Program nauczania w ramach Olimpiady Informatycznej

Wojciech Baranowski

1 maja 2024

Spis treści

1 Wstęp

W tej sekcji umieszczone będą pojęcia, których zrozumienie oraz znajomość praktyczna są niezbędne do dalszej kontynuacji nauki.

1.1 Git

Git - system kontroli wersji opracowany przez Linusa Torvalds'a w 2005 roku. Służy on do utrzymywania historii zmian danego zbioru plików i katalogów w potencjalnie rozdystrybuowanym środowisku. Pozwala na zrównoleglenie pracy wielu deweloperów nad jednym projektem, poprzez utrzymywanie nieliniowej struktury zmian.

1.1.1 Git - przykład liniowy:

Alice rozwija aplikację, jednak przydałaby się jej możliwość potencjalnego powrotu do poprzedniej (działającej) wersji programu, jeśli nowa zmiana pójdzie nie po jej myśli. Decyduje się więc skorzystać z git'a. Każdorazowo, gdy jej program znajdzie się w dostatecznie stabilnym stanie, Alice tworzy commit'a utworzonych przez nią zmian. Dzięki temu w dowolnym momencie może wrócić do dowolnego zacommitowanego punktu czasowego.

1.1.2 Git - w praktyce:

Mamy gotowy program, który liczy średnią arytmetyczną dwóch liczb (póki co całkowitych, zwracany wynik także jest całkowity).

1. inicjalizujemy repozytorium git'a: **git init**,
2. tworzymy gałąź (linię czasową) dla naszego programu: **git checkout -b nazwa**,
3. dodajemy nasz program do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
4. commitujemy nasze zmiany: **git commit -m "initial commit"**.

Następnie przerabiamy program tak, by obsługiwał liczby zmiennoprzecinkowe.

1. dodajemy nową wersję pliku do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
2. commitujemy nasze nowe zmiany: **git commit -m "floating points"**,
3. możemy zobaczyć historię naszych commitów przy pomocy następującej komendy: **git reflog**.

Załóżmy, że chcemy się cofnąć do poprzedniego commita, w którym program działał jedynie dla liczb całkowitych, a następnie zmienić jego działanie tak, że tylko wynik byłby liczbą zmiennoprzecinkową.

1. wylistowywujemy historię commitów: **git reflog**,
2. znajdujemy hash interesującego nas commita, do którego chcemy wrócić,
3. resetujemy program do wersji z wybranego commita: **git reset --hard hash-commita**,

W ten sposób możemy rozpocząć pracę na wcześniejszej wersji programu, a następnie zacommitować alternatywne rozwiązanie.

1.1.3 Repozytorium

Repozytorium gita pozwala na przechowywanie historii wersji w chmurze. Najpopularniejszym serwisem świadczącym takie usługi jest **github.com**. W celu skorzystania z githuba niezbędne jest utworzenie konta w serwisie. Następnie w ramach danego projektu należy utworzyć odpowiadające mu repozytorium, z którym następnie można zsynchronizować lokalnego gita.

1.1.4 Repozytorium - przykład

Dotychczasowo Alice rozwijała swoją aplikację na jednym urządzeniu. Ostatnio stwierdziła, że dostęp do projektu na drugim urządzeniu znacznie ułatwił by jej pracę. Decyduje się więc na zastosowanie repozytorium w serwisie github.com. Alice tworzy w serwisie repozytorium dla jej aplikacji, a następnie synchronizuje z nim swojego gita. Od tej pory Alice może w dowolnym momencie umieścić (wypchnąć) jej lokalne zmiany na githuba, a następnie pobrać (sklonować) na drugie urządzenie.

1.1.5 Repozytorium - klucze

Aby móc uwierzytelnić naszego klienta konsolowego w serwisie github, a co za tym idzie uzyskać dostęp do wypychania lokalnych zmian do repozytorium, musimy najpierw skonfigurować klucze dostępu dla danego urządzenia.

1. generujemy klucze w naszym systemie, które posłużą do uwierzytelniania w githubie **ssh-keygen -t rsa -b 4096 -C "adres-email-użyty-na-githubie"**,
2. domyślnie klucze dostępne są w katalogu **/home/nazwa-użytkownika/.ssh**,
3. kopiujemy zawartość klucza publicznego (klucz z rozszerzeniem **.pub**),
4. dodajemy klucz na githubie w zakładce **settings/SSH-and-GPG-keys** jako nowy klucz SSH.

Ponadto po stronie lokalnego gita wymagana będzie autoryzcja użytkownika. W konfiguracji gita należy ustawić nazwę oraz email użytkownika:

- git config --global user.name "IMIE NAZWISKO"
- git config --global user.email "EMAIL"

1.1.6 Repozytorium - w praktyce

Założyliśmy wcześniej konto w serwisie github.com. Teraz chcemy udostępnić w nim nasz projekt.

1. tworzymy nowe repozytorium na githubie, które będzie przeznaczone dla naszej aplikacji,
2. łączymy lokalne repozytorium gita z githubem: **git remote add origin adres-repozytorium**,
3. wypychamy lokalne zmiany do repozytorium: **git push origin nazwa-gałęzi**.

Następnie chcemy sklonować projekt z githuba na innym urządzeniu.

1. wchodzimy do katalogu, w którym chcemy przechowywać nasz projekt,
2. klonujemy projekt za pomocą linku: **git clone link-do-sklonowania-repozytorium**.

W przypadku, gdy wprowadziliśmy zmiany na jedym urządzeniu i chcemy zaktualizować stan projektu na innym urządzeniu, wykonujemy następujące czynności.

1. wchodzimy do katalogu projektu,
2. aktualizujemy gita o informacje odnośnie stanu repozytorium na githubie: **git fetch**,
3. aktualizujemy lokalnego gita: **git pull**.

W przypadku wystąpienia niezgodnych wersji wersjonowania pomiędzy gitem lokalnym oraz githubem, wystąpić mogą tzw. konflikty, czyli miejsca w projekcie, gdzie programista musi zadecydować jaką wersję części aplikacji git ma uznać za poprawną. W zależności od narzędzie konflikty można rozwiązywać na różne sposoby, dlatego przytoczę jedynie scenariusze, w których konieczne może okazać się zastosowanie działań forsownych.

1. Jeśli próba aktualizacji gita lokalnego się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na githubie, najprościej jest po prostu usunąć lokalny katalog z projektem, a następnie ponownie go sklonować.
2. Jeśli próba wypchnięcia zmian na githuba się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na lokalnym gicie, to możemy dokonać wypchnięcia siłowego: **git push --force origin nazwa-gałęzi**.

1.2 Złożoność obliczeniowa

Złożoność obliczeniowa jest miarą wzrostu liczby operacji wykonywanych przez algorytm w miarę wzrostu rozmiaru danych. W praktyce miara ta zakłada jedynie asymptotyczną ocenę wzrostu, pomijając czynniki mniej znaczące lub stałe. Stosowana jest głównie z uwagi na niezależność od środowiska wykonującego algorytm oraz trudności w szacowaniach opartych na fizycznych aspektach obliczeń takich jak częstotliwość cyklów zegarowych lub opóźnienia na poszczególnych komponentach jednostki wykonującej operacje.

1.2.1 Symbol O

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $O(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $f(x) \leq cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = O(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \geq 0.$$

1.2.2 Symbol O - w praktyce

- $f(x) = 2x + 100 = O(x)$,
- $f(x) = 3x^2 + 16x = O(x^2)$,
- $f(x) = 5xlog_4(x) = O(xlog(x))$,
- $f(x) = \frac{1}{x^2-4} = O(1)$,
- $f(x) = 72 = O(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = O(x)$,
- $f(x) = x^x + x! = O(x^x)$.

1.2.3 Symbol o

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $o(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{>0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $f(x) < cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = o(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

1.2.4 Symbol o - w praktyce

- $f(x) = 2x + 100 = o(x^2)$,
- $f(x) = 3x^2 + 16x = o(x^8)$,
- $f(x) = 5xlog_4(x) = o(x^2)$,
- $f(x) = \frac{1}{x^2-4} = o(1)$,
- $f(x) = 72 = o(n)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = o(xlog(x))$,
- $f(x) = x^x + x! = o(x^{x!})$.

1.2.5 Symbol Ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $g(x) \leq cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0 \vee \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.6 Symbol Ω - w praktyce

- $f(x) = 2x + 100 = \Omega(x)$,
- $f(x) = 3x^2 + 16x = \Omega(x^2)$,
- $f(x) = 5xlog_4(x) = \Omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \Omega(\frac{1}{x^{100}})$,
- $f(x) = 72 = \Omega(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \Omega(1)$,
- $f(x) = x^x + x! = \Omega(x!)$.

1.2.7 Symbol ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{\geq 0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $g(x) < cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.8 Symbol ω - w praktyce

- $f(x) = 2x + 100 = \omega(log(x))$,
- $f(x) = 3x^2 + 16x = \omega(x)$,
- $f(x) = 5xlog_4(x) = \omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \omega(\frac{1}{x^3})$,
- $f(x) = 72 = \omega(\frac{1}{x})$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \omega(log(x))$,
- $f(x) = x^x + x! = \omega(x^{100})$.

1.2.9 Symbol Θ

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Theta(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnych $c_1, c_2 \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Theta(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0.$$

1.2.10 Symbol Θ - w praktyce

- $f(x) = 2x + 100 = \Theta(x)$,
- $f(x) = 3x^2 + 16x = \Theta(x^2)$,
- $f(x) = 5xlog_4(x) = \Theta(xlog(x))$,
- $f(x) = 72 = \Theta(1)$,
- $f(x) = log(log(x)) = \Theta(log(log(x))$,
- $f(x) = 2^{x+1} = \Theta(2^x)$,
- $f(x) = 2^{1000000} = \Theta(1)$.

1.3 Szacowanie wzorcowej złożoności

1.3.1 Koncept

Popularną strategią przydatną w procesie projektowania wzorcowego rozwiązania danego problemu algorytmicznego zadanego na Olimpiadzie Informatycznej jest szacowanie pożądanej złożoności. Każde zadanie posiada jasno sprecyzowane dane wejściowe, a w szczególności ich rozmiar, co może pozwolić na określenie złożoności rozwiązania wzorcowego, poprzez oszacowanie maksymalnej liczby operacji jaką potencjalnie może wykonać program, skonfrontowanie jej z limitami czasowymi i szybkością procesora, a następnie dopasowanie do jednej ze standardowych postaci funkcji wyrażającej złożoność.

Dla przykładu podane są najpopularniejsze złożoności wzorcowych rozwiązań zadań z Olimpiady Informatycznej (podane w kolejności rosnącej):

- $O(n)$
- $O(nlog(n))$
- $O(n\sqrt(n))$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

1.3.2 Szacowanie

Biorąc pod uwagę specyfikację procesorów używanych do wykonywania zadań na środowiskach olimpiady w celu ich testowania (taktowanie 2GHz), Możemy założyć, że w przeciągu jednej sekundy będzie on w stanie wykonać około 2 miliardy poleceń. Z uwagi na architekturę procesorów, w szczególności fakt, że instrukcje znane nam z języka C++ często składają się z wielu podstawowych instrukcji procesora, bezpieczniej jest przyjąć, że procesor będzie w stanie przetworzyć parędziesiąt milionów instrukcji języka C++ na sekundę. Idąc dalej, jeśli chcielibyśmy zastosować powyższe szacowanie w kontekście złożoności obliczeniowej, zależnej od wielkości danych wejściowych, trzeba wziąć pod uwagę wszelakie stałe, które pojęcie złożoności pomija, tak więc ostatecznym szacowaniem, które należałoby zastosować jest kilka milionów operacji na sekundę.

1.3.3 Przykład

Dla przykładu weźmy pod uwagę zadanie, w którym wielkość danych określona jest przez $n \leq 200000$, natomiast algorytm powinien się wykonywać co najwyżej 5 sekund. Opierając się na powyższych założeniach, możemy więc przyjąć, że w przeciągu 5 sekund będzie on w stanie wykonać około kilkanaście milionów operacji. Spróbujmy dopasować liczbę operacji w przypadku największej możliwej wartości n ($n = 200000$) do jednej z typowych funkcji wyrażających złożoność:

- $O(n^2)$: $200000^2 = 4 * 10^{10}$, co znacznie wychodzi poza zakres kilkunastu milionów.
- $O(n\sqrt{n})$: $200000 * \sqrt{200000} \approx 450 * 200000 = 9 * 10^7$, co daje wynik o niecały rząd wielkości za duży.
- $O(nlog(n))$: $200000 * log(200000) \approx 15 * 200000 = 3 * 10^6$, co mieści się w naszych szacowaniach.
- $O(n)$, równe po prostu 200000 jest niedoszacowaniem o około 2 rzędy wielkości.

Po powyższej analizie można wywnioskować, że docelową złożonością rozwiązania, której prawdopodobnie spodziewają się autorzy zadania jest złożoność $O(nlog(n))$.

1.3.4 W praktyce

W praktyce ograniczenia czasowe zadań zazwyczaj mieszczą się w przedziale od jednej do kilkunastu sekund. Oznacza to, że dla poszczególnych rozmiarów danych wejściowych możemy przyporządkować złożoności, które z dużym prawdopodobieństwem mogą okazać się złożonościami docelowymi:

- $[1 - 10] : O(n!)$
- $[10 - 15] : O(3^n)$ / $O(2^n)$
- $[15 - 20] : O(2^n)$
- $[20 - 100] : O(n^4)$
- $[100 - 300] : O(n^3)$
- $[300 - 1000] : O(n^2 log(n))$
- $[1000 - 5000] : O(n^2)$
- $[5000 - 50000] : O(n\sqrt{n})$ / $O(nlog^2(n))$
- $[50000 - 500000] : O(nlog(n))$
- $[500000 - 1000000] : O(nlog(n))$ / $O(n)$
- $[1000000+] : O(n)$

Warto zaznaczyć, że jeśli rozwiązanie posiada nienaturanie niskie albo wysokie stałe, to powyższe szacowania mogą okazać się błędne. Należy więc polegać przede wszystkim na swojej własnej intuicji.

2 STL

STL - Standard Template Library jest biblioteką będącą zbiorem gotowych szablonów przydatnych algorytmów, struktur danych oraz innych narzędzi, które przydają się w pisaniu kodu. Narzędzie to umożliwia pominięcie implementacji niekiedy bardzo złożonego kodu, oferując gotowe rozwiązania, stworzone oraz rozwijane przez specjalistów w tej dziedzinie.

Aby skorzystać z całości STLa należy dołączyć do programu bibliotekę **bits/stdc++.h** poprzez umieszczenie w programie dyrektywy #**include**<**bits/stdc++.h**> W ramach uproszczenia przestrzeni nazw zaleca się także dodanie linii **using namespace std;**, dzięki której uniknąć można każdorazowego pisania przedrostka **std::** przed nazwą struktury bądź funkcji.

2.1 Przydatne narzędzia

2.1.1 Auto

Auto jest słowem kluczowym, którego można użyć w zastępstwie za typ zmiennej, pod warunkiem, że w momencie deklaracji następuje także przypisanie wartości. Spowoduje to wydedukowanie typu zmiennej przez kompilator na podstawie kontekstu. Przykładowo:

```
auto pewnaLiczbaCalkowita = 311;
```

Dzięki auto możemy nieprzejmować się jakiego typu danymi operujemy; w szczególności gdy nie jesteśmy tego typu pewni na przykład w momencie, gdy używamy nieznanej nam dotychczas funkcji języka. Minusem tego rozwiązania jest zmniejszona czytelność kodu, ponieważ czytając kod, każdorazowo musimy zastanawiać się jaki typ faktycznie kryje się pod słowem kluczowym auto. Szczególne zastosowanie auto znajduje w procesie iterowania po strukturach danych, które przedstawione będą w dalszej części rozdziału. Przykładowo chcą przeiterować się po wszystkich elementach danej struktury **nazwaStruktury**, możemy posłużyć się nastepującą pętlą:

```
for(auto elementStruktury : nazwaStruktury) {
    //elementStruktury jest elementem naszej struktury rozpatrywanym w danej iteracji
}
```

2.1.2 Komparator

Komparatory nie są stricte elementem STL'a, jednakże znajdują szerokie zastosowanie w połączeniu ze strukturami oraz funkcjami udostępnianymi przez STLa. Formalnie jest on funkcją typu bool, która na wejściu otrzymuje dwa elementy, natomiast na wyjściu zwraca informację czy element pierwszy jest mniejszy od drugiego. Dla przykładu komparator porządkujący liczby według ich wartości modulo 3:

```
bool komparatorMod3(int a, int b) {
    return (a % 3) < (b % 3);
}
```

Aby użyć komparatorów, najczęściej należy podać je tworzonej strukturze w postaci argumentu w operatorze diamentowym (<**..., komparator**>) albo jako argument do przyjmującej go funkcji (**(..., komparator)**). Istnieją także inne sposoby tworzenia komparatorów, takie jak wyrażenia lambda czy struktury z przeciążonymi operatorami, jednakże nam w zupełności wystarczy powyższa forma. Ciekawskich ponownie odsyłam do innych źródeł.

2.1.3 Iterator

Iterator sam w sobie nie ma zastosowania. Struktura ta znajduje zastosowanie dopiero w połączeniu z odpowiednimi strukturami danych. Iterator jest swojego rodzaj wskaźnikiem, który możemy poruszać między elementami struktury liniowej, a także odczytywać wartość elementu, na który aktalnie wskazuje. Iterator dla danej struktury danych deklarujemy następująco:

```
struktura::iterator nazwa;
```

Dla przykładu:

```
set<int>::iterator nazwaIteratora;
```

Aby zastosować iterator na wybranej strukturze danych, należy ustawić go na jednej z pozycji, do których wybrane struktury oferują dostęp; przykładowo dla kolejki będzie to **front()**, natomiast dla zbioru **begin()** oraz **end()**. Przykładowo chcąc ustawić iterator na początek zbioru:

```
set<int>::iterator nazwaIteratora = zbior.begin();
```

Iteratorem poruszamy tak samo jak klasycznym wskaźnikiem; dodając albo odejmując od niego wartość równą liczbie elementów o którą chcemy go poruszyć. Aby odczytać wartość elementu, na którym obecnie zatrzymał się iterator należy odwołać się do jego wartości identycznie jak w przypadku wskaźnika. Niech zbior będzie zbiorem o strukturze (1, 2, 3):

```
set<int>::iterator nazwaIteratora = zbior.begin();
*nazwaIteratora // odczytanie wartości 1
nazwaIteratora++;
*nazwaIteratora // odczytanie wartości 2
nazwaIteratora--;
*nazwaIteratora; odczytanie wartości 1
nazwaIteratora+=2;
*nazwaIteratora; odczytanie wartości 3
```

2.2 Struktury nieuporządkowane

2.2.1 Pair

Para jest strukturą danych oferującą możliwość agregacji dwóch elementów, niekoniecznie tego samego typu. Samą parę deklaruje się w następujący sposób:

```
pair<typ1, typ2> nazwa;
```

Na przykład:

```
pair<int, string> nazwaPary;
```

Aby odwołać się do poszczególnych elementów pary, należy odnieść się do elementów **first** oraz **second** w następujący sposób:

```
nazwaPary.first = 4;
nazwaPary.second = "cztery";
```

Ponadto w tworzeniu par przydatna może okazać się metoda **make_pair** działająca jako konstruktor:

```
pair<int, string> nazwaPary = make_pair(4, "cztery");
```

Dozwolona jest także inicjalizacja przy użyciu krotki rzędu 2:

```
pair<int, string> nazwaPary = {1, "cztery"};
```

W szczególności stworzyć można pary kaskadowe, które mogą pomieścić więcej elementów, kosztem zagłębienia struktury. Poniżej przykład implementacji oraz użycia pary pozwalającej na przechowywanie czterech elementów, przydatnej chociażby w implementacji niektórych struktur grafowych:

```
pair<pair<int, double>, pair<bool, string> > czteroPara;
czteroPara = make_pair(make_pair(1, 2.5), make_pair(true, "tak"));
pair.first.first = 2;
pair.first.second = 3.5;
pair.second.first = false;
pair.second.second = "nie";
```

Pary pożna porównywać poprzez użycie standardowych komparatorów. Priorytetowo porównane zostaną pierwsze elementy, a następnie drugie. Przykładowo:

```
{1, 3} > {1, 2}
{2, 1} > {1, 10}
{5, 8} < {6, 1}
{1, 2} = {1, 2}
```

2.2.2 Vector

Vector jest strukturą zbliżoną do klasycznej tablicy. Oferuje on możliwość zapisu oraz odczytu wartości dowolnego elementu. W przeciwieństwie do tablicy jego rozmiar jest elastyczny - vector potrafi dynamicznie zmieniać rozmiar w zależności od zapotrzebowania, a także na życzenie, poprzez wykorzystaniem odpowiednich komend. Wadą vectora jest jego wydajność, która jest mniejsza, niż w przypadku klasycznej tablicy. Aby zadeklarować vector należy posłużyć się następującą składnią:

```
vector<typ> nazwa;
```

Na przykład:

```
vector<int> nazwaVectora;
```

W celu inicjalizacji vectora o danym rozmiarze początkowym należy sprecyzować rozmiar w nawiasach na końcu deklaracji:

```
vector<int> nazwaVectora(128);
```

Odwoływanie się do danego elementu vectora przebiega identycznie jak w przypadku tablicy:

```
nazwaVectora[12] = 21;
```

Dodatkowo możliwe jest dodawanie oraz usuwanie nowych elementów na początku, jak i końcu vectora, za pomocą komend **push_back**, **pop_back** oraz **push_front**, **pop_front**. Przykładowo dla vectora o wstępnej strukturze [1, 2, 3, 4] po wykonaniu komend

```
nazwaVectora.push_back(5)
nazwaVectora.pop_front()
nazwaVectora.pop_back()
nazwaVectora.push_back(6)
nazwaVectora.push_front(0)
```

otrzymamy następujący vector: [0, 2, 3, 4, 6]. Co więcej vector oferuje możliwość odczytu aktualnej liczby elementów, które się na nim znajdują - komenda **size()**, a także zapytanie czy jest on w danym momencie pusty - komenda **empty()**. Możliwe jest też usunięcie wszystkich elementów vectora komendą **clear()**, jednakże usuwa ona tylko same wartości, a nie pamięć przydzieloną vectorowi. W celu usunięcia pamięci przydzielonej vectorowi służy komenda **shrink_to_fit()**. Warta uwagi jest także funkcja **reverse()** umożliwiająca odwrócenie kolejności elementów w vectorze.

2.2.3 Queue

Queue, a właściwie kolejka jest strukturą opartą na zasadzie FIFO (first in - first out). Oznacza to, że jak w przypadku klasycznej kolejki, element który trafił do niej na początku, opuści ją jako pierwszy. Struktura oparta jest o listę wiązaną, co oznacza, że jesteśmy w stanie odwoływać się jedynie do jej pierwszego i ostatniego elementu. Zaletą tego rozwiązania jest dynamiczność struktury, która w dowolnym momencie użytkowania zajmuje jedynie tyle pamięci, ile w danej chwili potrzebuje, adaptując się każdorazowo po zmianie liczby elementów w niej zawartej. Kolejkę deklarujemy w następujący sposób:

```
queue<typ> nazwa;
```

Na przykład:

```
queue<int> nazwaKolejki;
```

Aby dodawać oraz usuwać elementy z kolejki, stosuje się komendy **push()** oraz **pop()**. Ponadto kolejka pozwala na odczyt elementu aktualnie znajdującego się na jej początku, przy pomocy komendy **front()**:

```
queue<int> nazwaKolejki;
nazwaKolejki.push(1);
nazwaKolejki.push(2);
nazwaKolejki.push(3);
nazwaKolejki.front(); // zwraca wartość 1
nazwaKolejki.pop();
nazwaKolejki.front(); // zwraca wartość 2
nazwaKolejki.push(4)
nazwaKolejki.front(); // zwraca wartość 2
```

Ponadto tak, jak w przypadku vectora, kolejka także pozwala na dostęp do aktualnie przechowywanej liczby elementów - komenda **size()** oraz zapytanie czy struktura jest pusta - komenda **empty()**.

2.2.4 Stack

Stack znany także jako stos jest strukturą bardzo zbliżoną do kolejki. Jego struktura oparta jest także na koncepcie listy wiązanej, przez co dostępny jest jedynie jego element wierzchni. Sam stos charakteryzuje się architekturą LIFO (last in, first out), dzięki której, jak w przypadku chociażby stosu książek, element który został dodany do stosu jako ostatni, opuści strukturę jako pierwszy. Stos deklarujemy w następujący sposób:

```
stack<typ> nazwa;
```

Na przykład:

```
stack<int> nazwaStosu;
```

Podobnie jak w przypadku kolejki, stos udostępnia metody pozwalające na dodanie do niego elementu - komenda **push()** oraz na usunięcie - komenda **pop()**. Odpowiednikiem komendy front, znanej z kolejki, jest w tym przypadku komeda **top()**, pozwalająca na odczyt elementu znajdującego się na szczycie stosu. Przykładowo:

```
stack<int> nazwaStosu;
nazwaStosu.push(1);
nazwaStosu.push(2);
nazwaStosu.push(3);
nazwaStosu.top(); // zwraca wartość 3
nazwaStosu.pop();
nazwaStosu.top(); // zwraca wartość 2
nazwaStosu.push(4);
nazwaStosu.top(); // zwraca wartość 4
```

Identycznie jak w przypadku kolejki oraz vectora, mamy dostęp do liczby elementów aktualnie wchodzących w skład stosu - komenda **size()** oraz zapytania czy struktura jest pusta - komenda **empty()**.

2.2.5 Deque

Deque jest strukturą zachowującą się dokładnie tak samo jak kolejka, z tą różnicą, że operacje dodawania, usuwania oraz odczytu elementu można wykonywać na obydwu jej końcach. Bikolejkę deklaruje się w następujący sposób:

```
deque<typ> nazwa;
```

Dla przykładu:

```
deque<int> nazwaBikolejki;
```

Operacje dodania, usunięcia oraz odczytu elementu z przodu bikolejki dokonuje się kolejno komendami **push_front()**, **pop_front()** oraz **front()**. Operacje wykonywane na końcu bikolejki noszą analogiczne nazwy, tylko z wykorzystaniem słowa back; **push_back()**, **pop_back()** oraz **back()**. Na przykład:

```
deque<int> nazwaBikolejki;
nazwaBikolejki.push_back(1); // (1)
nazwaBikolejki.push_front(2); // (1, 2)
nazwaBikolejki.push_back(3); // (3, 1, 2)
nazwaBikolejki.push_front(4); // (3, 1, 2, 4)
nazwaBikolejki.front(); // zwraca wartość 4
nazwaBikolejki.pop_back(); // (1, 2, 4)
nazwaBikolejki.pop_front(); // (1, 2);
nazwaBikolejki.back(); // zwraca wartość 1
```

Identycznie jak w przypadku kolejki klasycznej, dla bikolejki dostępne są także funkcje **size()** oraz **empty()**.

2.2.6 Bitset

Bitset jest strukturą będącą tablicą o stałym rozmiarze przechowywującą jedynie zera i jedynki. W przeciwieństwie do tablicy przechowywującej wartości typu bool, bitset zapisuje każdą wartość na jednym bicie, a nie bajcie, dzięki czemu zajmuje on 8 razy mniej miejsca oraz wykonuje operacje 8 razy szybciej. Bitset deklaruje się następująco:

```
bitset<rozmiar> nazwa;
```

Na przykład:

```
bitset<32> nazwaBitsetu;
```

Co więcej, podczas deklaracji bitsetu można przypisać do niego wartość początkową równą binarnej reprezentacji danej zmiennej:

```
int nazwaZmiennej = 311;
bitset<32> bitsetZeZmiennej(nazwaZmiennej);
```

Możliwa jest także operacja odwrotna - aby zamienić bitset na zmienną zapisaną w systemie dziesiątkowym należy zastosować funkcję **to_ulong()**, która zamienia zawartość bitsetu na zmeinną typu unsigned long:

```
bitset<32> nazwaBitsetu(311);
int wartoscBitsetu = (int) nazwaBitsetu.to_ulong(); // dodatkowo castujemy wynik na inta.
```

Proces odwołania do wybranego bitu struktury przebiega analogicznie jak w przypadku klasycznej tablicy. Przykładowo, aby odwołać się do bitu drugiego (odpowiadającego wartości 2^2 w zapisie dziesiątkowym), należy zastosować następujacą składnię:

```
nazwaBitsetu[2];
```

Ponadto bitset oferuje także metodę **test(i)**, która zwraca wartość typu bool w zależności, czy i-ty bit jest w tym momencie zapalony. Podobna składniowo jest także metoda **flip(i)**, zmieniająca wartość i-tego bitu na odwrotną:

```
bitset<3> nazwaBitsetu(0); // początkowo wszystkie bity są zgaszone
nazwaBitsetu.flip(1); // 010
nazwaBitsetu.flip(2); // 110
nazwaBitsetu.test(1); // true
nazwaBitsetu.test(0); // false
nazwaBitsetu.to_ulong(); // 6
```

Przydatna może się okazać także funkcja **count()**, która zwraca liczbę aktualnie zapalonych (mających wartość 1) bitów w bitsecie:

```
bitset<8> nazwaBitsetu(100) // 01100100
nazwaBitsetu.count() // 3
```

Prawdziwa potęga bitsetów ukazuje się w momencie stosowania na nich operacji logicznych. Istnieje bowiem możliwość przeprowadzenia logicznego ORa, ANDa oraz XORa na bitsecie w czasie ośmiokrotnie mniejszym, niż w przypadku klasycznej tablicy:

```
bitset<8> bitset1(120); // 01111000
bitset<8> bitset2(167); // 10100111
bitset1 | bitset2 // OR daje wynik 11111111
bitset1 & bitset2 // AND daje wynik 00100000
bitset1 ^ bitset2 // XOR daje wynik 11011111
```

2.3 Struktury uporządkowane

2.3.1 Priority queue

Priority queue, czyli kolejka priorytetowa jest strukturą zbliżoną działaniem do kolejki klasycznej. Jedyna różnica polega na tym, że kolejkę każdorazowo opuszcza element o największej wartości, a nie element najwcześniej dodany. Wewnętrznie struktura kolejki priorytetowej znacznie różni się od kolejki klasycznej, gdyż zamiast listy wiązanej, priority queue opiera się na kopcu binarnym. Deklaracja kolejki priorytetowej przebiega następująco:

```
priority_queue<typ> nazwa;
```

Przykładowo:

```
priority_queue<int> nazwaKolejkiPriorytetowej;
```

W związku z architekturą kopca biranrego operacje dodania elementu - komenda **push()** oraz usunięcia elementu - komenda **pop()** obarczone są złożonością rzędu **O(log(n))**, gdzie n to liczba elementów obecnie znajdująca się w strukturze. Wartość elementu największego dostępna jest przy pomocy komendy **top()**, a jej złożoność, podobnie jak w przypadku operacji na poprzednich strukturach, jest stała. Na przykład:

```
priority_queue<int> nazwaKolejkiPriorytetowej;
nazwaKolejkiPriorytetowej.push(1);
nazwaKolejkiPriorytetowej.push(11);
nazwaKolejkiPriorytetowej.push(3);
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 11
nazwaKolejkiPriorytetowej.pop();
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 3
nazwaKolejkiPriorytetowej.push(4);
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 4
```

Analogicznie do poprzednich struktur, kolejka priorytetowa także oferuje metody **size()** oraz **empty()**. W przypadku, gdy potrzebujemy kolejki priorytetowej, która priorytetyzuje elementy najmniejsze zamiast największych, możemy wykorzystać pewną właściwość matematyczną. Niech A będzie zbiorem wartości znajdujących się w strukturze, natomiast A^* zbiorem zawierającym elmenty przeciwne do elementów zbioru A (np. dla $A = (-1, 2, 3)$, $A^* = (1, -2, -3)$). Prawdziwe jest następujące zdanie logiczne:

$$a = max(A) \iff a = min(A^*)$$

Oznacza to, że w sytuacji, gdy chcemy priorytetyzować elementy o najmniejszej wartości, wystarczy, że do kolejki będziemy dodawać elementy do nich przeciwne, natomiast odczytując wartości z kolejki, ponownie je negować. Dla przykładu chcemy umieścić na kolejce następujące elementy: $(4, 5, 2, 1)$, aby następnie móc je pozdejmować w kolejności niemalejącej:

```
priority_queue<int> kolejkaMinimalizujaca;
kolejkaMinimalizujaca.push(-4); // negujemy umieszczane wartości
kolejkaMinimalizujaca.push(-5);
kolejkaMinimalizujaca.push(-2);
```

```
kolejkaMinimalizujaca.push(-1);
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 1
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 2
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 4
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 5
```

Możliwe jest także zastosowanie własnego komparatora, dzięki któremu możliwe będzie porównywanie elementów w dowolny sposób:

```
priority_queue<int, vector<int>, komparator> kolejkaZKomparatorem;
```

Warto zaznaczyć, że użycie komparatora wymusza na nas podanie enigmatycznego argumentu **vector<int>**, jednakże nie jest to istotne w danym momencie. Ciekawskich odsyłam do dokumentacji biblioteki STL.

2.3.2 Set

Set jest strukturą danych odzwierciedlającą klasyczny zbiór znany z matematyki. Jego główna właściwość polega na tym, że każdy element znajdujący się w zbiorze musi mieć unikatową wartość. Oznacza to, że zestaw elementów **(1, 2, 3)** jest poprawnym zbiorem, natomiast **(1, 2, 2)** już nie. Ponadto elementy zbioru ułożone są w ustalonej kolejności; domyślnie tworzą ciąg rosnący. Set deklaruje się następujaco:

```
set<typ> nazwa;
```

Na przykład:

```
set<int> nazwaZbioru;
```

Główną zaletą zbioru jest jego architektura, która pozwala na wykonywanie większości działań na strukturze w czasie logarytmicznym względem jego złożoności, czyli **O(nlog(n)**. Elementy dodaje się do zbioru za pomocą komendy **insert()**, natomiast usuwa komendą **erase(element)**. Co więcej element możemy usunąć także poprzez iterator; **erase(iterator)**. Dla przykładu:

```
set<int> nazwaZbioru;
nazwaZbioru.insert(1); // zbiór: {1}
nazwaZbioru.insert(2); // zbiór: {1, 2}
nazwaZbioru.insert(2); // zbiór: {1, 2}
nazwaZbioru.insert(3); // zbiór: {1, 2, 3}
nazwaZbioru.erase (2); // zbiór: {1, 3}
set<int>::iterator iteratorZbioru = nazwaZbioru.begin(); // iterator na początek zbioru
nazwaZbioru.erase(iteratorZbioru); // zbiór: {3}
```

Set oferuje także możliwość wyszukiwania elementów, które się na nim znajdują. Komenda **find(wartość)** zwraca iterator do elementu zbioru o podanej wartości. W przypadku, gdy nie uda się znaleźć elementu, zwracany jest iterator na koniec seta (**nazwaZbioru.end()**). Ponadto istnieją także komendy **lower_bound(wartość)** oraz **upper_bound(wartość)**, które w analogiczny sposób zwracają iterator do pierwszego eementu nie mniejszego, niż zadana wartość oraz pierwszego elementu większego od podanej wartości. Podobnie jak w przypadku **find()**, jeśli wyszukiwanie się nie powiedzie, zwrócony zostanie iterator na koniec seta. Przykładowo:

```
set<int> nazwaZbioru; // niech zbiór prezentuje się następująco: {1, 3, 5, 7, 9}.
nazwaZbioru.find(3) // zwróci iterator na trzeci element zbioru
*nazwaZbioru.find(3) // zwróci wartość 5
*nazwaZbioru.upper_bound(6) // zwróci wartość 7
*nazwaZbioru.upper_bound(5) // zwróci wartość 7
*nazwaZbioru.lower_bound(4) // zwróci wartość 5
*nazwaZbioru.lower_bound(5) // zwróci wartość 5
nazwaZbioru.find(12) // zwróci iterator na koniec zbioru
```

Oczywiście jak we wszystkich powyższych strukturach, set także dysponuje metodami **size()** oraz **empty()**. Dodatkowo posiada także metodę **clear()**, która usuwa ze zbioru wszystkie elementy.

Zbiór umożliwia także zastosowanie własnego komparatora, dzięki czemu można trzymać na nim elementy według interesującej nas, niestandardowej kolejności:

```
set<typ, nazwaKomparatora> niestandardowoUporzadkowanyZbior;
```

2.3.3 Multiset

Multiset jest strukrą niemalże identyczną do zbioru, z tą różnicą że multizbiór potrafi przechowywać wiele elementów o tej samej wartości. Dlatego też zarówno **(1, 2, 3)**, jak i **(1, 2, 2)** są poprawnymi multizbiorami. Udostępnia on metody analogiczne, co zwykły set. Deklaruje się go następująco:

```
multiset<typ> nazwa;
```

Na przykład:

```
multiset<int> nazwaMultizbioru;
```

Warto zaznaczyć, że komenda **erase(wartość)** w tym przypadku usunie z multisetu wszystkie elementy o podanej wartości. W przypadu, gdy interesuje nas usunięcie jedynie jednego elementu, musimy wspomóc się usuwaniem elementu po iteratorze:

```
multiset<int> nazwaMultizbioru; // niech multizbiór prezentuje się następująco: {1, 2, 2, 2, 3}
// ustalmy, że chcemy usunąć tylko jeden element o wartości dwa.
multiset<int>::iterator iteratorMultizbioru = nazwaMultizbioru.find(2); // deklarujemy iterator
// wskazujący na pierwszy z brzegu element dwa
nazwaMultizbioru.erase(iteratorMultizbioru); // usuwamy element 2, na który wskazuje iterator
// teraz multizbiór prezentuje się następujaco: {1, 2, 2, 3}
```

2.3.4 Map

Ostatnią rodziną omawianych struktur będą mapy. Mapa znana jest także pod nazwą słownika albo tabeli asocjacji, gdyż jej zadaniem jest przyporządkowywanie elementów jednego zbioru do elementów innego zbioru. Dla przykładu mapa pozwala nam na przyporządkowanie każdemu elementowi typu **string** przyporządkować element typu **int** odpowiadający jego długości. Podobnie jak pozostałe struktury uporządkowane, jej rozmiar zmienia się dynamicznie, w zależności od liczby elementów, które zawiera, co czyni ją strukturą efektywną pamięciowo. Operacje na niej wykonywane charakteryzują się złożnością **O(log(n))** względem jej rozmiaru. Mapę deklaruje się następująco:

```
map<typKluczy, typWartosci> nazwaMapy;
```

Przykładowo:

```
map<string, int> nazwaMapy;
```

Dodawanie oraz odczytywanie elementów z mapy odbywa się analogicznie, jak w przypadku klasycznej tablicy. Jedyna różnica jest taka, że zamiast podawania indeksu elementu podajemy jego klucz, który zresztą niekoniecznie musi być liczbą. Dla przykładu:

```
map<string, int> nazwaMapy;
nazwaMapy["cztery"] = 4;
nazwaMapy["dwa"] = 2;
nazwaMapy["cztery"] // zwraca wartość 4
```

Ponadto możemy także usuwać elementy z naszej mapy za pomocą komendy **erase()** oraz pytać się czy istnieje element o danym kluczu poprzez metodę **contains()**. Na przykład:

```
map<string, int> nazwaMapy; // niech będzie to mapa z poprzedniego przykładu
nazwaMapy.contains("cztery") // zwraca wartość true
nazwaMapy.contains("trzy") // zwraca wartość false
nazwaMapy.erase("cztery")
nazwaMapy.contains("cztery") // zwraca wartość true
```

2.3.5 Unordered map

Unordered map znana także jako haszmapa jest mapą nieuporządkowaną. Struktura ta swoim działaniem odzwierciedla klasyczną mapę, jednakże jej implementacja różni się znacząco. Zamiast drzew czerwono-czarnych używa on tablicy haszującej, przez co umożliwia ona przeprowadzanie na niej operacji w średnim czasie stałym $\Theta(1)$. Haczyk polega na tym, że w przypadku podania nieuporządkowanej mapie złośliwych danych, złożoność ta moze wzrosnąć do złożoności liniowej względem jej rozmiaru - $O(n)$. Należy więc używać jej z rozwagą. Deklaruje się ją następująco:

```
unordered_map<typKluczy, typWartosci> nazwa;
```

Przykładowo:

```
unordered_map<string, int> nazwaHaszmapy;
```

Zainteresowanych działaniem mapy nieuporządkowanej zachęcam do zapoznania się z tematem haszowania oraz tablicy haszującej.

2.4 Przydatne funkcje

2.4.1 Sort

Jedną z najbardziej przydatnych funkcji, które oferuje STL jest bez wątpienia funkcja **sort()**. Oferuje ona możliwość posortwania tablicy elementów według ustalonej reguły sortowania. Funkcja **sort()** Domyślnie sortuje elementy niemalejąco. Używa się jej następująco:

```
sort(tablica, tablica + rozmiarTablicy);
//ewentualnie w przypadku użycia komparatora
sort(tablica, tablica + rozmiarTablicy, komparator);
```

Dla przykładu:

```
// niech komparatorNierosnacy będzie komparatorem porządkującym elementy nierosnąco
int tab[n]; // niech tab będzie tablicą, którą chcemy posortować, natomiast n jej rozmiarem
sort(tab, tab + n); // funkcja posortuje niemalejąco tablicę tab
sort(tab, tab + n, komparatorNierosnacy) // funkcja posortuje nierosnąco tablicę tab
```

Sort potrafi także sortować między innymi wektory. W tym wypadku składnia jest nieco inna:

```
vector<int> wektor; // niech wektor będzie wektorem, który chcemy posortować
sort(wektor.begin(), wektor.end());
```

2.4.2 Permutation

STL posiada także mechanizm do generowania kolejnych permutacji danego ciągu. Do generowania następnej leksykograficznie permutacji tablicy należy użyć funkcji **next_permutation()**, natomiast do permutacji poprzedniej; komendy **prev_permutation)**. Przykładowo:

```
int tab[5] = {2, 5, 3, 7, 1};
prev_permutation(tab, tab + 5) // po wywołaniu funkcji, tab przyjmie postać {2, 5, 3, 1, 7}
next_permutation(tab, tab + 5) // po wywołaniu funkcji, tab przyjmie postać {2, 5, 7, 1, 3}
```

Funkcje te posiadają także wartość zwracaną. W przypadku, gdy istnieje permutacja poprzednia / następna, to zwracają one wartość true, w przeciwnym wypadku false. Na przykład:

```
int tab[5] = {1, 2, 3, 5, 4};
next_permutation(tab, tab + 5) // zwróci true
//w tym momencie tab prezentuje się następująco: {1, 2, 3, 4, 5}
next_permutation(tab, tab + 5) // zwróci false, ponieważ nie ma następnej permutacji
```

Ciekawostką jest, że istnieje także trzecia komenda **random_shuffle()**, która dokonuje losowej permutacji ciągu wartości.

3 Podstawowe zagadnienia

3.1 Sumy prefiksowe

Dany jest ciąg n liczb. Naszym zadaniem jest odpowiadanie na pytania o sumę wartości elementów ciągu na danym przedziale. Na poniższym przykładnie pytania mają formę **a b**, gdzie a, b są krańcami przedziałów oraz $a \leq b$:

```
ciąg = {1, 6, 3, 8, 4, 4};
2 3 -> 6 + 3 = 9
2 4 -> 6 + 3 + 8 = 17
4 5 -> 8 + 4 = 12
1 6 -> 1 + 6 + 3 + 8 + 4 + 4 = 26
3 6 -> 3 + 8 + 4 + 4 = 19
```

3.1.1 Podejście naiwne

Spróbujmy rozwiązać problem naiwnie; nie przejmując się złożonością. Dla każdego zapytania będziemy sumować wartości elementów po kolei, analogicznie jak w powyższym przykładzie. Poniżej znajduje się algorytm rozwiązujący problem w ten sposób:

```
for i := 1 to liczba_zapytan do
    read a, b;
    sum := 0;
    for j := a to b do
        sum += ciąg[j];
    done
    print sum;
done
```

Zastanówmy się nad złożonością powyższego rozwiązania. Oznaczmy długość ciągu przez n, natomiast liczbę zapytań jako k. Zauważmy, że w najgorszym przypadku, czyli zapytaniu o sumę całego ciągu, wewnętrzna pętla będzie musiała wykonać się n razy. Daje to Złożoność $O(nk)$, co w przypadku, gdy $n \sim k$, jest złożonością kwadratową. Spróbujmy lepiej.

3.1.2 Definicja

Suma prefiksowa jest niczym innym, jak sumą wartości elementów prefiksu (spójnego podciągu początkowych elementów) ciągu o danej długości. Dla przykładu sumą prefiksową długości 3 nazwiemy sumę wartości trzech pierwszych elementów ciągu. W szczególności (dla ciągu o długości n) suma prefiksowa długości n będzie zwyczajnie sumą wartości elementów całego ciągu.

3.1.3 Obliczanie sum prefiksowych

Zanim przystąpimy do obliczania sum prefiksowych zauważmy następujące cechy:

- sumą prefiksową długości 1 jest po prostu wartość pierwszego elementu ciągu,
- znając sumę prefiksową długości k jesteśmy w stanie obliczyć sumę prefiksową długości $k + 1$ poprzez dodanie wartości elementu $k + 1$ do sumy k-elementowej.

Dzięki powyższemu podejściu jesteśmy w stanie obliczyć sumy prefiksowe wszystkich długości w czasie liniowym:

```
pref[0] := ciąg[0]; // stosujemy cechę 1
for i := 1 to długość_ciągu do
    pref[i] := pref[i - 1] + ciąg[i]; // stosujemy cechę 2
done
```

3.1.4 Podejście wzorcowe

Spróbujmy rozwiązać nasz pierwotny problem wykorzystując sumy prefiksowe. Spróbujmy więc polepszyć złożoność całego rozwiązania. Zastanówmy się jak wykorzystać sumy prefiksowe bezpośrednio do obliczania sum na dowolnym przedziale. Przykładowo rozważymy przdział $[a; b]$. Jedynym ograniczeniem jakie tak na prawdę napotykamy jest fakt, że suma prefiksowa z definicji musi zaczynać się od początku przedziału. Odczytując sumę pierwszych b elementów otrzymamy wartość za dużą. O ile? Dokładnie o sumę pierwszych $a-1$ elementów. Poprzez odjęcie od sumy prefiksowej b elementowej sumy $a-1$ elementowej otrzymamy dokładnie to co nas interesuje, czyli sumę elementów na przedziale od a do b. Ponadto dzięki temu, że wcześniej obliczyliśmy wartości wszystkich sum prefiksowych, to jesteśmy w stanie udzieliś odpowiedzi w czasie stałym! Daje to nam sumaryczną złożoność rzędu $O(n+k)$; w przypadku, gdy $n \sim k$ będzie ona liniowa. Poniżej znajduje się algorytm wzorcowy rozwiązujący problem:

```
pref[0] := ciąg[0];
for i := 1 to długość_ciągu do
    pref[i] := pref[i - 1] + ciag[i];
done
for i := 1 to liczba_zapytan do
    read a, b;
    a--; // tablice indeksujemy od zera
    b--;
    if a == 0 then // w tym przypadku nie mamy co odejmować
        print pref[b];
    else
        print pref[b] - pref[a - 1];
    endif
done
```

3.2 Metoda dwóch wskaźników

Wyobraźmy sobie, że posiadamy pewien ciąg liczb całkowitych, dla którego interesuje nas maksymalna suma elementów jego spójnego podciągu. Zadanie to jest trywialne, jeśli ciąg zawiera same liczby nieujemne, gdyż wtedy odpowiedzią jest oczywiście suma całego ciągu. Prawdziwy problem pojawia się, gdy rozważaniom poddamy ciąg zawierający także liczby ujemne. Przeanalizujmy poniższy przykład:

```
ciąg = {5, -6, 7, 3, -4, -5, 10};
// biorąc pod uwagę cały ciąg otrzymamy sumę 10.
// biorąc pod uwagę jedynie podciąg {7, 3, -4, -5, 10} suma ta wzrośnie do 11.
```

3.2.1 Podejście naiwne

Na początek spróbujmy rozwiązać dany problem możliwie prosto. Pomysłem, który zazwyczaj nasuwa się jako pierwszy jest policzenie sum wszytkich możliwych spójnych podciągów, a następnie wybranie tej największej. Spróbujmy zaprojektować program realizujący tę ideę:

```
max_sum := 0; // pusty podciąg gwarantuje nam, że suma będzie przynajmniej nieujemna
for i := 0 to długość_ciągu do // dla każdego początku ciągu
    current_sum = 0;
    for j := i to długość_ciągu do // dla każdego końca ciągu
        current_sum += ciąg[j];
        max_sum = max(max_sum, current_sum); // zapisujemy największą dotychczas znalezioną sumę
    done
done
print max_sum;
```

W praktyce obie pętle wykonają sumarycznie $1+2+3+...+n = \frac{n(n+1)}{2}$ operacji, co wskazuje na złożoność rzędu $O(n^2)$. Spróbujmy wymyślić lepsze rozwiązanie.

3.2.2 Definicja

Metoda dwóch wskaźników polega na przemierzaniu ciągu przy pomocy dwóch wskaźników tworzących swojego rodzaju przedział zawierający wszystkie aktualnie rozpatrywane elementy. Same wskaźniki zwane najczęściej **lewym** i **prawym** mają następujące własności:

- wskaźniki poruszają się tylko do przodu,
- wskaźnik lewy nie może wskazywać na element o większym indeksie, niż wskaźnik prawy,

W praktyce oznacza to, że decydując się na przesunięcie prawego wskaźnika, aktualnie rozpatrywany ciąg zwiększy swój rozmiar o nowy element. Analogicznie przesunięcie w prawo wskaźnika lewego spowoduje zmniejszenie rozmiaru podciągu o element dodany najwcześniej. Kluczem do poprawnego wykorzystania tej metody jest prawidłowe podejmowanie decyzji o wyborze wskaźnika, którym planujemy poruszyć.

3.2.3 Metoda dwóch wskaźników - w praktyce

Na początek spróbujmy przećwiczyć działanie nowo poznanej metody na najprostszym możliwym przykładzie. Załóżmy, że posiadamy funkcję **ktory(l, r)**, która przyjmując aktualne wartości wskaźników lewego i prawego podejmuje decyzję, którym wskaźnikiem należy w danym momencie poruszyć. Przeanalizujmy jak dla danego przykładu prezentuje się algorytm metody dwóch wskaźników:

```
l := 0;
r := 0; // na start ustawiamy wskaźniki na początku ciągu.
while l < długość ciągu do // nie przejmujemy się prawym wskaźnikiem,
                           // gdyż i tak będzie on po prawej stronie
if ktory(l, r) == 'r' && r < długość_ciągu then
        r++; // ruszamy prawym wskaźnikiem
        // w tym miejscu możemy chcieć zaktualizować rozpatrywany podciąg
    else
        l++; // ruszamy lewym wskaźnikiem
        // w tym miejscu możemy chcieć zaktualizować rozpatrywany podciąg
    endif
    // w tym miejscu możemy podsumować wynik otrzymany dla aktualnie rozpatrywanego podciągu
done
```

3.2.4 Podejście wzorcowe

Sprawdźmy czy metoda dwóch wskaźników pomoże nam w rozwiązaniu naszego pierwotnego problemu. Kluczową obserwacją jest tutaj fakt, że skoro suma elementów pustego podciągu wynosi 0, to jeżeli dotychczasowo rozpatrywana suma jest ujemna, możemy równie dobrze nie brać jej pod uwagę i zacząć składać nasz podciąg od nowa, na pewno nie pogarszając rozwiązania. Dzięki temu możemy opracować schemat ruszania wskaźnikami:

- jeśli suma jest nieujemna (i prawy wskaźnik może się jeszcze ruszyć) $\rightarrow$ rusz wskaźnikiem prawym,
- jeśli suma jest ujemna (i lewy wskaźnik może się jeszcze ruszyć) $\rightarrow$ rusz wskaźnikiem lewym (w przypadku, gdy wskaźnik lewy jest równy prawemu, należy ruszyć obydwa wskaźniki, aby nie złamać postulatów definicji metody),
- jeśli nie można się ruszyć wskaźnikiem lewym $\rightarrow$ skończ algorytm.

Dzięki takiemu podejściu na bieżąco będziemy obcinać niedodatnie prefiksy podciągu oraz aktualizować go o nowe elementy, które potencjalnie mogą przyczynić się do zwiększenia sumy jego elementów. Nie pominiemy żadnego kandydata na podciąg o maksymalnej sumie, gdyż usuwane prefiksy jedynie pogarszałyby potencjalny wynik. Zastanówmy się teraz nad złożonością tego rozwiązania. Jako że ruszamy dwoma wskaźnikami w tylko jednym kierunku, to każdy z nich przeiteruje się po każdym elemencie ciągu dokładnie raz. Daje to złożoność liniową względem rozmiaru ciągu, a więc $O(n)$. Poniżej znajduje się pseudokod rozwiązania:

```
l := 0;
r := 0;
max_sum := max(0, ciąg[0]); // punktem wyjściowym jest max z sumy wartości ciągu
                            // pustego oraz samego elementu początkowego
current_sum := max(0, ciag[0]);
while l < długość_ciągu do
    if current_sum >= 0 && r < długość_ciągu then
        r++;
        current_sum += ciag[r];
    else if l == r then // przypadek, w którym musimy poruszyć jednocześnie oba wskaźniki
        r++;
        l++;
        current_sum := ciag[l];
    else
        current_sum -= ciag[l];
        l++;
    endif
    max_sum := max(max_sum, current_sum);
done
print(max_sum);
```

3.3 Wyszukiwanie binarne

Dany jest **monotoniczny** ciąg n liczb. Chcielibyśmy dowiedzieć się jaka jest największa wartość elementu ciągu, która nie przekracza k. Na potrzeby naszych rozważań załóżmy bez straty ogólności, że ciąg ten jest niemalejący.

3.3.1 Podejście naiwne

Na początek spróbujmy rozwiązać problem w najprostszy sposób. Możemy przeiterować się po każdym elemencie ciągu, sprawdzając, że jest on niewiększy, niż k, a następnie w przypadku odpowiedzi pozytywnej, sprawdzać, czy jest to największy takowy element. Ideę realizuje poniższy kod:

```
max_element := -inf; // na początku nie mamy żadnego kandydata
for i := 0 to długość_ciągu do
    if ciąg[i] <= k then
        max_element = max(max_element, ciąg[i]);
    endif
done
print(max_element);
```

Dodatkowym krokiem, który możemy uczynić w celu polepszenia średniego czasu działania naszego algorytmu jest skorzystanie z faktu, że ciąg jest monotoniczny:

```
max_element := -inf;
for i := 0 to długość_ciągu do
    if ciąg[i] <= k then
        max_element = max(max_element, ciag[i]);
    else
        break; // pozostałe elementy ciągu będą większe od k, więc nie trzeba ich sprawdzać
    endif
done
print(max_element);
```

Mimo zastosowanej optymalizacji, która istotnie poprawia czas działania programu na danych losowych, złożoność pesymistyczna pozostaje niezmieniona i wynosi **O(n)**, gdzie n jest długością ciągu. Okazuje się, że monotoniczność ciągu pozwala na znaczącą poprawę tego wyniku.

3.3.2 Definicja

Wyszukiwanie binarne jest podejściem rekurencyjnym, którego idea polega na zawężaniu obszaru poszukiwań. W każdym wywołaniu rekurencyjnym wybierany jest element znajdujący się w połowie przeszukiwanego obszaru (zwany często jako **pivot**). Na podstawie wartości tego elementu oraz faktu monotoniczności ciągu, który jest przeszukiwany, podejmowana jest decyzja odnośnie zakresu przedziału dalszych poszukiwań (wybierany jest przedział na prawo bądź na lewo od pivota). Algorytm kończy się, gdy przedział zawęzi się do pojedynczego elementu, którego szukamy, albo w przypadku gdy takiego elementu nie ma.

3.3.3 Wyszukiwanie binarne - w praktyce

Prześledźmy działanie wyszukiwania binarnego w praktyce. Załóżmy, że posiadamy pewną funkcję o nazwie **ktory(p)**, która podejmuje decyzję o wyborze przedziału na podstawie wartości elementu znajdującego się po środku przedziału (pivota). Dodatkowo niech funkcja **szukany(e)** odpowiada ma pytanie czy element e jest szukanym elementem:

```
function binSearch(a: int, b: int): int
begin
    if a == b then
        if szukany(a) then
            return a;
        else
            return -1; // -1 oznacza, że szukany element ciągu nie istnieje
        endif
    endif
    int p := (a + b) / 2; // definiujemy pivot
if ktory(ciąg[p]) == "lewy" then
        return binSearch(a, p);
    else
        return binSearch(p, b);
    endif
end;

//W metodzie wywołującej:
print(binSearch(0, długość_ciągu - 1));
```

3.3.4 Podejście wzorcowe

Zastosujmy wyszukiwanie binarne w celu rozwiązania problemu z początku tego rozdziału. Skoro wyjściowy ciąg jest monotoniczny (niemalejący), to możemy przeprowadzić na nim wyszukiwanie binarne. Decyzję o doborze następnego rozpatrywanego przedziału podejmować będziemy na podstawie stosunku pivota do k. Jeśli pivot jest niewiększy, to wybierzemy przedział lewy, w przeciwnym wypadku przedział prawy. Poniżej przedstawiona została forma algorytmu wyszukiwania binarnego rozwiązująca zadany problem:

```
function binSearch(a: int, b: int): int
begin
    if a == b then
       return a;
    endif
    int p := (a + b) / 2;
if ciąg[p] > k
        return binSearch(a, p - 1);
```

```
    else
        return binSearch(p, b);
    endif
end;
```

Na koniec określmy złożoność wyszukiwania binarnego. Każde wywołanie rekurencyjne powoduje dwukrotne zmniejszenie przedziału poszukiwań. Złożoność ta rośnie więc odwrotnie do funkcji wykładniczej, czyli logarytmicznie. Oznacza to, że dla ciągu długości n, wyszukiwanie binarne będzie miało złożoność **O(log(n))**, co czyni ten algorytm bardzo efektywnym.

3.4 Sortowanie kubełkowe

Naszym dzisiejszym problemem będzie problem znajdowania dominanty pewnego ciągu. Dominanta jest niczym innym, jak elementem, który występuje w ciągu najczęściej. Dodatkowo niech elementy tego ciągu przyjmują jedynie wartości z przedziału $[1; 1000]$ Dla sprecyzowania załóżmy, że szukamy najmniejszego takiego elementu.

3.4.1 Podejście naiwne

Spróbujmy rozwiązać problem sortując ciąg niemalejąco, a następnie zliczając długości podciągów zawierających równe sobie elementy wyznaczmy najdłuższy z nich:

```
sort(ciąg, ciąg + długość_ciągu);
curr_streak := 1; // zakładamy, że pierwszy element jest już rozpatrzony
max_streak := 0;
for i := 1 to długość_ciągu do
    if ciąg[i] == ciąg[i - 1] then
        curr_streak++; //jeśli podciąg dalej trwa, to uwzgnędniamy kolejny element
    else // jeśli podciąg się zakończył, to podsumowujemy uzyskany wynik
        max_streak = max(max_streak, curr_streak);
        curr_streak++; // rozpatrywany element uwzględniamy w wyniku dla następnego ciągu
    endif
endfor
max_streak = max(max_streak, curr_streak); // podsumowujemy ostatni podciąg już poza pętlą
print(max_streak);
```

Złożoność rozwiązania jest narzucona przez algorytm sortujący. Funkcja **sort()** z biblioteki STL sortuje w czasie **O(nlog(n))**, co jednocześnie jest złożonością całego rozwiązania.

3.4.2 Definicja

Sortowanie kubełkowe jest specyficznym rodzajem algorytmu sortującego. Jego unikalność wynika z faktu, że złożonośc sortowania kubełkowego wynika nie tylko z rozmiaru sortowanego ciągu, ale także z rozmiaru jego przeciwdziedziny. Algorytm polega na tym, że każdej możliwej wartości elementu ciągu przypisuje się kubełek, który początkowo jest pusty. Następnie iterując się po elementach ciągu, inkrementuje się wartości w kubełkach odpowiadającym poszczególnym elementom. Następnie rozpatrując wszystkie kubełki w danej kolejności odczytuje się odpowiadającą im wartość tyle razy, ile wynosi liczba zapisana w kubełku. Podane rozwiązanie przegląda każdy element ciągu tylko raz, więc dla długości ciągu równej n ta część rozwiązania posiada złożonośc liniową względem n, czyli **O(n)**. Pozostała część wymaga wykonania pewnej stałej liczby operacji dla każdego kubełka. Jeśli kubełków (elementów przeciwdziedziny) jest S, to pozostała część algorytmu wykona się w złożoności **O(S)**, to daje sumaryczną złożoność rzędu **O(n + S)**. Zauważmy, że dla długich ciągów o małym zakresie wartości złożoność ta wypada lepiej, niż **O(nlog(n))** oferowane przez klasyczne algorytmy sortujące.

3.4.3 Sortowanie kubełkowe - w praktyce

Załóżmy, że rozpatrywany ciąg posiada elementy z zakresu od 0 do $S-1$. Prześledźmy algorytm sortowania kubełkowego w praktyce:

```
for i := 0 to S do
    cup[i] = 0; // przygotowujemy puste kubełki
done
for i := 0 to długość_ciągu do
    cup[ciąg[i]]++; // rejestrujemy wystąpienie elementu w odpowiadającym mu kubełku
done
for i :=0 to S do
    for j := 0 to cup[S] do // wypisujemy tyle elementów ile jest w danym kubełku
        print i;
    endfor
endfor
```

3.4.4 Podejście wzorcowe

Wracając do pierwotnego problemu, spróbujmy zastosować sortowanie kubełkowe. Sama zamiana funkcji odpowiadającej za sortowanie poprawi złożoność do $\mathbf{O(n + S)}$. Można jednak rozwiązać ten problem jeszcze prościej. Zamiast zwracać sortowany ciąg, możemy wykorzystać fakt, że kubełki zawierają w sobie liczbę elementów o danej wartości i bezpośrednio stąd odczytać dominantę:

```
max_streak := 0; // największa dotychczas znaleziona dominanta
for i := 0 to S do
    cup[i] = 0;
done
for i := 0 to długość_ciągu do
    cup[ciąg[i]]++;
done
for i := 0 to S do
    max_streak = max(max_streak, cup[i]); // sprawdzamy czy wartość odpowiadająca danemu
                                          // kubełkowi jest nowym kandydatem na dominantę
endfor
print(max_streak);
```

3.4.5 Wada sortowania kubełkowego

Na koniec warto podkreślić główną wadę sortowania kubełkowego. Jest nią zależność od rozmiaru danych. Aby trafniej nakreślić dlaczego jest to aż tak istotne rozpatrzmy przykład, w którym do posotowania mamy ciąg składający się z dwóch elementów: 1 oraz 10^{18}. Dla klasycznego algorytmu jest to trywialne zadanie, jednakże algorytm sortowania kubełkowego musiałby stworzyć 10^{18} kubełków, co nie jest wykonalne na nawet najlepszych komputerach. Z tego powodu algorytm ten nie jest szczególnie popularny i znajduje zastosowanie tylko dla specyficznych danych.

3.5 Kolejka maksimów

Z pewnością każdy pamięta kolejkę priorytetową z rozdziału o bibliotece STL. Dla przypomnienia była to struktura danych posiadająca następującą właściwość: każdorazowo w jej wierzchołku przechowuje ona element o największej wartości. Ponadto jest to jedyny element, który można z takiej kolejki usunąć, jednocześnie zastępując go elementem następnym co do wielkości. Z drugiej strony mamy klasyczną kolejką będącą implementacją modelu FIFO - najwcześniej dodany element opuszcza strukturę jako pierwszy. Załóżmy, że interesuje nas pewne rozwiązanie hybrydowe; struktura zachowująca właściwości modelu FIFO, jednakże potrafiąca każdorazowo zwracać wartość najmniejszego (albo największego) elementu, który się w niej znajduje.

3.5.1 Podejście naiwne

Podejdźmy do problemu w możliwie prosty sposób: spróbujmy trzymać aktualną wartość maksymalną w zmiennej. Możemy ja bezproblemowo aktualizować przy każdym dodaniu elementu oraz odpytywać o nią w czasie stałym. Problem niestety pojawia się, gdy przychodzi czas na usunięcie elementu o tej wartości. Jak wybrać następny element maksymalny? Nasuwającym się rozwiązaniem jest przeiterowanie się po całej strukturze w celu ustalenia nowego maksimum. Niestety w przypadku, gdy na kolejce umieścimy elementy w kolejności malejącej, a następnie zniczniemy je usuwać, to każde usunięcie zajmie nam liniowo dużo czasu. Finalnie otrzymamy więc złożoność $O(n^2)$, gdzie n jest liczbą operacji.

3.5.2 Definicja

Kolejka maksimów (minimów) jest strukturą łączącą operacje dodawania oraz usuwania elementów znane z klasycznej kolejki, wraz z odczytywaniem elementu zawartego w jej wierzchołku zgodnie z zasadami panującymi w kolejce priorytetowej. Innymi słowy kolejka zawsze będzie usuwała elemnt dodany najwcześniej oraz zapytana o wartość wierzchołka będzie zwracała element o największej wartości. Przykładowo:

```
//załóżmy, że kolejka posiada wierzchołek po prawej stronie
kolejka.push(4) // (4)
kolejka.push(1) // (1, 4)
kolejka.push(3) // (3, 1, 4)
kolejka.push(2) // (2, 3, 1, 4)
kolejka.front() // 4
kolejka.pop()   // (2, 3, 1)
kolejka.front() // 3
kolejka.pop()   // (2, 3)
kolejka.front() // 3
```

3.5.3 Kolejka maksimów - w praktyce

Kolejka maksimów opiera się na jednym prostym fakcie; w momencie, kiedy dodamy nowy element, to będzie on wartością maksymalną tak długo, aż nie zostanie usunięty ze struktury albo nie pojawi się element o wartości większej od jego. Wynika z tego, że wszystkie elementy dodane do wcześniej do struktury nie mają znaczenia; ich wartość nie zostanie nigdy użyta! Jedyną istotną informacją jest liczba takowych elementów. W momencie dodanie nowego maksimum możemy więc skompresować wszystkie elementy poprzedzające do licznika reprezentującego ich aktualną liczbę. Zauważmy, że powyższa cecha nie tyczy się tylko elementu maksymalnego: każdy nowododany element, który jest niemniejszy, niż pewna liczba elementów dodanych przed nim może je kompresować na identycznej zasadzie. Zastanówmy się jak będzie wyglądała nasza struktura. Każdy element kompresuje wszystkie elementy niewiększe od niego, które zostały dodane wcześniej. Oznacza to, że pierwszy wcześniejszy, nieskompresowany element będzie niemniejszy. Tworzą więc one ciąg monotoniczny, w którym im wcześniej element został dodany, tym ma większą (niemniejszą) wartość. Oznacza to, że usuwając element ze struktury kolejnym maksimum będzie następny element na kolejce, co rozwiązuje pierwotny problem.

3.5.4 Podejście wzorcowe

Jako struktury użyjemy znanej z rozdziału o bibliotece STL bikolejki. Na strukturze trzymać będziemy pary elementów; pierwszy z nich będzie reprezentował wartość, natomiast drugi liczbę elementów przez niego skompresowanych.

- Operację dodania elementu zrealizujemy w następujący sposób: na początku zliczymy liczbę elementów, które zamierzamy skompresować tj. elementów niewiększych od dodawanego, równocześnie je usuwając. Następnie dodamy nowy element wraz z liczbą skompresowanych przez niego elementów.
- Operację usuwania elementu możemy wykonać poprzez zmniejszanie liczby skompresowanych elementów w wierzchołku kolejki. W przypadku, gdy liczba ta będzie równa zero, usuwamy element znajdujący się w wierzchołku.

- Odczyt wartości największej będzie realizowany poprzez zwykły odczyt wartości elementu znajdującego się w wierzchołku.

Poniżej przedstawiono realizację poszczególnych operacji w postaci funkcji:

```
procedure push(kolejka: deque<pair<int, int> >, x: int)
begin
    cnt := 0;
    while !kolejka.empty() and kolejka.back().x >= x do
        cnt := cnt + kolejka.back().cnt + 1;
        kolejka.pop_back();
    done
    kolejka.push_back({x, cnt});
end;

procedure pop(kolejka: deque<pair<int, int> >)
begin
    if kolejka.front().cnt == 0 then
        kolejka.pop_front();
    else
        kolejka.front().cnt := kolejka.front().cnt - 1;
    endif
end;

function front(kolejka: deque<pair<int, int> >)
begin
    return kolejka.front().x;
end;
```

Na koniec wypadałoby zastanowić się nad złożonością rozwiązania. Funkcje **pop()** oraz **front()** z pewnością działają w czasie stałym. Procedura **push()** zawiera pętle, co sugeruje, że jej działanie przebiega w czasie liniowym. Spójrzmy jednak na szerszy obraz; załóżmy, że wykonamy n operacji (niekoniecznie tylko typu push). Zliczając iteracje pętli możemy dojść do wniosku, że za każdym razem usuwa ona jeden element z naszej struktury. Może więc wykonać się co najwyżej n razy. Oznacza to, że pomimo możliwości wykonania $O(n)$ operacji naraz, pętla ta będzie działała w **zamortyzowanym średnim czasie stałym**, to oznacza że sama struktura mieści się w złożoności $O(n)$!

3.6 Liczby pierwsze

Definicja zbioru liczb pierwszych jest znana chyba wszystkim. W algorytmice, podobnie zresztą jak w pozostałych dziedzinach ścisłych znajdują one dużo zastosowań. Samo ich wykorzystanie jest głównie uzależnione od problemu z jakim mamy do czynienia, jednakże aby takowe liczby zastosować, potrzebujemy je pierw wyznaczyć. Dokładniej dla danej liczby p chcielibyśmy znać efektywny sposób na stwierdzenie czy jest ona pierwsza.

3.6.1 Podejście naiwne

W pierwszym podejściu skorzystamy wprost z definicji liczby pierwszej. Skoro jest to liczba, która dzieli się tylko przez jeden i samą siebie, to z praw De Morgana możemy przekształcić równoważnie ten warunek jako liczba, która nie posiada dzielników innych, niż jeden lub ona sama. Jesteśmy w stanie to stwierdzić przy pomocy następującego fragmentu kodu:

```
function pierwsza(x: int)
begin
    if x == 1 then
        return false; // jeden nie jest liczbą pierwszą
```

```
    endif
    for i := 2 to x - 1 do
        if x % i == 0 then
            return false; // liczba nie jest pierwsza, gdyż dzieli się przez i
        endif
    done
    return true; // liczba dzieli się tylko przez jeden i samą siebie, więc jest pierwsza
end;
```

Złożoność takiego podejścia jest bez wątpienia liniowa.

3.6.2 Podejście optymalne

W kolejnym podejściu wykorzystamy pewną prostą właściwość podzielności; jeśli liczba b dzieli liczbę a bez reszty, to liczba $c = a/b$ także dzieli a bez reszty. Dowód jest trywialny, więc został pozostawiony jako ćwiczenie dla czytelnika. Co daje powyższa właściwość? Zastanówmy się nad przeciwdziedziną liczb a oraz b. Niech $b \in [1; \sqrt{a}]$. Implikuje to, że $c \in [\sqrt{a}; a]$. W połączeniu z powyższą właściwością oznacza to, że testując dzielniki z przedziału $[1; \sqrt{a}]$ jednocześnie testujemy wszystkie dzielniki z przedziału $[\sqrt{a}; a]$. Możemy więc ograniczyć obszar poszukiwań do pierwszych $\sqrt{a}$ wartości. Polepsza to naszą złożoność do złożoności pierwiastkowej ($\mathbf{O(\sqrt{n})}$):

```
function pierwsza(x: int)
begin
    if x == 1 then
        return false; // jeden nie jest liczbą pierwszą
    endif
    for i := 2 to sqrt(x) do
        if x % i == 0 then
            return false; // liczba nie jest pierwsza, gdyż dzieli się przez i (oraz x / i)
        endif
    done
    return true; // liczba dzieli się tylko przez jeden i samą siebie, więc jest pierwsza
end;
```

3.6.3 Dalsze optymalizacje

Okazuje się, że podejście to można jeszcze bardziej zoptymalizować. Nie będzie to jednak optymalizacja złożoności, a jedynie średniego czasu działania. Mimo to jest warta uwagi, gdyż zmiany są trywialne, a efekty zauważalne. W pierwszej kolejności zauważmy, że jedyną parzystą liczbą pierwszą jest liczba 2. Pozostałe liczby parzyste nie mogą być pierwsze. Nic nie stoi na przeszkodzie, aby rozpatrzyć liczbę 2 osobno, natomiast dalsze operacje przeprowadzać jedynie na liczbach nieparzystych. Pozwala to nam na dwukrotną redukcję kandydatów na dzielniki! Kolejną optymalizacją z pewnością będzie usunięcie ciężkiego obliczeniowo liczenia pierwiastka stanowiącego granicę naszego przedziału poszukiwań. W tym celu możemy skorzystać z następującej właściwości (warto zaznaczyć, że nasze liczby są dodatnie!): $i \leq \sqrt{x} \iff i * i \leq x$. Z pozoru nie wygląda to na lepsze rozwiązanie, jednakże komputer radzi sobie z nim istotnie szybciej. Warto zaznaczyć, że rozszerza to naszą przeciwdziedzinę, więc koniecznym może okazać się dostosowanie odpowiedniego zakresu zmiennych poprzez na przykład zastosowanie typu **long long**. Ostatecznie nasza funkcja prezentuje się następująco:

```
function pierwsza(x: int)
begin
    if x == 1 then
        return false; // jeden nie jest liczbą pierwszą
    endif
    if x == 2 then
        return true; // dwa jest liczbą pierwszą
```

```
    endif
    for i := 3; i * i <= x; i += 2 do
        if x % i == 0 then
            return false; // liczba nie jest pierwsza, gdyż dzieli się przez i (oraz x / i)
        endif
    done
    return true; // liczba dzieli się tylko przez jeden i samą siebie, więc jest pierwsza
end;
```

Na końcu warto wspomnieć o istnieniu rozwiązań heurystycznych, które z bardzo wysokim prawdopodobieństem są w stanie określić czy liczba jest pierwsza, działając w złożoności nawet $O(\sqrt[4]{n})$. Przykładem takiego algorytmu jest **algorytm Rho Pollard'a**.

3.6.4 Podejście alternatywne

W problemach algorytmicznych często możemy spotkać się z potrzebą wielokrotnego przeprowadzania testu pierwszości liczb. W przypadku, gdy liczba liczb jest duża, natomiast ich wartości pochodzą z odpowiednio ograniczonego przedziału, lepszym pomysłem może okazać się w pierwszej kolejności przeprowadzenie testu pierwszości dla całego zakresu wartości, a następnie odpowiadanie na zapytania w stałym czasie. Wydawać się może, że jest to zbędny nakład obliczeniowy, jednakże w przypadku masowych testów pierwszości okazuje się, że ich złożoność znacząco się redukuje.

3.6.5 Sito Eratostenesa

Sito Eratostenesa zwane także jako sito liczb pierwszych jest algorytmem służącym do testowania pierwszości liczb z przedziału $[1; n]$. Jego działanie rozpoczynamy od tablicy wartości typu bool, której wszystkie komórki wypełnione są wartością true. Rozpartywać będziemy po kolei komórki o indeksach od 2 do n (dla wszystkich poprzednich możemy od razu zapisać wartosc false). Zasada jest prosta: jeśli aktualnie rozpatrywana komórka zawiera wartość false, to ją pomijamy. W przeciwnym wypadku dla komórek o indeksach będących wielokrotnością indeksu obecjen komórki ustawiamy wartość false. Okazuje się, że po zakończeniu algorytmu jedynie komórki o indeksach będących liczbami pierwszymi będą posiadały wartosc true. Dalczego? Każdorazowo gdy odwiedziamy komórkę o wartości true oznacza to, że żadna z poprzednich liczb nie była jej dzielnikiem, gdyż w przeciwnym wypadku jej wartość uległa by zmianie na false podczas procesu wykreślania wielokrotności. Skoro rozpatrywana liczba jest pierwsza, to możemy wykreślić jej wszystkie wielokrotności, gdyż one z pewnością nie będą pierwsze, itd. Komórek z wartoścta false nie rozpatrujemy, gdyż ich wielokrotności zostały i tak wykreślone poprzez liczby pierwsze, z których, na mocy podstawowego twierdzenia arytmetyki, jest ona złożona. Algorytm prezentuje się następująco:

```
pierwsza[0] := false;
pierwsza[1] := false;
for i := 2 to n do
    pierwsza[i] = true;
done
for i := 2 to n do
    if pierwsza[i] == true then
        j := i + i; //pierwsza wielokrotność i
        while j <= n do
            pierwsza[j] = false; // wykreślamy wielokrotności
            j += i; // przechodzimy do następnej wielokrotności
        done
    endif
done
```

Na koniec zastanówmy się nad złożonością sita. Okazuje się, że pętla while wykona się r razy, gdzie $2^{2^r} = n$ (zrozumienie sensu równania pozostawiono jako ćwiczenie dla czytelnika). Po przekształceniu okazuje się, że $r = log_2(log_2(n))$, a więc finalna złożonośc wynosi $O(nlog(log(n))$, co dla wielkości danych użytych w zadaniach olimpijskich czyni ją praktycznie liniową.

3.7 Arytmetyka modulo

Operacja modulo jest operacją binarną, której wynikiem jest reszta z dzielenia pierwszego operandu przez drugi. Używając standardowej notacji matematycznej możemy zdefiniować operację modulo w następujący sposób: $a \equiv b \ (mod\ c) \iff [\frac{a}{c}] * c = b$, gdzie $[x]$ jest podłogą z liczby x.

3.7.1 Właściwości pierścieni modulo

W pierścieniu modulo p prawdziwe są następujące tożsamości:

- dodawanie: $a \ (mod\ p) + b \ (mod\ p) \equiv (a + b) \ (mod\ p)$,
- odejmowanie: $a \ (mod\ p) - b \ (mod\ p) \equiv (a - b) \ (mod\ p)$,
- mnożenie: $a \ (mod\ p) * b \ (mod\ p) \equiv (a * b) \ (mod\ p)$.

Niestety nie wszystko przebiega analogicznie do pierścieni liczb całkowitych czy reczywistych. Oto lista tożsamości, które niestety nie są prawdziwe:

- dzielenie: $a \ (mod\ p)/b \ (mod\ p) \not\equiv (a/b) \ (mod\ p)$,
- potęgowanie: $a^{b \ (mod\ p)} \ (mod\ p) \not\equiv (a^b) \ (mod\ p)$.

Okazuje się, że istnieją sposoby na poradzenie sobie z tymi problemami.

3.7.2 Małe Twierdzenie Fermata

Pomocnym narzędziem okazuje się być **Małe Twierdzenie Fermata (MTF)**. Brzmi ono następująco:

$$a^p \equiv a \ (mod\ p)$$

W dalszych rozważaniach załóżmy, że liczba p jest pierwsza. Czemu? Jest to warunek wystarczający do tego, aby w pierścieniu $mod\ p$ istniała odwrotność elementu a tj. element a^{-1}. Dla dociekliwych; warunkiem koniecznym jest to, aby $NWD(a, p) = 1$ tj. a oraz p były względnie pierwsze. Korzystając z faktu, że klasyczne działanie mnożenia jest zachowane nic nie stoi na przeszkodzie, aby pomnożyć obydwie strony naszej kongruencji przez a^{-1}. Otrzymamy wtedy:

$$a^{p-1} \equiv 1 \ (mod\ p)$$

Natomiast po kolejnej takiej operacji:

$$a^{p-2} \equiv a^{-1} \ (mod\ p)$$

3.7.3 Potęgowanie

Jak dobrze już wiemy $a^{b \ (mod\ p)} \ (mod\ p) \not\equiv (a^b) \ (mod\ p)$, jednakże korzystając z przekształconej fromy MTF możemy rozpisać prawą stronę kongruencji w nastepujący sposób:

$$(a^b) \ (mod\ p) \equiv (a^{k(p-1)+l}) \ (mod\ p) \equiv (a^{k(p-1)} * a^l) \ (mod\ p) \equiv (1^k * a^l) \ (mod\ p) \equiv a^l \ (mod\ p)$$

Gdzie $k(p-1) + l = b$. Oznacza to, że możemy modulować wykładnik przez modulo $p - 1$. Otrzymujemy więc tożsamość:

$$a^{b \ (mod\ p-1)} \ (mod\ p) \equiv (a^b) \ (mod\ p)$$

3.7.4 Dzielenie

Cały problem w dzieleniu tkwi w perspektywie, z której patrzymy na problem. Co jeśli zamiast dzielenia będziemy mnożyć przez odwrotność? Okazuje się, że operacja ta będzie poprawna! Nasuwa się więc pytanie; skąd wziąć odwrotnośc liczby? Odpowiedzią jest MTF. Kongruencja $a^{p-1} \equiv 1 \ (mod\ p)$ jest niczym innym, jak przepisem na odwrotności w pierścieniu mod p. Niestety modulo często jest liczbą bardzo dużą, natomiast potęgowanie na piechotę zajmuje liniowo dużo czasu. Potrzebujemy więc lepszego algorytmu potęgującego.

3.7.5 Szybkie potęgowanie modulo

W celu szybkiego policzenia wartości liczby $a^b \ (mod\ p)$ skorzystamy z następujących sprytnych tożsamości matematycznych:

- Jeśli $b = 0$, to $a^b = 1 \ (mod\ p)$,
- jeśli b jest parzyste, to $a^b \equiv ((a * a) \ (mod\ p))^{\frac{b}{2}} \ (mod\ p)$,
- natomiast w ogólności $a^b \equiv (a * a^{b-1}) \ (mod\ p)$.

Nasze podejście będzie rekurencyjne; za każdym razem, gdy wykładnik będzie parzysty, użyjemy drugiej tożsamości, co pozwoli nam zredukować go o połowę. W przypadku, gdy wykładnik będzie nieparzysty, wtedy zastosujemy trzecią tożsamość, czyniąc go parzystym w następnym wywołaniu rekurencyjnym. Ostatecznie, gdy osiągnie on wartość 0, po prostu zwrócimy 1. Jaka jest złożoność powyższego podejścia? Zauważmy, że nigdy nie zastosujemy trzeciej tożsamości dwa razy pod rząd. Oznacza to, że będziemy redukować wykładnik o połowę w co drugim wywołaniu rekurencyjnym. Daje to więc złożoność logarytmiczną względem wielkości potęgi (**O(log(b))**). Poniższy algorytm możemy stosować z powodzeniem w operacji dzielenia, jak i samego odwracania liczby:

```
function potega(a: int, b: int, p: int)
begin
    if b == 0 then
        return 1;
    elif b % 2 == 0 then
        long long wynik = potega(a, b / 2); // jeśli nie przechowamy wyniku tej operacji w
                                            // zmiennej, to złożonosc pozostanie liniowa
        return (wynik * wynik) % p;
    else
        return (a * potega(a, b - 1) % p;
    endif
end;
```

3.7.6 Szczegóły implementacyjne

Warto zaznaczyć, że teoria nie jest idealnym odzwierciedleniem rzeczywistości; w praktyce problematyczne okazują się zakresy zmiennych oraz ujemne operandy operacji modulo. Komputer sobie z nimi nie radzi. Musimy więc sami zadbać o to, by operacje modulo mieściły się w zakresie oraz ich operandy były zawsze dodatnie (nieujemne). Poniżej przedstawiono metody pozwalające na bezpieczne korzystanie z podstawowych działań w pierścieniu modulo p:

```
function dodawanie(a: int, b: int, p: int)
begin
    return ((a % p) + (b % p)) % p;
end;
function odejmowanie(a: int, b: int, p: int)
begin
    return ((a % p) - (b % p) + p) % p; // + p powoduje, że wynik nie będzie ujemny
end;
function mnożenie(a: int, b: int, p: int)
begin
    return ((a % p) * (b % p)) % p;
end;
function dzielenie(a: int, b: int, p: int)
begin
    return ((a % p) * odwrotność(b, p)) % p; // zamiast dzielenia mnożymy przez odwrotność
end;
```

```
function odwrotność(a: int, p: int)
begin
    return potega(a, p - 2) % p; // stosujemy MTF
end;
```

3.8 Problemy NP-trudne

Problemy informatyczne można podzielić na pewien zbiór klas w zależności od złożoności jakimi charakteryzują się zarówno ich rozwiązanie, jak i sprawdzenie kandydata na potencjalne rozwiązanie. Głównym rodzajem badanych problemów będą problemy decyzyjne, czyli takie, na które odpowiedź może brzmieć jedynie **tak** albo **nie**. W szczególności problemy decyzyjne można poznać po ich strukturze, gdyż odpowiadają one na pytanie, które zaczyna się od frazy **Czy...**.

3.8.1 Klasa P

Znacząca większość problemów, które spotykamy na codzień jest klasy P (polynomial). Problemy tej klasy charakteryzuje fakt, że można je rozwiązać w złożoności wielomianowej (a co za tym idzie także zweryfikować poprawność rozwiązania). Zdecydowana większość problemów na Olimpiadzie Informatycznej należy do właśnie tej grupy.

3.8.2 Klasa NP

Drugą istotną podgrupą problemów są problemy klasy NP (non-polynomial). Są to problemy, dla których nie zostało dotąd przedstawione rozwiązanie o złożoności wielomianowej. Rozwiązania problemów NP mają więc złożoność wykładniczą. W definicji problemu NP wysoce istotne jest istnienie sposobu na weryfikację rozwiązania w złożoności wielomianowej tj. jeśli mamy kandydata na rozwiązanie, to pomimo że samo rozwiązanie wymagało użycia algorytmu o złożoności wykładniczej, to sama weryfikacja czy rozwiązanie jest poprawne charakteryzuje się złożonością wielomianową.

3.8.3 Klasa NPC

Podgrupą problemów NP są problemy NPC. W dużym uproszczeniu są to najtrudniejsze z problemów NP. Najważniejszą cechą problemów NPC jest fakt, że rozwiązanie jednego z nich w złożoności wielomianowej oznaczało by, że dowolny problem klasy NP da się rozwiązać w złożności wielomianowej, a więc P = NP. Jest to jeden z problemów milenijnych.

3.8.4 Klasa NPH

Klasą NPH nazwiemy wszystkie problemy, które są co najmniej tak trudne jak problemy NPC. W szczególności mowa tutaj o problemach niedecyzyjnych, a optymalizacyjnych, powstałych w wyniku generalizacji problemów NPC. Warto zaznaczyć, że problemy te nie muszą już być weryfikowalne w czasie wielomianowym (ale jak najbardziej mogą).

3.8.5 Klasa EXP

Pozostałe problemy mieszczą się w klasie EXP. Klasa ta charakteryzuje wszystkie problemy, których zarówno rozwiązanie, jak i weryfikacja przebiega w złożoności co najmniej wykładniczej.

3.8.6 Podejście do problemów NP

Z racji, że dane wielkości kilkudziesięciu zmiennych binarnych stają się bardzo czasochłonne w przetwarzaniu, postanowiono obrać inny sposób radzenia sobie z problemami NP. Głównym sposobem na radzenie sobie ze złożonością problemów niewielomianowych jest używanie rozmaitych heurystyk. Pozwalają one na osiągnięcie wyniku często zbliżonego, bądź równego prawdziwemu, jednocześnie zachowując złożoność o postaci wielomianowej. Co więcej spora część z nich posiada formalnie udowodniony zakres błędu, jakim mogą zostać obarczone rezultaty.

Niekiedy zdarza się też, że specyfika danych wejściowych problemu, bądź towarzyszące mu reguły logiczne pozwalają na zredukowanie problemu, do z pozoru bardzo podobnej wersji, jednakże posiadającej dużo bardziej przystępne rozwiązanie. Na samej Olimpiadzie zadania bazujące na problemach klasy NP pojawiają się bardzo sporadycznie. Dużo częstszym widokiem są właśnie zadania bazujęce na problemach niewielomianowych, jednakże z podowu charakterystyki zadania rozwiązywalne w złożoności wielomianowej.

3.9 Haszowanie

W informatyce często zdarza się sytuacja, w której musimy porównać dwa obiekty ze sobą. Jeśli obiekty są dużych rozmiarów, a nam zależy na szybkości rozwiązania, to dobrym pomysłem byłoby obliczenie pewnego identyfikatora (w ogólności krótszego, niż zawartość obiektu), a następnie użycie go do porównania obiektów. Jeśli identyfikator ustalamy na podstawie wartości obiektu, to mamy gwarancję, że dwa takie same obiekty będą posiadały identyczny identyfikator, dzięki czemu zaoszczędzamy potencjalnie dużo czasu na samej operacji porównania, zwłaszcza gdy jest ich wiele. Proces obliczania identyfikatora na podstawie obiektu nazywamy **haszowaniem**.

3.9.1 Wybór funkcji haszującej

Haszowanie posiada jedną istotną wadę; funkcja haszowania nie jest injekcją. Oznacza to, że dla dwóch różnych obiektów wynik przepuszczenia ich przez funkcję haszującą może być równy. Operacja porównania dwóch haszów może więc dać niepoprawny rezultat. Kluczowy jest więc dobór takiej fukcji haszującej, dla której błąd zdarza się możliwie sporadycznie. Na potrzeby zadań olimpijskich idealnie sprawdza się podejście zwane **haszowaniem wielomianowym**.

3.9.2 Haszowanie wielomianowe

Dla skupienia uwagi załóżmy, że haszowanymi obiektami będą słowa składające się wyłącznie z małych liter alfabetu angielskiego. Haszowanie wielomianowe zakłada wygenerowanie haszu za pomocą pewnego wielomianu, którego współczynniki będą zależeć od kolejnych liter słowa. Wszystkie operacje będą miały miejsce w pewnym pierścieniu modulo.

Na początku należy ustalić kilka wartości zwanych jako parametry haszowania. Pierwszym z nich jest liczba p stanowiąca podstawę wielomianu haszującego. W celu uzyskania możliwie dużej dziedziny wyników, a co za tym idzie możliwie małego prawdopodobieństwa błędu należy wybrać liczbę pierwszą nieco większą od rozmiaru zbioru elementów, które będziemy haszować. Jako że nasz alfabet zawiera 26 różnych znaków, to odpowiednią wartością p będzie na przykład 31. Jako modulo warto przyjąć możliwie dużą liczbę pierwszą. Doskonale sprawdza się liczba 10^9+7. Aby postawa funkcji haszującej była istotnie większa od współczynników należy także pamietać o odpowiednim przeskalowaniu wartości poszczególnych elementów wchodzących w skład alfabetu ($a = 0$, $b = 1$, ..., $z = 25$).

Zdefiniujmy więc naszą funkcję haszującą. Oznaczmy poszczególne litery słowa S jako $s_1, s_2, ..., s_n$. Funkcja haszująca H prezentuje się następująco:

$$H(S) = (\sum_{i=1}^{n} S_i * p^{i-1})(mod\ M)$$

Dla przykładu wyznaczmy wartość haszu słowa *laptopy*. Poszczególne litery mają wartości kolejno 11, 0, 15, 19, 14, 15, 24. Poszczególne potęgi liczby p przyjmują wartości 1, 31, 961, 29791, 923521, 28629151, 887503681. Podstawiając liczby do funkcji otrzymujemy:

$$H(S) = (11 * 1 + 0 * 31 + 15 * 961 + 19 * 29791 + 14 * 923521 + 15 * 28629151 + 24 * 887503681)(mod\ M) =$$

$$= (11 + 31 + 14415 + 566029 + 12929294 + 429437265 + 300088155)(mod\ M) = 743035200$$

3.9.3 Szczegóły implementacyjne

Podczas implementacji haszowania szczególnie istotna jest ostrożność w wykonywaniu działań modulo. Należy pamiętać o zakresach zmiennych, które w razie potrzeby można powiększać. Ponadto potęgi liczby p

można ztablicować, dzięki czemu unikniemy ponownego ich wyliczania. Przykładowy algorytm haszowania prezentuje się następująco:

```
function preHash(p: int, n: int) // funkcja, którą wywołujemy przed pierwszym haszowaniem
begin
    pwr = new int[n + 1];
    pwr[0] := 1;
    for i := 1 to n do
        pwr[i] := (pwr[i - 1] * p) % M; // wyliczamy i tablicujemy polejne potęgi liczby p
    done
    return pwr;
end;

function hash(S: string, pwr: int[], p: int, M: int)
begin
    result := 0;
    for i := 1 to n do
        result = (result + (S[i] * pwr[i - 1]) % M) % M; // wyznaczamy hasz
    done
    return result;
end;
```

3.10 DSU - unia zbiorów rozłącznych

Wyobraźmy sobie, że działamy na pewnych zbiorach elementów. Każdy element należy do dokładnie jednego zbioru (w szczególności jednoelementowego). W dowolnym momencie chcemy móc przeprowadzić jedną z dwóch operacji:

- połączyć dwa wybrane zbiory w jeden (dokonać unii zbiorów),
- sprawdzić, czy dwa wybrane elementy należą do tego samego zbioru.

Ponadto bardzo zależy nam na przetwarzaniu danych online (w taki sposób, że dane przychodzą stopniowo, więc nie możemy podejrzeć kolejnych wartości) oraz liniowej złożoności rozwiązania. Z pomocą przychodzi sruktura **DSU** zwana także jako Disjoint Set Union lub Unia Zbiorów Rozłącznych.

3.10.1 Założenia struktury

Struktura składa się z zaledwie jednej tablicy, która dla każdego jej elementu przyporządkowuje tzw. *reprezentanta*. Jest to element, który jednoznacznie reprezentuje zbiór, w którym się znajduje. Dzięki takiemu podejściu wszystkie elementy, które posiadają tego samego reprezentanta wiedzą, że znajdują się w tym samym zbiorze. Analogicznie w drugą stronę - elementy o różnych reprezentantach nie mogą znajdować się w tym samym zbiorze. Na początku, przed rozpoczęciem łączenia elementów w zbiory każdy element jest jednocześnie reprezentantem swojego własnego, jednoelementowego zbioru.

Drugim istotnym faktem, który stanowi niejako ograniczenie struktury jest brak możliwości usuwania znajdujących się w niej elementów. Jeśli koniecznie zależy nam na takiej operacji, zainteresowanych odsyłam do zapoznania się ze strukturą *linktree*.

3.10.2 Operacja find

Załóżmy, że struktura nie musi przechowywać bezpośrednich reprezentantów zbioru, a jednynie pośrednie elementy, których reprezentanci rekurencyjnie prowadzą do właściwego reprezentanta całego zbioru. Będzie to przydatne podczas operacji unii. *Find*() jest operacją, której zadaniem jest zwrócić głównego reprezentanta zbioru, do którego należy pewien element. Działa w oparciu o prostą obserwację; główny reprezentant zbioru jest reprezentantem sam dla siebie. Możemy więc wyznaczyć go prostą rekurencją:

```
function find(a: int)
begin
    if rep[a] != a then
        return find(a);
    endif
    return rep[a];
end;
```

Dodatkowo zauważmy, że po odnalezieniu reprezentanta, możemy wracając się rekurencyjnie przypisać go bezpośrednio do wszystkich elementów, jakie napotkaliśmy na swojej drodze. Dzięki temu zredukujemy liczbę wywołań rekurencyjnych dla przyszłych operacji *find* i istotnie zamortyzujemy złożonośc naszego rozwiązania, co poskutkuje działaniem operacji w zamortyzowanym czasie liniowym. Oto poprawiony kod:

```
function find(a: int)
begin
    if rep[a] != a then
        rep[a] = find(a);
    endif
    return rep[a];
end;
```

Zmienione zostało jedno słowo, natomiast efekt działania jest zupełnie inny.

3.10.3 Operacja union

Dzięki wprowadzeniu pojęcia reprezentantów zbiorów operacja unii staje się trywialna. Wystarczy, że reprezentantowi pierwszego zbioru ustawimy reprezentanta zbioru drugiego jako nowego reprezentanta. Takie przypisanie spowoduje, że wszystkie elementy zbioru pierwszego po wywołaniu funkcji *find* jako wynik otrzymają reprezentanta zbioru drugiego, który tym samym stanie się głównym reprezentanem nowopowstałego zbioru.

Aby zredukować średni czas działania procedury oraz uniknąć nieprzyjemnych przypadków pesymistycznych, gdzie reprezentanci utworzą swojego rodzaju ścieżkę, dobrą praktyką jest przeprowadzanie unii mniejszego zbioru do większego. W tym celu przechowujmy aktualny rozmiar zbiorów reprezentowanych przez poszczególne elementy oraz aktualizujmy go w miarę przeprowadzania kolejnych unii. Kod rozwiązania prezentuje się następująco:

```
procedure onion(a: int, b: int) // niestety union jest zarezerwowanym słowem kluczowym
begin
    a = find(a);
    b = find(b);
    if size[a] > size[b] then
        swap(a, b);
    endif
    rep[a] = b;
    size[b] += size[a];
end;
```

Sumaryczna złożoność wszystkich zapytań do struktury wyniesie $O(n\alpha)$, gdzie α jest odwrotnością funkcji Aackermana - ekstremalnie szybko rosnącej funkcji rekurencyjnej. W praktyce wartość α nigdy nie przekroczy 5, więc można uznać, że struktura ma złożoność liniową.

4 Grafy

Teoria grafów stanowi bardzo ważną gałąź matematyki. Jej adaptacje w dziedzinach informatycznych się niezliczone, a rozwiązania przez nią oferowane niezastąpione, stąd będą one tematem odrębnego działu.

4.1 Wstęp do teorii grafów

Na wstępnie wypadałoby zdefiniować zarówno terminologię używaną w teorii grafów, jak i samo pojęcie grafu w sposób formalny.

Graf (nieskierowany) G jest uporządkowaną parą dwóch zbiorów: V oraz E. Zapisujemy więc $G = (V, E)$. V jest zbiorem elementów zwanych jako wierzchołki grafu, natomiast zbiór E zawiera nieuporządkowane pary wierzchołków ze zbioru V zwane krawędziami. Mamy więc: $v_i, v_j \in V : \{v_i, v_j\} = e \in E$. Na bazie tej definicji możemy stopniowo wyprowadzać kolejne pojęcia:

- Krawędź grafu e jest *incydentna* z wierzchołkiem v, jeśli $e = \{v, x\}$.
- Liczba krawędzi incydentnych z wierzchołkiem v zwana jest *stopniemwierzchokav* i zapisywana jako $deg(v)$.
- Wierzchołki v_1 oraz v_2 są sąsiadujące, jeśli istnieje krawędź $e = \{v_1, v_2\} \in E$.
- Pętlą nazwiemy krawędź e taką, że $e = \{v, v\}$.
- Drogą w grafie nazywamy sekwencję krawędzi pomiędzy wierzchołkami $v_1, v_2, ..., v_k$, taką że $\{v_1, v_2\} \in E, \{v_2, v_3\} \in E, ..., \{v_{k-1}, v_k\} \in E$.
- Ścieżką w grafie nazwiemy drogę, dla której wszystkie wierzchołki $v_1, v_2, ..., v_k$ są unikalne.
- Cyklem nazywa się zamkniętą ścieżkę tj. ścieżkę, która dodatkowo posiada krawędź między wierzchołkami v_1 oraz v_k.
- Graf pusty jest grafem, który nie posiada krawędzi.
- Graf pełny, to graf, który posiada krawędzie pomiędzy każdą parą wierzchołków tj. dla dowolnego v_i oraz v_j istnieje $e = \{v_i, v_j\} \in E$.
- Dopełnieniem grafu G nazywamy graf G' taki, że $e \in E(G) \iff e \notin E(G')$ oraz $e \in E(G') \iff e \notin E(G)$.
- Grafy G oraz G' są izomorficzne wtedy, gdy można spermutować wierzchołki grafu G' tak, by powstały graf był grafem G.
- Grafem dwudzielnym nazwiemy graf, w którym da się wyróżnić dwie grupy wierzchołków V_1 i V_2, że każdy wierzchołek grafu należy do jednej z nich oraz nie istnieje krawędź między dowolnymi wierzchołkami należącymi do tego samego zbioru tj. $\neg\exists\, e \in E : e = \{v_i, v_j\}, v_i \in V_k, v_j \in V_k, k \in \{1, 2\}$.
- Grafem dwudzielnym pełnym nazwiemy graf dwudzielny, w którym dla każdej pary wierzchołków $v_i \in V_1$ oraz $v_j \in V_2$ istnieje krawędź $e = \{v_i, v_j\} \in E$.
- Gwiazdą wierzchołka v nazwiemy zbiór wierzchołków sąsiadujących z v. Oznaczamy go $st(v)$.
- Ścieżką eulerowską nazywa się ścieżkę, która przechodzi przez każdą krawędź dokładnie raz.
- Cyklem eulerowskim nazywa się cykl, który przechodzi przez każdą krawędź dokładnie raz.
- Ścieżką hamiltonowską nazywa się ścieżkę, która przechodzi przez każdy wierzchołek dokładnie raz.
- Cyklem hamiltonowskim nazywa się cykl, który przechodzi przez każdy wierzchołek dokładnie raz.
- Graf spójny to graf, w którym istnieje ścieżka między dowolną parą wierzchołków.
- Wierzchołek izolowany to wierzchołek, który dla którego nie istnieje krawędź do niego incydentna.
- Drzewo to spójny graf acykliczny tj. graf, który nie zawiera cyklu.
- Las jest grafem będącym zbiorem drzew.
- Graf planarny, to graf, który można narysować na płaszczyźnie bez przecięć miedzy krawędziami.

- Multigrafem nazywamy graf, który może posiadać więcej niż jedną krawędź rozpiętą między dwoma tymi samymi wierzchołkami.
- Graf ważony go traf, dla którego każdej krawędzi odpowiada pewna liczba zwana wagą krawędzi.
- Grafem skierowanym (digrafem) oznaczamy graf, którego krawędzie mogą biegnąć tylko w jednym kierunku.
- Turniejem nazywamy skierowany graf pełny.
- DAG'iem nazywamy skierowany graf acykliczny.
- Meduza to graf zawierający tyle samo wierzchołków, co krawędzi.

4.2 Sposoby reprezentacji grafów

W praktyce grafy nie są implementowane w ich stricte matematycznej formie. Stosuje się alternatywne struktury, które często wiążą się z wyższym nakładem pamięciowym oraz bardziej skomplikowanym sposobem realizacji, jednakże oferują o wieje bardziej przystępny sposób uporządkowania danych.

4.2.1 Dane wejściowe

W treściach zadań olimpijskich graf najczęściej definiowany jest jako liczby n oraz m, reprezentujace kolejno liczbę wierzchołków oraz krawędzi grafu wejściowego. Jeśli graf jest drzewem często rezygnuje się z podawania liczby krawędzi, gdyż ta wynika wprost z liczby jego wierzchołków (dla drzewa liczba krawędzi jest zawsze o jeden mniejsza od liczby wierzchołków). Następnie podawane jest m par liczb będących indeksami wierzchołków, pomiędzy którymi znajdują się krawędzie grafu. W przypadku grafów ważonych dodatkoło dołączana jest trzecia wartość oznaczająca wagę krawędzi.

4.2.2 Macierz incydencji

Reprezentacja macierzowa polega na reprezentowaniu grafu w postaci macierzy wartości boolowskich. Każda kolumna, jak i rząd odpowiada danemu wierzchołkowi. Komórka na przecięciu i-tego rzędu z j-tą kolumną zawiera wartość *true* jeśli między danymi wierzchołkami istnieje krawędź. W przeciwnym wypadku przyjmuje wartość $false$. W przypadku grafu ważonego zamiast wartości boolowskich stosuje się wartości całkowitoliczbowe oznaczające wagi krawędzi, natomiast w przypadku braku połączenia używa się umownie dużej wartości reprezentującej nieskończoność. Poniżej przedstawiono przykładową implementację wczytania grafu w postaci macierzy incydencji:

```
int n, m;
int v1, v2;
int[][] gr;

read n, m;
for i := 0 to n do
    read v1, v2;
    v1--;
    v2--;
    gr[v1][v2] := true;
    gr[v2][v1] := true;
done
```

Główną zaletą macierzy incydencji jest możliwość sprawdzania czy graf zawiera dane połączenie (krawędź) w czasie stałym. Niestety jest to okupione faktem, że chcąc otrzymać zbiór wszystkich sąsiadów danego wierzchołka zawsze musimy przeglądnąć cały rząd (lub kolumnę), co powoduje, że niektóre operacje działają w złożoności kwadratowej.

4.2.3 Lista sąsiedztwa

Alternatywną reprezentacją grafu jest reprezentacja w postaci listy sąsiedztwa. Polega ona na utworzeniu dla każdego wierzchołka grafu listy wierzchołków, które z nim sąsiadują. W ten sposób narzut pamięciowy dostosowany jest do faktycznego rozmiaru zbioru krawędzi grafu. Listy wierzchołków dostosowywane są dynamicznie, także dobrze sprawdzają się tutaj wektory. Oto przykładowa implementacja:

```
int n, m;
int v1, v2;
vector<int>[] gr;

read n, m;
for i := 0 to n do
    read v1, v2;
    v1--;
    v2--;
    gr[v1].push_back(v2);
    gr[v2].push_back(v1);
done
```

Jak widać realizacja implementacji listy sąsiedztwa niewiele różni się od macierzy incydencji. Główną zaletą listy jest możliwość przeszukiwanai grafu w liniowym czasie. Niestety w tym przypadku cierpi złożoność sprawdzania połączenia wierzchołków, gdyż wynosi ona $O(\Delta)$, gdzie delta jest największym stopniem wierzchołka w grafie.

4.2.4 Pozostałe reprezentacje

Oczywiście istnieje o wiele więcej reprezentacji grafów, chociażby takich jak pęki wyjściowe. Nie są one jednak tak praktyczne, jak powyższe dwie i wykorzystywane bardzo sporadycznie (oczywiście mówimy tutaj o kontekście zadań obecnych na Olimpiadzie Informatycznej oraz pozostałych konkursach), stąd nie będą wymieniane w tym rozdziale. Zainteresowanych odsyłąm do lektury na temat min. pęków wyjściowych.