >?= (T& x, T y) {if(y>x) x=y; return x;}
091 template<class T> T& operator <?= (T& x, T y) {if(y<x) x=y; return x;}
092 template<class T> T operator >? (T x, T y) {return x>y?x:y;}
093 template<class T> T operator <? (T x, T y) {return x<y?x:y;}
094
095 #define Pa(xy) ((xy).first)
096 #define Ir(xy) ((xy).second)
097
098 string cts(char c) {string s=""; s+=c; return s;}
099
100 template<class T> ostream& operator<<(ostream& os, const vector<T>& v) {
101     os << "{";
102     for(LET(k,v.begin()); k != v.end(); ++k) {os << (*k); os << ",";}
103     os << "}";
104     return os;
105 }
106
107 template<class T, class U> ostream& operator<<(ostream& os, const pair<T,U>&
p) {
108     return os << "(" << p.first << "," << p.second << ")";
109 }
110
111 template<class T> ostream& operator<<(ostream& os, const set<T>& v) {
112     os << "{";
113     for(LET(k,v.begin()); k != v.end(); ++k) {os << (*k); os << ",";}
114     os << "}";
115     return os;
116 }
117
118 #ifdef floyd
119 FOR(k,0,100) FOR(i,0,100) FOR(j,0,100) w0[i][j] <?= w0[i][k] + w0[k][j];
120 #endif
121
122 #define BINFIRST(k,a,b,cond) \
123 CLC( \
124 LET(k##mIn, a); LET(k##mAx, b); \
125 while(k##mIn != k##mAx) { \
126 LET(k, (k##mIn+k##mAx)>>1); \
127 if(cond) k##mAx = k; \
128 else k##mIn = k+1; \
129 }, \
```

Listing A.1: (c.d. listingu z poprzedniej strony)

```
130 k##mIn \
131 )
```

Dodatek B

Sposoby na sukces w zawodach

W październiku 2004 roku, po eliminacjach do zawodów drużynowych ACM ICPC na Uniwersytecie Warszawskim, nastąpiły istotne zmiany w składzie drużyn związane, w głównym stopniu, z odejściem zawodników z lat ubiegłych. Z uwagi na odnoszone sukcesy poprzedników, prof. Krzysztof Diks zwrócił się do nich z prośbą o udzielenie młodszym kolegom wskazówek związanych ze strategią uczestnictwa i osiągania dobrych wyników w konkursach. W omawianym dodatku zostały zebrane wskazówki oraz dobre rady udzielone przez Tomka Czajkę, Andrzeja Gąsienicę-Samka, Tomasza Malesińskiego, Krzysztofa Onaka oraz Marcina Stefaniaka. Zgromadzone uwagi i sugestie dotyczą przede wszystkim konkursów zespołowych takich jak ACM ICPC, ale wiele z nich jest tak samo pożytecznych w przypadku innych konkursów. Bez wątpienia, będą one przydatne dla wielu nie tylko początkujących zawodników. Oto co mówią laureaci konkursów informatycznych:

Tomasz Czajka, trzykrotny zwycięzca w konkursie organizowanym przez TopCoder, mistrz Świata w Programowaniu Zespołowym 2003.

- Należy trenować indywidualnie na *http://acm.uva.es/*. To najważniejsze.
- Startować wspólnie z kolegami z drużyny w konkursach na *http://acm.uva.es/contest/*
- „Coś", co zmieniło naszą drużynę nie do poznania: Osoba, która zrobiła najmniej zadań w konkursie, stawiała pozostałym obiad. Jest to banalne ale bardzo ważne, gdyż bez tego nie ma szans traktować konkursu poważnie i z pełną koncentracją. Zrobione zadania trzeba liczyć tak jak strzelone bramki w piłce nożnej. Inaczej nikt nie weźmie na siebie odpowiedzialności robienia trudnego zadania, licząc na innych. Dopuszczalna jest nawet sytuacja, gdzie dwie osoby konkurują nad zrobieniem tego samego, przeważnie ostatniego zadania konkursowego. Zdarzało się to nam nie raz i zawsze miało pozytywne skutki.
- Organizować sobie symulowane konkursy z zadaniami z byłych ACM-ów (zadania i testy można znaleźć na stronie *http://icpc.baylor.edu/*). Należy zrobić sobie skrypt do testowania rozwiązań, który odpowiada 'OK', 'Zła odpowiedź' ... Po zakończeniu treningu należy przeprowadzić porównanie do innych drużyn, które w danym konkursie startowały.
- Zasada stawiania obiadów odnosi się również do treningów.
- Należy omawiać w obrębie zespołu różne strategie, modyfikować je zależnie od doświadczeń. Możliwe są różne podejścia, ale generalnie najlepsza jest strategia — dzielimy

się losowo zadaniami, każdy rozwiązuje jedno na kartce, implementuje na komputerze, wysyła, rozwiązuje następne, implementuje i tak dalej. Ewentualnie można odkładać zadania, które są rodzaju niezbyt lubianego przez siebie, na specjalnie wydzielony „wspólny stosik zadań na później".

- Należy „wywalać" zaakceptowane zadania pod stół i zapominać o nich do końca konkursu.
- Należy pisać całe rozwiązania zadań na kartce. Komputera powinno używać się tylko do wpisania gotowego rozwiązania i sprawdzenia czy działa. Jeśli działa — wysyłamy, jak nie działa — drukujemy i zwalniamy komputer. Jeśli sprawdzanie rozwiązania na serwerze trwa zbyt długo, to w razie czego należy wydrukować program, aby o ile okaże się, że rozwiązanie jest złe, nie blokować ponownie komputera
- Podczas szukania błędu w rozwiązaniu, czytamy cały wydruk linijka po linijce, zaznaczając wszystkie znalezione błędy. Po 30 - 60 minutach program na kartce powinien być bezbłędny. Siadamy wtedy do komputera i poprawiamy błędy. Jeśli okaże się, że program dalej nie działa, to znowu drukujemy i zwalniamy komputer.
- Nie wolno równocześnie zajmować komputera i myśleć. Jeśli okaże się (nie powinno), że zaczęliśmy wpisywać program i nie do końca wiemy co dalej, to drukujemy, zwalniamy komputer, dopracowujemy szczegóły na kartce i czekamy na swoją kolejkę dostępu do komputera w celu zaimplementowania brakującej części.
- Nie wolno debugować przy komputerze. Źle działający program trzeba czytać na wydruku do skutku, to znaczy do znalezienia błędu. Do tego trzeba się przyzwyczaić.

  Ewentualny wyjątek: w pewnym momencie pod koniec konkursu możemy podjąć decyzję, że pracujemy już tylko nad jednym zadaniem. Inne definitywnie odkładamy. Dopiero wtedy można debugować. Zostawia się komputer piszącemu rozwiązanie do pełnej dyspozycji, reszta drużyny idzie np. na colę, żeby mu nie przeszkadzać.
- Nikt nie jest przyzwyczajony do konkursowego trybu pracy, dlatego potrzebny jest wspólny udział w treningach.
- Po konkursie należy analizować jakie były błędy strategiczne — w szczególności czy ktoś marnował dużo czasu czekając na komputer.

Andrzej Gąsienica-Samek, Mistrz Świata w Programowaniu Zespołowym 2003

- Zasadniczym kryterium na dobrą drużynę jest zebranie trzech zawodników i rozdzielenie pomiędzy nich zadania do rozwiązania na samym początku zawodów (przykładowo pierwsza osoba otrzymuje zadania A, D, G, druga B, E, H a trzecia C, F) — każdy z nich stara się rozwiązać swoje zadania jak najszybciej.
- Dobrym pomysłem jest napisanie programu najpierw na kartce. Dopiero gdy rozwiązanie zadania jest już w pełni gotowe, rozpoczyna się pracę na komputerze.
- Nie jest możliwe wygrać zawody, jeśli któryś z członków drużyny nie jest w stanie rozwiązać samodzielnie 1-2 zadań.
- Komunikacja między członkami drużyny musi być bardzo efektywna. Zawodnicy powinni znać się na tyle dobrze, aby w momencie napotkania na problem szybko ustalić kogo powinni zapytać o radę oraz być w stanie niezwłocznie przedstawić powstały

problem. Bezcelowe jest zadawanie pytań, które doprowadzą do wciągnięcia kolegi w dłuższą dyskusję — w takiej sytuacji często sensowniejsze jest po prostu przekazanie rozwiązywanego zadania komuś innemu.

- Wygrywa ten kto ma największy „ciąg na bramkę". Wysoki poziom techniczny jest warunkiem koniecznym, ale nie wystarczającym. Zmagania między drużynami Uniwersytetu Warszawskiego w latach 2002-2003 — Hawks i Eagles, nie toczyły się na poziomie technicznym, bo ten był stały.

- Liczy się zespół i tylko w przypadku pracy całego zespołu można wygrać. Niezależnie od poziomu technicznego indywidualnych zawodników, brak pracy dowolnego zawodnika uniemożliwia zwycięstwo.

- Strategia: na początku zadania dzielone są między zawodników alfabetycznie. Potem każdy robi swoje, a gdy zrobi wszystkie, bierze nierozpoczęte zadanie od kolegi. Tylko w bardzo wyjątkowych sytuacjach można wymieniać zadania. Zadania są na tyle proste, że praktycznie nie ma specjalistów od określonej dziedziny — każdy może zrobić każde zadanie.

- Każdy powinien niezależnie realizować swoje zadania od A do Z — tylko to buduje odpowiedzialność za zadania.

- Po każdym treningu należy policzyć ile zadań kto rozwiązał. Bardzo dobrym pomysłem jest, aby osoba, która zrobiła najmniejszą liczbę zadań stawiała każdorazowo obiad. W przypadku, gdy któremuś z zawodników to nie odpowiada, gdyż „on zawsze, w takim układzie, będzie stawiał obiad", to lepiej wymienić takiego zawodnika na dowolnego bardziej zmotywowanego.

- Pojęcie kapitana zespołu jest bez sensu. Trzyosobowy zespół powinien rozumieć się bez słów. Jeśli tak nie jest, to nie ma szans na zwycięstwo.

- Regularne treningi są bardzo potrzebne. Jeśli ktoś nie ma na nie czasu lub ochoty, to lepiej go wymienić.

- Należy wyznaczać sobie wysokie cele. Jeśli komuś zależy tylko na pokonaniu innych drużyn z uczelni, to jest to za mało. Jedyny sensowny cel to zwycięstwo w całym konkursie. Mniej ambitne cele łatwiej się realizuje, ale niewiele się dzięki temu osiąga. Niech każdy pomyśli sobie jaki jest jego cel, gdyż mam wrażenie, że dla większości drużyn był nim sam wyjazd na finały.

Tomasz Malesiński, brązowy medalista Mistrzostw Świata w Programowaniu Zespołowym 2004

- Treningi nie powinny być smutnym obowiązkiem. Po każdym treningu, niezależnie od jego wyników, drużyna jest lepsza, bardziej zgrana i bliższa celu. W czasie treningów sprawdza się przydatność wskazówek starszych kolegów i dopracowuje strategię.

- Ważna jest motywacja, wiara we własne siły i uparte dążenie do celu. W finałach łatwo jest pokonać kilkadziesiąt drużyn, ale prawdziwa gra toczy się między kilkunastoma najlepszymi. Można z nimi wygrać, ale nie przyjdzie to łatwo.

- Komunikację należy ograniczyć do minimum. Na początku konkursu należy podzielić się równo zadaniami, każdy członek drużyny zapoznaje się ze swoimi i rozwiązuje je sam w kolejności od najłatwiejszego do najtrudniejszego.

- Nie należy myśleć nad rozwiązaniem zadania zanim go się dokładnie nie zrozumie. Nie powinno się także pisać kodu, dopóki nie ma pewności co do metody rozwiązania zadania. Nie wysyła się też rozwiązania, bez upewnienia się, że jest poprawne. Należy pracować samodzielnie, nie przeszkadzając innym. Zadanie może być odłożone tylko wówczas, gdy stwarza duże problemy i czas pozwala na rozpoczęcie innego od nowa.
- Jeśli komputer jest zajęty, kod należy pisać na kartce. Gdy przepisuje się kod do komputera, należy jeszcze raz dokładnie przeczytać, aby nie było pomyłek (przy okazji poprawić).
- W przypadku, gdy program nie działa, należy zwolnić klawiaturę i szukać dalej błędów na wydruku. Debugując przy komputerze należy spróbować wypisać wyniki częściowe działania programu. Zwykle jest to skuteczniejsze od wykonywania programu krok po kroku, a dodatkowo można te wyniki wydrukować i przeanalizować nie zajmując komputera.
- Kilka gotowców warto mieć, ale tak naprawdę, to rzadko się one przydają. Liczą się umiejętności. Gotowce tylko je uzupełniają.
- Pod koniec konkursu zasady nieco się zmieniają. Należy ocenić, które zadania można jeszcze dokończyć i wówczas pracuje się razem. Każdy zawodnik zapoznaje się z treścią zadania, otrzymując wydruk bieżącej wersji programu. Można wtedy wysyłać rozwiązania, nawet gdy są pewne wątpliwości, co do ich poprawności, tylko trzeba mieć świadomość, że desperackie próby zaliczenia zadania rzadko skutkują.
- Trzeba umieć sobie radzić z błędami. Błąd staje się groźny dopiero wtedy, gdy powoduje rezygnację z dalszej walki.
- Nie należy zapominać o odpoczynku. Chwila relaksu ułatwia spojrzenie na problem z innej strony i zazwyczaj pozwala na znalezienie jakiegoś rozwiązania.

Krzysztof Onak, Mistrz Świata w Programowaniu Zespołowym 2003

- Każdy zawodnik w grupie musi dawać z siebie jak najwięcej, nie licząc na pozostałych i walczyć tak, jakby wszystko zależało wyłącznie od niego.
- Nie należy bać się odpowiedzialności. Jeżeli zadanie jest proste, należy je rozwiązać, nie czekając na innych.
- Zadanie, które się zaczyna należy dociągnąć do końca, ale też trzeba mieć odwagę zrezygnować, gdy naprawdę nie wychodzi.
- Należy nauczyć się debugować na kartce. W ten sposób oszczędza się czas, a ponadto czytając program na papierze, często wychwytuje się błędy, które byłoby trudno zdebugować.
- Jeżeli coś nie działa, to przede wszystkim należy zakładać, że błąd tkwi po stronie zawodnika, a nie po stronie jury. Wówczas łatwiej zmobilizować się do większego wysiłku.
- Nie należy dyskutować za wiele w czasie konkursu. Warto skoncentrować się wyłącznie na swoim zadaniu.
- Na początku konkursu najlepiej losowo podzielić się zadaniami i każdy rozwiązuje swoje. Oczywiście zadaniami można się wymieniać. W końcówce odrzućcie zadania, co do których wiecie, że ich nie zrobicie i być może wtedy rozwiązujcie wspólnie jakieś zadanie.

- Musicie nabrać pewności siebie i wiedzieć, że możecie na sobie polegać. Musicie dobrze czuć się w zespole.
- Nie ufajcie do końca tym zasadom. Znajdźcie coś, co jest dobre w przypadku Waszego zespołu.

Marcin Stefaniak, srebrny medalista Mistrzostw Świata w Programowaniu Zespołowym 2001

- Nie używaj debuggera. Jeśli coś nie działa, należy patrzeć na wydruk programu, aż dostąpi się oświecenia. Drukuj zadanie za każdym razem, gdy jest wysyłane do oceny, nawet jeśli okaże się, że rozwiązanie było poprawne, to papier przyda się do czegoś innego.
- Bądźcie biegli w tym, jak działa Wejście/Wyjście w językach konkursu (dokładnie w tej wersji kompilatora jaka będzie używana), jak efektywnie (ważne dla dużych wejść i wyjść), jakie ma błędy i nietypowe cechy. Jest to w gruncie rzeczy jedyna biblioteka typowo informatyczna (czyli zasadniczo niedoskonała i niepewna) z której musicie korzystać.
- Zabierzcie na zawody papier w kratkę, papier w heksy już wyszedł z mody.
- Po zaliczonym zadaniu, przybijajcie sobie dłonie, najlepiej hałasując na tyle głośno, aby zdeprymować drużyny w pobliżu.
- Nie patrzcie na ranking! To zazwyczaj przynosi pecha.
- Pamiętajcie, że aby wygrać finały, należy wpierw wygrać konkurs regionalny. Nie mówiąc już o zawodach ogólnopolskich.
- Wypracujcie własny, efektywny styl pracy i podział zadań. Nie dajcie sobie narzucić czegoś z zewnątrz; każda drużyna jest inna i nie ma tu jednakowej recepty. Nie wszyscy przecież muszą lubić np. geometrię obliczeniową.
- Zdarza się że zadania mają nieprawidłowe testy i treści. Zdarza się, ze komputery nawalają. Mimo to walczcie! Może to właśnie wam dopisze szczęście i jeszcze na tym skorzystacie.
- Nie musicie zrobić wszystkich zadań, wystarczy więcej niż inni.

Dodatek C

Zbiór zadań na programowanie dynamiczne

Literatura
[ASD] - 1.8.2
[WDA] - 16

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 348	acm.uva.es - zadanie 531	acm.uva.es - zadanie 116
acm.uva.es - zadanie 674	acm.uva.es - zadanie 562	acm.uva.es - zadanie 147
acm.uva.es - zadanie 10003	acm.uva.es - zadanie 10069	acm.uva.es - zadanie 357
acm.uva.es - zadanie 10081	acm.uva.es - zadanie 10201	acm.uva.es - zadanie 366
acm.uva.es - zadanie 10131	acm.uva.es - zadanie 10280	acm.uva.es - zadanie 711
acm.uva.es - zadanie 10198	acm.uva.es - zadanie 10296	acm.uva.es - zadanie 10032
acm.uva.es - zadanie 10259	acm.uva.es - zadanie 10400	acm.uva.es - zadanie 10154
acm.uva.es - zadanie 10271	acm.uva.es - zadanie 10405	acm.uva.es - zadanie 10157
acm.uva.es - zadanie 10304	acm.uva.es - zadanie 10549	acm.uva.es - zadanie 10261
acm.uva.es - zadanie 10482	acm.uva.es - zadanie 10558	spoj.sphere.pl - zadanie 350
acm.uva.es - zadanie 10529	acm.uva.es - zadanie 10930	spoj.sphere.pl - zadanie 366
acm.uva.es - zadanie 10811	spoj.sphere.pl - zadanie 292	spoj.sphere.pl - zadanie 388
spoj.sphere.pl - zadanie 346	spoj.sphere.pl - zadanie 338	acm.sgu.ru - zadanie 132
spoj.sphere.pl - zadanie 365	spoj.sphere.pl - zadanie 345	acm.sgu.ru - zadanie 278
spoj.sphere.pl - zadanie 394	spoj.sphere.pl - zadanie 348	
spoj.sphere.pl - zadanie 402	spoj.sphere.pl - zadanie 364	
acm.sgu.ru - zadanie 104	acm.sgu.ru - zadanie 183	
	acm.sgu.ru - zadanie 269	
	acm.sgu.ru - zadanie 304	

Dodatek D

Zbiór zadań na programowanie zachłanne

Literatura
[ASD] - 1.8.3
[WDA] - 17

Ćwiczenia

Proste	Średnie	Trudne
acm.uva.es - zadanie 120	acm.uva.es - zadanie 10461	acm.uva.es - zadanie 410
acm.uva.es - zadanie 10020	acm.uva.es - zadanie 10563	acm.uva.es - zadanie 714
acm.uva.es - zadanie 10249	acm.uva.es - zadanie 10716	acm.uva.es - zadanie 10382
acm.uva.es - zadanie 10440	acm.uva.es - zadanie 10720	acm.uva.es - zadanie 10665
acm.uva.es - zadanie 10821	acm.uva.es - zadanie 10785	acm.uva.es - zadanie 11006
acm.uva.es - zadanie 10954	acm.uva.es - zadanie 10982	spoj.sphere.pl - zadanie 417
acm.uva.es - zadanie 10965	spoj.sphere.pl - zadanie 247	acm.sgu.ru - zadanie 257
spoj.sphere.pl - zadanie 661	acm.sgu.ru - zadanie 259	

Bibliografia

[WDA] Wprowadzenie do algorytmów, Thomson H. Cormen, Charles E. Leiserson, Ronald L. Rivest, WNT, 2004

[ASD] Algorytmy i struktury danych, L. Banachowski, K. Diks, W. Rytter, WNT, 2003

[KDP] Kombinatoryka dla programistów, W. Lipski, WNT, 2004

[SZP] Sztuka Programowania, Donald E. Knuth, WNT, 2002

[ASP] Algorytmy + struktury danych = programy, Niklaus Wirth, WNT, 2001

[RII] Rzecz o istocie informatyki: Algorytmika, D. Harel, WNT, 2000

[MD] Matematyka dyskretna, K. A. Ross, C. R. B. Wright, PWN, 2000

[MK] Matematyka konkretna, R. L. Graham, D. E. Knuth, O. Patashnik, PWN, 1998

[TLK] Wykład z teorii liczb i kryptografii, N. Koblitz, WNT, 2006

[OPK] Optymalizacja Kodu. Efektywne wykorzystanie pamięci - programowanie, Kris Kaspersky, RM, 2005

[OI] Olimpiada Informatyczna (http://www.oi.edu.pl/)

[OI1] I Olimpiada Informatyczna 1993/1994. Warszawa - Wrocław, 1994

[OI2] II Olimpiada Informatyczna 1994/1995. Warszawa - Wrocław, 1995

[OI3] III Olimpiada Informatyczna 1995/1996. Warszawa, 1996

[OI4] IV Olimpiada Informatyczna 1996/1997. Warszawa, 1997

[OI5] V Olimpiada Informatyczna 1997/1998. Warszawa, 1998

[OI6] VI Olimpiada Informatyczna 1998/1999. Warszawa, 1999

[OI7] VII Olimpiada Informatyczna 1999/2000. Warszawa, 2000

[OI8] VIII Olimpiada Informatyczna 2000/2001. Warszawa, 2001

[OI9] IX Olimpiada Informatyczna 2001/2002. Warszawa, 2002

[OI10] X Olimpiada Informatyczna 2002/2003. Warszawa, 2003

[OI11] XI Olimpiada Informatyczna 2003/2004. Warszawa, 2004

[OI12] XII Olimpiada Informatyczna 2004/2005. Warszawa, 2005

[SGU] Saratov State University :: Online Contester (http://acm.sgu.ru/)

[UVA] Valladolid Programming Contest Site (http://acm.uva.es/)

[SPO] Sphere Online Judge (http://spoj.sphere.pl/)

[SGU] Saratov State University :: Online Contester (http://acm.sgu.ru/)

[PCH] Programming Challenges: The Programming Contest Training Manual, S. S. Skiena, M. A. Revilla, Springer - Verlag New York, Inc., 2003

[STL] Standard Template Library Programmer's Guide (http://www.sgi.com/tech/stl/)

[GCC] GCC home page (http://gcc.gnu.org/)

[ECP] Effective C++, Third Edition, S. Meyers, Addison - Wesley (2005)

[LCA] The LCA Problem Revisited, M. A. Bender, M. Farach - Coltion. (http://www.cs.sunysb.edu/ bender/pub/lca.ps)

[AKS] Primes is in P, M. Agrawal, N. Kayal, N. Saxena, Annals of Mathematics 160(2): 781-793 (2004). (http://www.cse.iitk.ac.in/users/manindra/)

[PCC] Polyhedral Combinatorics and Combinatorial Optimization, A. Schrijver, CWI and University of Amsterdam (http://homepages.cwi.nl/ lex/)

[NRP] Numerical Recipes in C, The Art of Scientific Computing, W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge University Press

[EAT] Efficient Algorithms on Texts, M. Crochemore, W. Rytter

[CON] Combinatorial Optimization: Networks and Matroids, Holt, Rinehart, Winston, 1976

[CST] On-line Construction of Suffix Trees, E. Ukkonen, Algorithmica, 14(3) pp249-260, 1995

[BST] Randomized Binary Search Trees, C. Martinze, S. Roura, Journal of the ACM, Vol. 45, No. 2, March 1998, pp 288-323

Skorowidz