Uniwersytet Mikołaja Kopernika w Toruniu
Wydział Matematyki i Informatyki

Marcin Wojdat
nr albumu: 291849
informatyka, studia inżynierskie

Praca inżynierska

Biblioteka algorytmów i struktur danych używanych w zawodach programistycznych

Opiekun pracy dyplomowej
dr Marcin Piątkowski

Toruń 2021

Tytuł pracy

Biblioteka algorytmów i struktur danych używanych w zawodach programistycznych

Thesis title

Library of algorithms and data structures used in programming competitions

Streszczenie

Praca zawiera implementacje algorytmów i struktur danych najczęściej używanych w zawodach programistyczno-algorytmicznych takich jak Olimpiada Informatyczna, Akademickie Mistrzostwa Polski w Programowaniu Zespołowym, Central Europe Regional Contest, Top Coder. Duży nacisk położony jest na swobodę w korzystaniu z tych gotowych rozwiązań w swoich programach oraz czytelną dokumentację.

Abstract

The thesis contains implementations of algorithms and data structures most often used in programming and algorithmic competitions such as the Polish Olympiad in Informatics, Polish Collegiate Programming Contest, Central Europe Regional Contest, Top Coder. Great emphasis is placed on the freedom to use these ready-made solutions in their programs and clear documentation.

Słowa kluczowe

algorytmy i struktury danych, teoria grafów, geometria obliczeniowa, teoria liczb

Keywords

algorithms and data structures, graph theory, computational geometry, number theory

Spis treści

Rozdział 1

Wstęp

1.1 Przedmowa

Comptetive programming jest to rodzaj sportu umysłowego, w którym uczestnicy rozwiązują przygotowany zestaw problemów logicznych, matematycznych czy algorytmicznych w warunkach konkursowych, tzn. kto rozwiąże największą ilość zadań w określonym czasie - wygrywa. Większość tego typu konkursów odbywa się przez Internet, jednak istnieje również wiele cyklicznych zawodów stacjonarnych. Najpopularniejsze wydarzenia o randze międzynarodowej to: ACM ICPC, Top Coder, Google Code Jam, Międzynarodowa Olimpiada Informatyczna. W Polsce największym rozgłosem cieszą się konkursy takie jak Akademickie Mistrzostwa Polski w Programowaniu Zespołowym, Olimpiada Informatyczna, Potyczki Algorytmiczne.

Zadania, z którymi zmagają się uczestnicy tego sportu umysłowego mają zazwyczaj pewną ustaloną formę. W treści zadania znajduje się szczegółowy opis danych wejściowych, danych wyjściowych oraz na czym ma polegać przetworzenie tych danych. Nie jest jednak określony sposób, w jaki należy dokonać tej przemiany - to jest właśnie zadanie uczestników. Rozwiązaniem tych zadań jest program komputerowy, który dla dowolnych danych wejściowych zgodnych ze specyfikacją podaną w opisie, wyznaczy (w określonym czasie) poprawną odpowiedź zgodną z wytycznymi problemu. Przykładowo zawodnicy, mogą otrzymać zestaw liczb (np. [5, 7, 9, -3, 2, 15, -6, 0]), a ich zadaniem będzie wyliczenie ich sumy (w tym przypadku 29). Uczestnicy wiedzą zatem *co* należy zrobić z otrzymanymi danymi, muszą tylko (a może i aż) znaleźć sposób *jak* to osiągnąć.

Dla zawodników jednym z najważniejszych zasobów podczas trwania konkursu jest czas. Zawody często trwają tylko kilka godzin, a do rozwiązania jest kilka/kilkanaście zadań. W celu przyśpieszenia pracy nad rozwiązywaniem zadań uczestnicy zabierają na zawody tzw. *biblioteczki*, czyli zbiory gotowych popularnych algorytmów i struktur danych, które stanowią podstawę rozwiązań trudniejszych problemów. Poniższa praca jest właśnie przykładem takiej biblioteki, której podstawowym celem jest przyśpieszenie procesu tworzenia programów

rozwiązujących zadania algorytmiczne. Większość ogólnodostępnych biblioteczek skupia się na wytłumaczeniu działania prezentowanych algorytmów, często także na formalnych dowodach poprawności, co jest przydatne, ale w trakcie treningu. Podczas zawodów, często brakuje czasu na naukę nowych algorytmów, dlatego aspekt edukacyjny biblioteczki schodzi na dalszy plan. W mojej pracy duży nacisk położony jest na praktyczność - prezentowane rozwiązania zawierają dokumentację najważniejszych elementów z punktu widzenia zawodników, czyli opis argumentów funkcji, zwracanych wartości oraz złożoności obliczeniowej i pamięciowej. To powoduje, że można ich z powodzeniem używać nawet bez znajomości danego zagadnienia.

1.2 Założenia

Oto podstawowe założenia poniższej pracy:

1. Przyśpieszenie implementacji rozwiazań zadań konkursowych poprzez zebranie gotowych algorytmów i struktur danych, które często przydają się podczas zawodów algorytmiczno-programistycznych.
2. Szczegółowa dokumentacja prezentowanych kodów, w tym opis argumentów funkcji i metod, zwracanych wartości, korelacji z innymi algorytmami, złożoność obliczeniowa i pamięciowa.
3. Schematyczność rozdziałów, dzięki czemu czytelnik nie musi zapoznać się z całą treścią, by efektywnie znajdować potrzebne informacje.
4. Brak konieczności znajomości działania przedstawionych algorytmów, by z powodzeniem móc je uzywać w swoich programach (zasada *black box*).
5. Do każdego podrozdziału dołączony jest krótki opis. Nie ma on na celu wyjaśnienia zasady działania algorytmu, lecz przybliżenie czytelnikowi problemu. W celach edukacyjnych odsyłam do zamieszczonej w blibliografii literatury lub innych źródeł informacji.

1.3 Wymagania wstępne

Podstawową umiejętnością wymaganą do efektywnego korzystania z poniższej pracy jest znajomość programowania w języku C++, gdyż w tym języku programowania zostałe one napisane. Znajomość algorytmiki nie jest wymagana, ponieważ celem podczas przygotowywania materiałów do tej pracy była możliwość łatwej migracji prezentowanych rozwiązań do swoich programów, nawet bez znajomości omawianych zagadnień.

Podczas rozwiązywania problemów z tej dziedziny zdarza się, że do rozwiązania zadania, trzeba lekko zmodyfikować pewien podstawowy algorytm, by dopasować go do specyfikacji rozwiązywanego problemu (np. aby znaleźć maksymalne drzewo rozpinające w grafie, można użyć któregoś z popularnych

algorytmów na znajdowanie minimalnego drzewa rozpinającego i zmienić kolejność przetwarzania krawędzi). Zatem znajomość prezentowanych zagadnień z zakresu algorytmów i struktur danych umożliwia ewentualną modyfikację kodu źródłowego, by pasował on do specyfikacji zadania.

Warto jednak rozumieć podstawowe pojęcia w algorytmice takie jak złożoność czasowa, złożoność pamięciowa, rozwiązanie brutalne (ang. Brute-Force), rozwiązanie zachłanne, podejście dziel i zwyciężaj, rekurencja itp., by świadomie korzystać z zamieszczonych w pracy algorytmów i orientować się, który algorytm będzie wystarczająco efektywny w rozwiązywanym zadaniu.

1.4 Nagłówki

Ważnym czynnikiem podczas rozwiązywania zadań na zawodach jest czas poświęcony na implementację kodu. Czas ten można efektywnie skrócić, poprzez korzystanie z przygotowanych implementacji pewnych podstawowych algorytmów, jak również dzięki korzystaniu z tak zwanych nagłówków (makr). Pozwalają one na krótszy zapis długich, często używanych konstrukcji językowych podczas kodowania, dzięki czemu uczestnicy mają do napisania mniej znaków, a co za tym idzie poświęcają mniej czasu na kodowanie.

Na większości konkursów (szczególnie stacjonarnych) obowiązuje zakaz używania wszelakich materiałów elektronicznych. Dopuszczalne jest natomiast korzystanie z papierowych pomocy naukowych, zatem aby korzystać z takich nagłówków na zawodach, trzeba na początku trwania konkursu poświęcić kilka minut na ich przepisanie, ale oszczędność czasu podczas dalszej pracy nad kodem jest zauważalna. W warunkach konkursowych każda minuta, czy nawet sekunda, może przesądzić o wyniku, dlatego każdy zabieg pozwalający zaoszczędzić trochę czasu jest mile widziany.

Na poniższym listingu przedstawione zostały wszystkie nagłówki, które są potrzebne do prawidłowego działania innych algorytmów w tej pracy. Zaleca się przepisanie wszystkich nagłówków na początku konkursu, by nie musieć poświęcać im uwagi w dalszej części zawodów.

Listing 1.1: "Wymagane nagłówki"

```
1  // Biblioteczki
2  #include <iostream>
3  #include <cstdio>
4  #include <cmath>
5  #include <algorithm>
6  #include <vector>
7  #include <queue>
8  #include <stack>
9  #include <numeric>
10 using namespace std;
11 // Typy proste
12 typedef unsigned int        UI;
```

```
13 typedef long long              LL;
14 typedef unsigned long long  ULL;
15 // Pary
16 typedef pair<int, int> PI;
17 #define MP               make_pair
18 #define ST                first
19 #define ND               second
20 // Vectory
21 typedef vector<int> VI;
22 typedef vector<bool> VB;
23 typedef vector<double> VD;
24 typedef vector<string> VS;
25 typedef vector<VI> VVI;
26 typedef vector<PI> VPI;
27 #define VT               vector<T>
28 #define VVT              vector<VT >
29 #define PB               push_back
30 #define SIZE(v)          (int(v.size()))
31 #define ALL(v)           v.begin(), v.end()
32 // Pętle
33 #define LOOP(i, a, b) for (int i = (a); i < (int)(b); ++i)
34 #define FOREACH(x, v) for (auto x: v)
35 #define REP(i, n)        LOOP(i, 0, n)
36 // Stałe
37 const int INF = 1e9+9;
38 const double EPS = 1e-9;
39 // Grafy
40 #define ET  Edge<T>
41 #define GT  Graph<T>
42 #define VE  vector<ET*>
43 #define VVE vector<VE >
44 #define PVTVE pair<VT, VE >
45 // Geometria
46 #define VP vector<Point>
47 #define ISZERO(x) abs(x) < EPS
48 // Inne
49 #define MOD(a, m) (a % m + m) % m
50 #define TEMPL template <typename T>
51 #define NP nullptr
52 #define BETWEEN(x, a, b) (((a)<= x && x<=(b)) || ((b)<=x && x<=(a)))
```

Poza nagłówkami podstawowymi zaleca się używanie makr dodatkowych. Nie są one konieczne do działania kodów źródłowych zamieszczonych w poniższej pracy, ale zawierają przydatne skróty oraz funkcje np. zerowanie tablic, prosty mechanizm debuggowania, czy wczytywanie/wypisywanie tablic. Należy traktować poniższe nagłówki jako sugestię, nie jest również wymagane przepisa-

nie wszystkich nagłówków, a tylko wybranych (większość z nich jest niezależna od innych).

Listing 1.2: "Dodatkowe nagłówki"

```
1  // Biblioteczki
2  #include <string>
3  #include <map>
4  #include <set>
5  #include <list>
6  #include <bitset>
7  #include <stack>
8  #include <queue>
9
10 // Vectory
11 typedef vector<UI>     VUI;
12 typedef vector<LL>     VLL;
13 typedef vector<ULL>    VULL;
14
15 // Pary
16 typedef pair<double, double> PD;
17 typedef pair<string, string> PS;
18 typedef pair<UI, UI>         PUI;
19 typedef pair<LL, LL>         PLL;
20 typedef pair<ULL, ULL>       PULL;
21
22 // Pętle
23 #define DLOOP(i, a, b) for (int i = (a); i <= (b); ++i)
24 #define RLOOP(i, a, b) for (int i = (a); i >= (b); --i)
25 #define TESTS(t)       int t; cin >> t; REP(i, t)
26
27 // Stałe
28 const LL LLINF = 1e18+9;
29
30 // Konwersja char na int i odwrotnie
31 #define CTOI(c) (int(c))
32 #define ITOC(x) (char(x))
33
34 // I/O
35 #define FASTIO() ios_base::sync_with_stdio(0); cin.tie(NULL);
36 TEMPL void printTab(VT &v) {
37     cout << "[";
38     REP(i, v.size())
39         cout << (i ? ", " : "") << v[i];
40     cout << "]\n";
41 }
42 TEMPL void readTab(VT &v) {
```

```
43      int n;
44      cin >> n;
45      v.resize(n);
46      REP(i, n) cin >> v[i];
47  }
48
49  // Debugowanie
50  #ifdef DEBUGGING
51  #define DEBUG(x) cout << #x << ": " << x << endl;
52  #else
53  #define DEBUG(x)
54  #endif
```

1.5 Uwagi

- W dokumentacjach funkcji/metod prezentowanych w poniższej pracy zamieszczony jest opis każdego argumentu wraz z zakresem wartości, dla których można założyć, że program zadziała poprawnie. Dla zmiennych, które mają wpływ na złożoność obliczeniową podany jest zalecany rząd wielkości.
- Poprawność działania programów testowana była na zadaniach wypisanych w rozdziale "Lista zadań". Ponadto w celu lepszego przetestowania wypisanych algorytmów, każdy z nich ma przygotowane wiele testów jednostkowych.
- Wiele prezentowanych funkcji w argumencie przyjmuje referencję do tablic lub do złożonych struktur (opartych na tablicach) w celu szybszego działania programu, jednak przesyłane obiekty nie są modyfikowane wewnątrz tych funkcji (wyjątki: `readGraph()` - listing 4.4, strona 33 oraz `makeGraph()` - listing 4.5, strona 34)

Rozdział 2

Teoria liczb

2.1 NWD

NWD - największy wspólny dzielnik (ang. greatest common divisor) dla dwóch lub więcej liczb, jest to największa liczba naturalna, która jest dzielnikiem każdej z nich.

Listing 2.1: "NWD"

```
1 ULL gcd(LL a, LL b) {
2     a = abs(a);
3     b = abs(b);
4     if (a < b) swap(a, b);
5     return b ? gcd(b, a % b) : a;
6 }
```

Argumenty

- **long long a**: pierwsza wartość, $a \in [-2^{63}, 2^{63})$ (zakres typu `long long`)
- **long long b**: druga wartość, $b \in [-2^{63}, 2^{63})$ (zakres typu `long long`)

Wartość zwracana

- **unsigned long long result**: największy wspólny dzielnik liczb a i b, result $\in [0, min(|a|, |b|)]$
- wartość 0, gdy $a = b = 0$ ($nwd(0, 0)$ nie istnieje)

Złożoność

- czasowa: O($log(|a| + |b|)$)
- pamięciowa: O($log(|a| + |b|)$)

Uwagi

- W blibliotece `algorithm` występuje funkcja `__gcd(a, b)` rozwiązująca problem największego wspólnego dzielnika dwóch liczb naturalnych.

Listing 2.2: "Przykład użycia"

```
1 int main() {
2     int a, b;
3     while (cin >> a >> b)
4         printf("nwd(%d, %d) = %llu\n", a, b, gcd(a, b));
5 }
```

Wejście	**Wyjście**
2 3	nwd(2, 3) = 1
123 456	nwd(123, 456) = 3
-5 -10	nwd(-5, -10) = 5
123456789 -123456789	nwd(123456789, -123456789) = 123456789
0 0	nwd(0, 0) = 0

2.2 NWW

NWW - najmniejsza wspólna wielokrotność (ang. lowest common multiple) dla dwóch lub więcej liczb, jest to najmniejsza liczba naturalna, która dzieli się przez każdą z tych liczb.

Listing 2.3: "NWW"

```
1 ULL lcm(int a, int b) {
2     a = abs(a);
3     b = abs(b);
4     return (a || b) ? (ULL) a / gcd(a, b) * b : 0;
5 }
```

Argumenty

- **int a**: pierwsza wartość, $a \in [-2^{31}, 2^{31})$ (zakres typu `int`)
- **int b**: druga wartość, $b \in [-2^{31}, 2^{31})$ (zakres typu `int`)

Wartość zwracana

- **unsigned long long result**: najmniejsza wspólna wielokrotność liczb a i b, result $\in [0, |a * b|]$

Złożoność

- czasowa: O($log(|a| + |b|)$)
- pamięciowa: O($log(|a| + |b|)$)

Wymagania

- funkcja `gcd(long long, long long)` - listing 2.1, strona 12

Listing 2.4: "Przykład użycia"

```
1 int main() {
2     int a, b;
3     while (cin >> a >> b)
4         printf("nww(%d, %d) = %llu\n", a, b, lcm(a, b));
5 }
```

Wejście	Wyjście
2 3	nww(2, 3) = 6
123 456	nww(123, 456) = 18696
-5 -10	nww(-5, -10) = 10
123456789 -123456789	nww(123456789, -123456789) = 123456789
0 0	nww(0, 0) = 0

2.3 Odwrotność modularna

Odwrotnością modularną liczby $x \in Z_m$ nazwiemy liczbę $y \in Z_m$, taką, że

$$x \cdot y \equiv 1(mod\ m)$$

lub inaczej

$$x \cdot y\ mod\ m = 1$$

Taką liczbę, można znaleźć w czasie logarytmicznym za pomocą rozszerzonego algorytmu Euklidesa.

Uwaga: W przypadku, gdy liczby x i m nie są względnie pierwsze, to nie istnieje odwrotność liczby x w pierścieniu Z_m.

Listing 2.5: "Odwrotność modularna"

```
1 UI invMod(LL a, UI m) {
2     LL b = m, x = 1 % m, y = 0;
3     a = MOD(a, m);
4     while (a > 1) {
5         if (!b) return 0;
```

```
6           x -= a / b * y;
7           a %= b;
8           swap(x, y);
9           swap(a, b);
10      }
11      return MOD(x, m);
12  }
```

Argumenty

- **long long a**: liczba, dla której chcemy wyznaczyć odwrotność, $a \in [-2^{63}, 2^{63})$ (zakres typu `long long`)
- **unsigned int m**: liczba modulo, $m \in [1, 2^{32})$

Wartość zwracana

- **unsigned int result**: wartość a^{-1} w Z_m, result $\in [1, m)$
- wartość 0, gdy nie istnieje odwrotność modularna liczby a w Z_m (gdy liczby a i m nie są względnie pierwsze) lub $m = 1$

Złożoność

- czasowa: O($log(m)$)
- pamięciowa: O(1)

Listing 2.6: "Przykład użycia"

```
1   int main() {
2       int a, m;
3       while (cin >> a >> m)
4           printf("%d ^ -1 mod %d = %u\n", a, m, invMod(a, m));
5   }
```

Wejście	Wyjście
3 7	3 ^ -1 mod 7 = 5
0 10	0 ^ -1 mod 10 = 1
1 999	1 ^ -1 mod 999 = 1
123456 1000000007	123456 ^ -1 mod 1000000007 = 78351802
-333 1000	-333 ^ -1 mod 1000 = 3

2.4 Potęgowanie modularne

Potęgowanie modularne polega na podniesieniu pewnej liczby a do określonej potęgi b w czasie logarytmicznym. Jako, że operacja potęgowania liczb całkowitych jest operacją bardzo szybko rosnącą do wielkich wartości, często wystarcza znajomość reszty z dzielenia wyniku potęgowania przez określoną liczbę m (najczęściej jest to liczba pierwsza).

Listing 2.7: "Potęgowanie modularne"

```
1 UI powMod(LL a, ULL b, UI m) {
2     if (!b) return 1 % m;
3     a = MOD(a, m);
4     UI res = powMod((ULL) a * a % m, b >> 1, m);
5     if (b & 1) res = res * a % m;
6     return res;
7 }
```

Argumenty

- **long long a**: podstawa potęgi, $a \in [-2^{63}, 2^{63})$ (zakres typu `long long`)
- **unsigned long long b**: wykładnik potęgi, $b \in [0, 2^{64})$ (zakres typu `unsigned long long`)
- **unsigned int m**: liczba modulo, $m \in [1, 2^{32})$

Wartość zwracana

- **unsigned int result**: wynik działania $a^b \bmod m$, res $\in [0, m)$

Złożoność

- czasowa: O($log(b)$)
- pamięciowa: O($log(b)$)

Uwagi

- jeżeli chcemy obliczyć poprawny wynik działania funkcji, gdy $b < 0$ należy użyć polecenia `powMod(invMod(a, m), -b, m)` (dodatkowe wymagania: funkcja `invMod(long long, unsigned int)` listing 2.5, strona 14)

Listing 2.8: "Przykład użycia"

```
1 int main() {
2     int a, b, m;
3     while (cin >> a >> b >> m)
4         printf("%d ^ %d mod %d = %u\n", a, b, m, powMod(a, b, m));
5 }
```

Wejście	Wyjście
0 999 100	0 ^ 999 mod 100 = 0
3 4 5	3 ^ 4 mod 5 = 1
15 20 1000	15 ^ 20 mod 1000 = 625
-10 3 15	-10 ^ 3 mod 15 = 5
123 456 789	123 ^ 456 mod 789 = 699

2.5 Symbol Newtona

Symbol Newtona - funkcja dwóch całkowitych nieujemnych argumentów określająca liczbę k-elementowych podzbiorów zbioru n-elementowego. Współczynnik dwumianowy Newtona możemy wyliczyć za pomocą wzoru:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}, \; dla \; 0 \leqslant k \leqslant n$$

Skracając czynnik $(n-k)!$ otrzymujemy wzór:

$$\prod_{i=1}^{k} \frac{n+1-i}{i}$$

Z racji, że w podstawowym wzorze na symbol Newtona występuje silnia, to wartości funkcji dla małych argumentów mogą być względnie duże (np. $\binom{30}{15} = 155117520$), dlatego często musimy się ograniczyć do obliczenia reszty z dzielenia wyniku przez pewną liczbę pierwszą p.

Listing 2.9: "Symbol Newtona"

```
1  UI newton(int n, int k, UI p = INF) {
2      if (n < 0 || k < 0 || n < k) return 0;
3      ULL res = 1;
4      LOOP(i, 0, min(k, n-k)) {
5          res = res * (n-i) % p;
6          res = res * invMod(i+1, p) % p;
7      }
8      return res;
9  }
```

Argumenty

- **int n**: pierwsza wartość, $n \in [-2^{31}, 2^{31})$ (zakres typu `int`)
- **int k**: druga wartość, $k \in [-2^{31}, 2^{31})$
- **unsigned int p**: liczba modulo (domyślnie $10^9 + 9$), p - liczba pierwsza, $p \in [2, 2^{32})$

Wartość zwracana

- **unsigned int result**: wynik działania $\binom{n}{k}$ $mod\ p$, result $\in [0, p)$
- wartość 0, dla $n < 0$, $k < 0$ lub $n < k$

Złożoność

- czasowa: O($\overline{k} \cdot log(p)$)*
- pamięciowa: O(1)

* $\overline{k} := min(k, n - k)$

Wymagania

- funkcja `invMod(long long, unsigned int)` - listing 2.5, strona 14

Uwagi

- Jeżeli wartości n i k nie są zbyt duże ($n, k < 1000$) oraz mamy do obsłużenia wiele zapytań o wartość symbolu Newtona warto sie zastanowić czy nie szybsze będzie obliczenie ich (i stablicowanie) dynamicznie korzystając ze wzoru

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}\ mod\ m,\ dla\ n \geqslant k > 0 \\ 1,\ dla\ n = 0\ \vee\ k = 0 \\ 0,\ wpw. \end{cases}$$

Listing 2.10: "Przykład użycia"

```
1 int main() {
2     int n, k, p;
3     while (cin >> n >> k >> p)
4         printf("(%d %d) mod %d = %u\n", n, k, p, newton(n, k, p));
5 }
```

Wejście	**Wyjście**
4 2 1000000009	(4 2) mod 1000000009 = 6
4 2 5	(4 2) mod 5 = 1
456 123 1000000009	(456 123) mod 1000000009 = 782313475
123 456 1000000009	(123 456) mod 1000000009 = 0
1000 1000 1000000009	(1000 1000) mod 1000000009 = 1

2.6 Sito Eratostenesa

Sito Eratostenesa służy do znalezienia wszystkich liczb pierwszych mniejszych od zadanej liczby n. Poniższa implementacja dla każdej liczby z przedziału $[0, n)$ zapisuje informację, czy liczba jest pierwsza czy nie, dzięki czemu po wyznaczeniu tej tablicy możemy w czasie stałym odpowiadać na zapytania dotyczące pierwszości liczb z tego przedziału. Ponadto za pomocą tej tablicy możemy w czasie liniowym wyznaczyć listę liczb pierwszych znajdujacych się w przedziale $[0, n)$.

Listing 2.11: "Sito Eratostenesa"

```
1  VB eratosthenes(int n) {
2      VB res = VB(n, 1);
3      res[0] = res[1] = 0;
4      int sq = sqrt(n);
5      LOOP(i, 2, sq+1)
6          if (res[i])
7              for (int j = i+i; j < n; j += i)
8                  res[j] = 0;
9      return res;
10 }
```

Argumenty

- **int n**: liczba do której chcemy uzyskać informacje o pierwszości liczb, $n \in [2, \sim 10^6]$

Wartość zwracana

- **vector<bool>result**: (tab[x] = 1 ⇒ x jest pierwsza; tab[x] = 0 ⇒ x nie jest pierwsza), |result| = n

Złożoność

- czasowa: $O(n \cdot log\ log(n))$
- pamięciowa: $O(n)$

Listing 2.12: "Przykład użycia"

```
1  int main() {
2      int n;
3      cin >> n;
4      printf("Lista liczb pierwszych mniejszych od %d:\n", n);
5      VB result = eratosthenes(n);
6      REP(i, n)
```

```
7           if (result[i])
8               cout << i << " ";
9   }
```

Wejście	Wyjście
100	Lista liczb pierwszych mniejszych od 100: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

2.7 Pierwiastki równania kwadratowego

Pierwiastki równania kwadratowego - są to rozwiązania równania postaci:

$$ax^2 + bx + c = 0,\ a \neq 0$$

Aby rozwiązać takie równanie należy policzyć wartość wyróżnika równania kwadratowego (tzw. delty)

$$\Delta := b^2 - 4ac$$

W zależności od wartości delty rozwiązanie to może mieć 0, 1 lub 2 pierwiastki.
$\Delta > 0 \Rightarrow$ 2 pierwiastki: $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
$\Delta = 0 \Rightarrow$ 1 pierwiastek: $x_0 = \frac{-b}{2a}$
$\Delta < 0 \Rightarrow$ brak pierwiastków (w dziedzinie liczb rzeczywistych)

Listing 2.13: "Pierwiastki równania kwadratowego"

```
1   VD rqe(double a, double b, double c) {
2       if (ISZERO(a)) return ISZERO(b) ? VD() : VD(1, -c/b);
3       else {
4           double dlt = b*b - 4*a*c;
5           if (ISZERO(dlt)) return VD(1, -b / (2*a));
6           else if (dlt < 0) return VD();
7           else {
8               VD res;
9               res.PB((-b-sqrt(dlt)) / (2*a));
10              res.PB((-b+sqrt(dlt)) / (2*a));
11              return res;
12          }
13      }
14  }
```

Argumenty

- **double a**: wartość parametru stojącego przy x^2, $a \in \mathbb{R}$ *

- **double b**: wartość parametru stojącego przy x, $b \in \mathbb{R}$ *
- **double c**: wartość parametru wolnego, $c \in \mathbb{R}$ *

* uwaga na dokładność obliczeń na liczbach zmiennoprzecinkowych dla parametrów o wysokich wartościach bezwzględnych

Wartość zwracana

- **vector<double>result**: tablica zawierająca wszystkie rzeczywiste pierwiastki równania kwadratowego, $|\text{result}| \in \{0, 1, 2\}$
- pusta tablica, gdy równanie ma nieskończenie wiele rozwiązań ($a = b = c = 0$)

Złożoność

- czasowa: $O(1)$
- pamięciowa: $O(1)$

Listing 2.14: "Przykład użycia"

```
1 int main() {
2     double a, b, c;
3     while (cin >> a >> b >> c) {
4         printf("Pierwiastki rownania: ");
5         VD result = rqe(a, b, c);
6         printTab(result);
7     }
8 }
```

Wejście	Wyjście
1 2 1	Pierwiastki rownania: [-1]
1 2.5 0	Pierwiastki rownania: [-2.5, 0]
-6 0 -8	Pierwiastki rownania: []
123 123 -123	Pierwiastki rownania: [-1.61803, 0.618034]
0 156.23 174.14	Pierwiastki rownania: [-1.11464]

Rozdział 3

Algorytmy tablicowe

3.1 Sumy prefiksowe

Sumy prefiksowe umożliwiają obliczenie sumy elementów pewnej tablicy na dowolnym spójnym przedziale w czasie stałym. Dla n-elementowej tablicy wejściowej (`tab`), musimy najpierw wyznaczyć tablicę:

$$sumTab[i] := \sum_{j \in [0,i)} tab[j],\ dla\ 0 \leqslant i \leqslant n.$$

Taką tablicę możemy obliczyć dynamicznie ze wzoru:

$$sumTab[i] = \begin{cases} 0,\ dla\ i = 0 \\ sumTab[i-1] + tab[i-1],\ dla\ 1 \leqslant i \leqslant n \end{cases}$$

Mając tak przygotowaną tablicę, aby obliczyć sumę na zadanym przedziale $[a, b)$ wystarczy odwołać się do sum prefiksowych na krańcach przedziału:

$$sum(a, b) := sumTab[b] - sumTab[a]\ (= \sum_{i \in [a,b)} tab[i])$$

Listing 3.1: "Sumy prefiksowe"

```
1  template <typename T, typename T2 = T> struct PrefixSums {
2      vector<T2> sumTab;
3      PrefixSums(VT &tab = VT()) {
4          sumTab.PB(0);
5          FOREACH(x, tab) sumTab.PB(sumTab.back() + x);
6      }
7
8      T2 sum(int a, int b) {
9          return a < b ? sumTab[b] - sumTab[a] : 0;
10     }
11 };
```

Szablon

- **typename T**: typ elementów wejściowej tablicy
- **typename T2**: typ elementów w tablicy sum prefiksowych (domyślnie typ T, zaleca się go zmienić gdy spodziewamy się, że suma na pewnych przedziałach może wykraczać poza rozmiar typu T)

Atrybuty

- **vector<T2>sumTab**: sumTab[i] := suma elementów wejściowej tablicy na przedziale $[0, i)$, $|\text{sumTab}| = n + 1$

PrefixSums(vector<T>&) - konstruktor

Argumenty

- **vector<T>&tab**: wejściowa tablica (domyślnie pusta), $n := |\text{tab}| \in [0, \sim 10^6]$

Złożoność

- czasowa: $O(n)$
- pamięciowa: $O(n)$

T2 sum(int, int)

Argumenty

- **int a**: początek przedziału, $a \in [0, n]$
- **int b**: koniec przedziału, $b \in [0, n]$

Wartość zwracana

- **T2 result**: suma elementów na przedziale $[a, b)$
- wartość 0, gdy podany przedział jest pusty ($a \geqslant b$)

Złożoność

- czasowa: $O(1)$
- pamięciowa: $O(1)$

Listing 3.2: "Przykład użycia"

```
1
2  int main() {
3      VI tab;
4      readTab(tab);
5      PrefixSums<int> prefixSums(tab);
6      int a, b;
7      while (cin >> a >> b)
8          printf("Suma na przedziale[%d, %d): %d\n",
9              a, b, prefixSums.sum(a, b));
10 }
```

Wejście	Wyjście
10 1 2 3 4 5 6 7 8 9 10 0 10 0 5 1 5 7 7 7 8	Suma na przedziale[0, 10): 55 Suma na przedziale[0, 5): 15 Suma na przedziale[1, 5): 14 Suma na przedziale[7, 7): 0 Suma na przedziale[7, 8): 8

3.2 Ekstremum na przedziale

Ekstemum na przedziale jest to najmniejszy (minimum) lub największy (maksimum) element znajdujący się na spójnym przedziale w pewnej tablicy. Przedstawiony algorytm umożliwia znalezienie ekstremum w czasie stałym dla dowolnego spójnego przedziału $[a, b)$ w n-elementowej tablicy wejściowej (`tab`). Należy najpierw obliczyć ekstremum dla wszystkich przedziałów o długościach będących potęgą liczby 2:

$$minTab[i][j] := \min_{k \in [j, j+2^i)} tab[k],\ dla\ *$$

$$maxTab[i][j] := \max_{k \in [j, j+2^i)} tab[k],\ dla\ *$$

* $0 < i \leqslant \lfloor log_2(n) \rfloor + 1,\ 0 \leqslant j < n - (2^i - 1)$
(minTab[i][j] - przechowuje minumum na przedziale zaczynającym się na indeksie j o długości 2^i, maxTab - analogicznie)

Takich przedziałów jest $O(n \cdot log(n))$. Po wyznaczeniu powyższych tablic, jesteśmy w stanie uzyskać odpowiedź dla każdego przedziału $[a, b)$ za pomocą co najwyżej 2-óch znanych przedziałów:

$$maximum(a, b) := max(maxTab[x][a], maxTab[x][b-2^x]),\ gdzie\ x = \lfloor log_2(b-a) \rfloor$$

$$minimum(a, b) := max(minTab[x][a], minTab[x][b-2^x]),\ gdzie\ x = \lfloor log_2(b-a) \rfloor$$

Listing 3.3: "Ekstremum na przedziale"

```
1  TEMPL struct MinMax {
2      VVT minTab, maxTab;
3      VI lg;
4      int n;
5      T inf;
6
7      MinMax(VT &tab = VT(), T inf = INF): n(tab.size()), inf(inf) {
8          if (n) {
9              minTab.resize(log2(n)+1);
10             maxTab.resize(log2(n)+1);
11             lg.resize(n+1);
12             minTab[0] = maxTab[0] = tab;
13             LOOP(i, 1, minTab.size()) {
14                 minTab[i].resize(n + 1 - (1<<i));
15                 maxTab[i].resize(n + 1 - (1<<i));
16                 REP(j, minTab[i].size()) {
17                     int x = j+(1<<(i-1));
18                     minTab[i][j] = min(minTab[i-1][j], minTab[i-1][x]);
19                     maxTab[i][j] = max(maxTab[i-1][j], maxTab[i-1][x]);
20                 }
21             }
22             LOOP(i, 1, n+1)lg[i] = log2(i);
23         }
24     }
25
26     T minimum(int a, int b) {
27         if (a >= b) return inf;
28         int x = lg[b-a];
29         return min(minTab[x][a], minTab[x][b-(1<<x)]);
30     }
31
32     T maximum(int a, int b) {
33         if (a >= b) return -inf;
34         int x = lg[b-a];
35         return max(maxTab[x][a], maxTab[x][b-(1<<x)]);
36     }
37 };
```

Szablon

- **typename T**: typ elementów wejściowej tablicy

Atrybuty

- **vector<vector<T>>minTab**: minTab[i][j] := minimum na przedziale $[j, j + 2^i)$, |minTab| $\approx n \cdot log(n)$
- **vector<vector<T>>maxTab**: maxTab[i][j] - maksimum na przedziale $[j, j + 2^i)$, |maxTab| $\approx n \cdot log(n)$
- **vector<int>lg**: lg[i] := $\lfloor log_2(i) \rfloor$ - tablica logarytmów, potrzebna by uniknąć obliczania logarytmów w metodach maximum i minimum, które mają działać w czasie stałym, |lg| $= n + 1$
- **int n**: ilość elementów w wejściowej tablicy
- **T inf**: określona wartość nieskończoności dla struktury (jeśli zostanie wywołane zapytanie o ekstremum na pustym przedziale to zostanie zwrócona wartość `inf` lub `-inf`)

MinMax(vector<T>&, T) - konstruktor

Argumenty

- **vector<T>&tab**: wejściowa tablica (domyślnie pusta), $n :=$ |tab| $\in [0, \sim 10^6]$
- **T inf**: wartość nieskończoności dla struktury (domyślnie $10^9 + 9$)

Złożoność

- czasowa: O($n \cdot log(n)$)
- pamięciowa: O($n \cdot log(n)$)

T minimum(int, int)

Argumenty

- **int a**: początek przedziału, $a \in [0, n]$
- **int b**: koniec przedziału, $b \in [0, n]$

Wartość zwracana

- **T result**: najmniejszy element na przedziale $[a, b)$
- ∞, gdy podany przedział jest pusty ($a \geqslant b$)

Złożoność

- czasowa: $O(1)$
- pamięciowa: $O(1)$

T maximum(int, int)

Argumenty

- **int a**: początek przedziału, $a \in [0, n]$
- **int b**: koniec przedziału, $b \in [0, n]$

Wartość zwracana

- **T result**: największy element na przedziale $[a, b)$
- $-\infty$, gdy podany przedział jest pusty ($a \geqslant b$)

Złożoność

- czasowa: $O(1)$
- pamięciowa: $O(1)$

Listing 3.4: "Przykład użycia"

```
1  int main() {
2      VI tab;
3      readTab(tab);
4      MinMax<int> minMax(tab);
5      int a, b;
6      while (cin >> a >> b) {
7          printf("Minimum na przedziale[%d, %d): %d\n",
8              a, b, minMax.minimum(a, b));
9          printf("Maksimum na przedziale[%d, %d): %d\n",
10             a, b, minMax.maximum(a, b));
11     }
12 }
```

Wejście	**Wyjście**
10 5 8 1 -3 8 4 2 7 -10 0 0 10 0 5 2 5 7 7 7 8	Minimum na przedziale[0, 10): -10 Maksimum na przedziale[0, 10): 8 Minimum na przedziale[0, 5): -3 Maksimum na przedziale[0, 5): 8 Minimum na przedziale[2, 5): -3 Maksimum na przedziale[2, 5): 8 Minimum na przedziale[7, 7): 1000000009 Maksimum na przedziale[7, 7): -1000000009 Minimum na przedziale[7, 8): 7 Maksimum na przedziale[7, 8): 7

Rozdział 4

Teoria grafów

4.1 Reprezentacja grafów

4.1.1 Struktura krawędzi

Listing 4.1: "Struktura krawędzi"

```
1 TEMPL struct Edge {
2     int a, b;
3     T w;
4     ET(int a, int b, T w = 1): a(a), b(b), w(w) {}
5 };
```

Szablon

- **typename T**: typ wag krawędzi

Atrybuty

- **int a**: początek krawędzi
- **int b**: koniec krawędzi
- **T w**: waga krawędzi

Edge(int, int, T) - konstruktor

Argumenty

- **int a**: początek krawędzi
- **int b**: koniec krawędzi
- **T w**: waga krawędzi (domyślnie 1)

Złożoność

- czasowa: $O(1)$
- pamięciowa: $O(1)$

Uwagi

- Jeżeli krawędzie grafu nie posiadają wagi, można nie podawać argumentu w. Krawędź przyjmie wtedy jako wagę domyślną wartość 1.

4.1.2 Struktura grafu

Listing 4.2: "Struktura grafu"

```
1  TEMPL struct Graph {
2      VVE adj;
3      VE E;
4      int n;
5      T inf;
6      Graph(int n = 0, T inf = INF): n(n), inf(inf) {
7          adj.resize(n);
8      }
9      void addEdgeD(int a, int b, int w = 1) {
10         ET *e = new ET(a, b, w);
11         E.PB(e);
12         adj[a].PB(e);
13     }
14     void addEdgeD(ET e) {
15         addEdgeD(e.a, e.b, e.w);
16     }
17     void addEdge(int a, int b, int w = 1) {
18         addEdgeD(a, b, w);
19         if (a != b) adj[b].PB(new ET(b, a, w));
20     }
21     void addEdge(ET e) {
22         addEdge(e.a, e.b, e.w);
23     }
24 };
```

Szablon

- **typename T**: typ wag krawędzi

Atrybuty

- **vector<vector<Edge<T>*>>adj**[acency list]: lista sąsiedztwa, adj[i]:=lista krawędzi wychodzących z wierzchołka i
- **vector<Edge<T>*>E**: lista wszystkich krawędzi w grafie
- **int n**: ilość wierzchołków w grafie
- **T inf**: określona wartość nieskończoności dla grafu (przykładowo, jeśli nie istnieje żadna ścieżka w grafie łącząca wierzchołki v i u to przyjmujemy, że odległość między nimi wynosi `inf`)

Graph(int, T) - konstruktor

Argumenty

- **int n**: ilość wierzchołków w grafie (domyślnie 0)
- **T inf**: wartość nieskończoności dla grafu (domyślnie $10^9 + 9$)

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

void addEdgeD(int, int, T)

Argumenty

- **int a**: początek nowej krawędzi skierowanej
- **int b**: koniec nowej krawędzi skierowanej
- **T w**: waga nowej krawędzi skierowanej (domyślnie 1)

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

void addEdgeD(Edge<T>)

Argumenty

- **Edge<T>e**: krawędź skierowana, którą chcemy dodać do grafu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

void addEdge(int, int, T)

Argumenty

- **int a**: początek nowej krawędzi nieskierowanej
- **int b**: koniec nowej krawędzi nieskierowanej
- **T w**: waga nowej krawędzi nieskierowanej (domyślnie 1)

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

void addEdge(Edge<T>)

Argumenty

- **Edge<T>e**: krawędź nieskierowana, którą chcemy dodać do grafu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Listing 4.3: "Przykład użycia"

```
1  int main() {
2      Graph<int> g;
3      readGraph(g, true, true);
4      REP(i, g.n) {
5          printf("Sasiedzi wierzcholka %d: ", i);
6          FOREACH(e, g.adj[i])
7              printf("%d ", e->b);
8          printf("\n");
9      }
10     sort(ALL(g.E), [](Edge<int> *e1, Edge<int> *e2)
11         {return e1->w < e2->w;});
12     printf("Lista posortowanych krawedzi:\n");
13     FOREACH(e, g.E)
14         printf("(%d %d %d)\n", e->a, e->b, e->w);
15 }
```

Wejście	Wyjście
5 8	Sasiedzi wierzcholka 0: 1 2
0 1 6	Sasiedzi wierzcholka 1: 2 3
0 2 8	Sasiedzi wierzcholka 2: 4
1 2 4	Sasiedzi wierzcholka 3: 4
1 3 -5	Sasiedzi wierzcholka 4: 3 4
2 4 1	Lista posortowanych krawedzi:
3 4 9	(1 3 -5)
4 3 2	(2 4 1)
4 4 8	(4 3 2)
	(1 2 4)
	(0 1 6)
	(0 2 8)
	(4 4 8)
	(3 4 9)

4.2 Dodatkowe funkcje grafowe

Poniższy rozdział przedstawia funkcje grafowe, które nie są wymagane, ale ułatwiają typowe operacje na grafach.

4.2.1 Wczytywanie grafu

Listing 4.4: "Wczytywanie grafu"

```
1  TEMPL void readGraph(GT &g, bool isD, bool isW, int id = 0) {
2      int n, m, a, b;
3      T w = 1;
4      cin >> n >> m;
5      g = GT(n);
6      REP(i, m) {
7          cin >> a >> b;
8          a -= id;
9          b -= id;
10         if (isW) cin >> w;
11         if (isD) g.addEdgeD(a, b, w);
12         else g.addEdge(a, b, w);
13     }
14 }
```

Argumenty

- **Graph<T>&g**[raph]: dowolny graf, który chcemy wczytać
- **bool isD**[irected]: `true`, gdy wczytywany graf ma być skierowany, `false` wpw.

- **bool isW**[eighted]: `true`, gdy wczytywany graf ma być ważony; `false` wpw.
- **int id**: 0 lub 1 w zależności, od której liczby indeksujemy wierzchołki (domyślnie 0)

Złożoność

- czasowa: O(m)
- pamięciowa: O($n + m$)

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

4.2.2 Tworzenie grafu z listy krawędzi

Listing 4.5: "Tworzenie grafu z listy krawędzi"

```
1 TEMPL void makeGraph(GT &g, VE E, bool isD, bool isW) {
2     g = GT(g.n);
3     FOREACH(e, E) {
4         ET edge(e->a, e->b, isW ? e->w : 1);
5         if (isD) g.addEdgeD(edge);
6         else g.addEdge(edge);
7     }
8 }
```

Argumenty

- **Graph**<**T**>**g**[raph]: dowolny graf
- **vector**<**Edge**<**T**>***>**E**[dges] - zbiór krawędzi, które chcemy dodać do grafu
- **bool isD**[irected]: `true`, gdy tworzony graf ma być skierowany, `false` wpw.
- **bool isW**[eighted]: `true`, gdy tworzony graf ma być ważony, `false` wpw.

Złożoność

- czasowa: O(m)
- pamięciowa: O($n + m$)

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

4.2.3 Suma wag krawędzi

Listing 4.6: "Suma wag krawędzi"

```
1 TEMPL T sumEdges(VE &E) {
2     T res = 0;
3     FOREACH(e, E)
4         if (e != NP)
5             res += e->w;
6     return res;
7 }
```

Argumenty

- **vector<Edge<T>*>E**[dges] - zbiór krawędzi, dla których chcemy policzyć sumę wag

Zwracana wartość

- **T result** - suma wag krawędzi

Złożoność

- czasowa: O(m)
- pamięciowa: O(1)

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

Uwagi

- Należy uważać na przepełnienie typu T (w pesymistycznym przypadku suma wag wszystkich krawędzi może wynosić $n \cdot W$, gdzie W oznacza limit wag krawędzi).

Listing 4.7: "Przykład użycia"

```
1  int main() {
2      Graph<int> g;
3      readGraph(g, true, true);
4      printf("Suma krawedzi grafu wazonego: %d\n", sumEdges(g.E));
5      makeGraph(g, g.E, true, false);
6      printf("Suma krawedzi grafi niewazonego: %d\n", sumEdges(g.E));
7  }
```

Wejście	**Wyjście**
5 8 0 1 6 0 2 8 1 2 4 1 3 -5 2 4 1 3 4 9 4 3 2 4 4 8	Suma krawedzi grafu wazonego: 33 Suma krawedzi grafi niewazonego: 8

4.3 Przeszukiwanie grafu w głąb

Algorytm DFS (ang. depth-first search) jest to jeden z podstawowych algorytmów z teorii grafów. Jest podstawą wielu algorytmów grafowych i ma wiele zastosowań, gdyż jest (względnie) najwygodniejszą metodą na poruszanie się po grafie.

Uwaga: Poniższy listing zawiera implementacje dwóch funkcji - `dfsR`[ercursive] oraz `dfs`. Zaleca się korzystanie tylko z funkcji `dfs`, ale należy zaimplementować obie funkcje (`dfs` wywołuje `dfsR`). Poniższa dokumentacja tyczy się tylko funkcji `dfs`.

Listing 4.8: "Algorytm DFS"

```
1  TEMPL void dfsR(VE &res, GT &g, int v, int r, ET *e = NP) {
2      if (res[v] == NP) {
3          res[v] = e;
4          FOREACH(e, g.adj[v]) {
5              if (e->b != r) {
6                  dfsR(res, g, e->b, r, e);
7              }
8          }
9      }
10 }
11
```

```
12 TEMPL VE dfs(GT &g, int v = -1) {
13     VE res(g.n, NP);
14     if (v != -1)
15         dfsR(res, g, v, v);
16     else
17         REP(i, g.n)
18             dfsR(res, g, i, i);
19     return res;
20 }
```

Argumenty

- **Graph<T>&g**[raph]: dowolny graf
- **int r**[oot]:
  - indeks korzenia, na którym ma zostać zapuszczony algorytm DFS (przetworzone zostaną wszystkie wierzchołki osiągalne z wierzchołka r), $r \in [0, n)$
  - wartość -1 (wartość domyślna), żeby zapuszczać algorytm DFS dopóki nie zostały przetworzone wszystkie wierzchołki

Wartość zwracana

- **vector<Edge<T>*>E**[dges]: E[v] := wskaźnik na ostatnią krawędź leżącą na znalezionej ścieżce z korzenia do wierzchołka v (`nullptr`, gdy wierzchołek v nie jest osiągalny z korzenia lub wierzchołek v jest korzeniem), |E| = n

Złożoność

- czasowa: O($n + m$)
- pamięciowa: O(n)

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

Uwagi

- W przypadku gdy graf wejściowy jest nieskierowany, wynikiem funkcji jest zbiór krawędzi tworzących:
  - drzewo rozpinające, gdy podano argument r lub graf wejściowy jest spójny

– las drzew rozpinających, wpw.

Listing 4.9: "Przykład użycia"

```
int main() {
    Graph<int> g;
    readGraph(g, true, false);
    auto result = dfs(g, 0);
    printf("Sciezki do korzenia:\n");
    REP(i, g.n) {
        printf("Wierzcholek %d: ", i);
        int v = i;
        while (result[v] != nullptr) {
            printf("%d->", v);
            v = result[v]->a;
        }
        printf("%s\n", v ? "brak" : "0");
    }
}
```

Wejście	Wyjście
5 8 0 1 0 2 1 2 1 3 2 4 3 4 4 3 4 4	Sciezki do korzenia: Wierzcholek 0: 0 Wierzcholek 1: 1->0 Wierzcholek 2: 2->1->0 Wierzcholek 3: 3->4->2->1->0 Wierzcholek 4: 4->2->1->0

4.4 Przeszukiwanie grafu wszerz

Algorytm BFS (ang. breadth-first search) jest to jeden z podstawowych algorytmów z teorii grafów. Jego podstawowe zastosowanie to znalezienie dla każdego wierzchołka najkrótszej ścieżki (pod względem ilości krawędzi na tej ścieżce) do ustalonego wierzchołka (korzenia).

Listing 4.10: "Algorytm BFS"

```
TEMPL PVTVE bfs(GT &g, int r) {
    VT dist(g.n, g.inf);
    VE E(g.n, NP);
    queue<int> q;
    dist[r] = 0;
    q.push(r);
```

```
7       while (!q.empty()) {
8           int v = q.front();
9           q.pop();
10          FOREACH(e, g.adj[v]) {
11              T d = dist[v] + 1;
12              if (d < dist[e->b]) {
13                  dist[e->b] = d;
14                  E[e->b] = e;
15                  q.push(e->b);
16              }
17          }
18      }
19      return MP(dist, E);
20  }
```

Argumenty

- **Graph<T>&g**[raph]: dowolny graf
- **int r**[oot]: indeks korzenia, $r \in [0, n)$

Wartość zwracana

- **pair<vector<T>, vector<Edge<T>*>>result**:
  para<odległości, krawędzie>
  - **vector<T>dist**[ance]: dist[v] := minimalna odległość (w ilościach krawędzi) od korzenia do wierzchołka v (`inf`, gdy wierzhołek v nie jest osiągalny z korzenia), |dist| = n
  - **vector<Edge<T>*>E**[dges]: E[v] := wskaźnik na ostatnią krawędź leżącą na najkrótszej ścieżce z korzenia do wierzchołka v (`nullptr`, gdy wierzchołek v nie jest osiągalny z korzenia lub wierzchołek v jest korzeniem), |E| = n

Złożoność

- czasowa: $O(n + m)$
- pamięciowa: $O(n)$

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

Listing 4.11: "Przykład użycia"

```
int main() {
    Graph<int> g;
    readGraph(g, true, false);
    auto result = bfs(g, 0);
    printf("Odleglosci od korzenia: ");
    printTab(result.ST);
    printf("Sciezki do korzenia:\n");
    REP(i, g.n) {
        printf("Wierzcholek %d: ", i);
        int v = i;
        while (result.ND[v] != NP) {
            printf("%d->", v);
            v = result.ND[v]->a;
        }
        printf("%s\n", v ? "brak" : "0");
    }
}
```

Wejście	**Wyjście**
5 8 0 1 0 2 1 2 1 3 2 4 3 4 4 3 4 4	Odleglosci od korzenia: [0, 1, 1, 2, 2] Sciezki do korzenia: Wierzcholek 0: 0 Wierzcholek 1: 1->0 Wierzcholek 2: 2->0 Wierzcholek 3: 3->1->0 Wierzcholek 4: 4->2->0

4.5 Algorytm Dijkstry

Algorytm dijkstry służy do wyznaczania najkrótszych ścieżek od danego wierzcholka (korzenia) do pozostałych wierzchołków. Odległość wierzchołka od korzenia jest długością najkrótszej ścieżki łączącej te wierzchołki. Jeżeli dla pewnego wierzchołka nie istnieje żadna ścieżka z korzenia to przyjmuje się, że jego odległość wynosi ∞ .

Listing 4.12: "Algorytm Dijkstry"

```
TEMPL PVTVE dijkstra(GT &g, int r) {
    VT dist(g.n, g.inf);
    VE E(g.n, NP);
    priority_queue<pair<T, int> > q;
    dist[r] = 0;
```

```
6       q.push(MP(0, r));
7       while (!q.empty()) {
8           auto x = q.top();
9           q.pop();
10          if (x.ST > dist[x.ND]) continue;
11          FOREACH(e, g.adj[x.ND]) {
12              T d = -x.ST + e->w;
13              if (d < dist[e->b]) {
14                  dist[e->b] = d;
15                  E[e->b] = e;
16                  q.push(MP(-d, e->b));
17              }
18          }
19      }
20      return MP(dist, E);
21  }
```

Argumenty

- **Graph<T>&g**[raph]: graf bez krawędzi o ujemnych wagach
- **int r**[oot]: indeks korzenia, $r \in [0, n)$

Wartość zwracana

- **pair<vector<T>, vector<Edge<T>*>>result**:
  para<odległości, krawędzie>
  - **vector<T>dist**[ance]: dist[v] := minimalna odległość od korzenia do wierzchołka v (`inf`, gdy wierzhołek v nie jest osiągalny z korzenia), |dist| = n
  - **vector<Edge<T>*>E**[dges]: E[v] := wskaźnik na ostatnią krawędź leżącą na najkrótszej ścieżce z korzenia do wierzchołka v (`nullptr`, gdy wierzchołek v nie jest osiągalny z korzenia lub wierzchołek v jest korzeniem), |E| = n

Złożoność

- czasowa: $O(m \cdot log(n))$
- pamięciowa: $O(n + m)$

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

Uwagi

- Odległość wierzchołków jest sumą wag krawędzi, dlatego należy uważać na przepełnienie typu T (w pesymistycznym przypadku odległość między dwoma wierzchołkami może wynosić $n \cdot W$, gdzie W oznacza limit wag krawędzi).

Listing 4.13: "Przykład użycia"

```
1  int main() {
2      Graph<int> g;
3      readGraph(g, true, true);
4      auto result = dijkstra(g, 0);
5      printf("Odleglosci od korzenia: ");
6      printTab(result.ST);
7      printf("Sciezki do korzenia:\n");
8      REP(i, g.n) {
9          printf("Wierzcholek %d: ", i);
10         int v = i;
11         while (result.ND[v] != NP) {
12             printf("%d->", v);
13             v = result.ND[v]->a;
14         }
15         printf("%s\n", v ? "brak" : "0");
16     }
17 }
```

Wejście	**Wyjście**
5 8	Odleglosci od korzenia: [0, 6, 8, 11, 9]
0 1 6	Sciezki do korzenia:
0 2 8	Wierzcholek 0: 0
1 2 4	Wierzcholek 1: 1->0
1 3 5	Wierzcholek 2: 2->0
2 4 1	Wierzcholek 3: 3->1->0
3 4 9	Wierzcholek 4: 4->2->0
4 3 2	
4 4 8	

4.6 Sortowanie topologiczne

Sortowanie topologiczne polega na uporządkowaniu wierzchołków grafu w takiej kolejności, aby dla każdej krawędzi w grafie (v, u), wierzchołek v znajduje się wcześniej w porządku topologicznym niż wierzchołek u. Kolejności takiej nie można uzyskać, gdy graf posiada przynajmniej jeden cykl (dlatego

porządku topologicznego nie można wyznaczyć dla grafów ważonych, gdyż w takim grafie każde 2 wierzchołki połączone krawędzią tworzą cykl). Określony graf może mieć wiele poprawnych porządków topologicznych.

Listing 4.14: "Sortowanie topologiczne"

```
1  TEMPL VI topoSort(GT &g) {
2      VI res;
3      VI rank = VI(g.n, 0);
4      FOREACH(e, g.E)
5          rank[e->b]++;
6      REP(i, g.n)
7          if (rank[i] == 0)
8              res.PB(i);
9      REP(i, res.size())
10         FOREACH(e, g.adj[res[i]])
11             if (--rank[e->b] == 0)
12                 res.PB(e->b);
13     return res;
14 }
```

Argumenty

- **Graph<T>&g**[raph]: graf skierowany bez cykli

Wartość zwracana

- **vector<int>result**: lista wierzchołków w porządku topologicznym, |res| = n

Złożoność

- czasowa: O($n + m$)
- pamięciowa: O(n)

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30

Listing 4.15: "Przykład użycia"

```
1  int main() {
2      Graph<int> g;
3      readGraph(g, false, false);
4      VI result = topoSort(g);
```

```
5       printf("Kolejnosc topologiczna:\n");
6       printTab(result);
7   }
```

Wejście	Wyjście
5 6 0 1 0 2 1 2 1 3 2 4 3 4	Kolejnosc topologiczna: [0, 1, 2, 3, 4]

4.7 Minimalne drzewo rozpinające

Minimalne drzewo rozpinające (ang. minimal spanning tree) jest to podgraf pewnego grafu, który zawiera wszystkie jego wierzchołki, jest spójny, ma strukturę drzewa oraz suma wag krawędzi w tym podgrafie jest możliwie jak najmniejsza. Dany graf może mieć wiele możliwych minimalnych drzew rozpinających.

Listing 4.16: "Minimalne drzewo rozpinające"

```
1   TEMPL VE mst(GT &g) {
2       VE res;
3       VE E = g.E;
4       sort(ALL(E), [](ET *e1, ET *e2){return e1->w < e2->w;});
5       FU fu(g.n);
6       FOREACH(e, E)
7           if (fu.Union(e->a, e->b))
8               res.PB(e);
9       return res;
10  }
```

Argumenty

- **Graph<T>&g**[raph]: spójny, ważony graf nieskierowany

Wartość zwracana

- **vector<Edge<T>*>result**: zbiór krawędzi, tworzących minimalne drzewo rozpinające, $|\text{result}| = n - 1$

Złożoność

- czasowa: $O(m \cdot log(m))$
- pamięciowa: $O(m)$

Wymagania

- struktura Edge listing 4.1, strona 29
- struktura Graph listing 4.2, strona 30
- struktura FU listing 7.1, strona 77

Uwagi

- W przypadku, gdy wejściowy graf nie będzie spójny, funkcja zwróci zbiór krawędzi tworzących las, gdzie każda spójna składowa będzie minimalnym drzewem rozpinającym dla pewnej spójnej składowej z grafu wejściowego.
- W przypadku, gdy wejściowy graf będzie skierowany, algorytm potraktuje go tak jakby był nieskierowany (każda krawędź będzie 2-kierunkowa).
- W przypadku, gdy wejściowy graf nie będzie ważony, każde drzewo rozpinające na tym grafie będzie minimalnym. Zaleca się wtedy użycie funkcji `dfs(Graph<T>, int)` listing 4.8, strona 36.

Listing 4.17: "Przykład użycia"

```
1 int main() {
2     Graph<int> g;
3     readGraph(g, true, true);
4     printf("Krawedzie tworzace mst:\n");
5     FOREACH(e, mst(g))
6         printf("(%d %d %d)\n", e->a, e->b, e->w);
7 }
```

Wejście	**Wyjście**
5 8 0 1 6 0 2 8 1 2 4 1 3 5 2 4 1 3 4 9 4 3 2 4 4 8	Krawedzie tworzace mst: (2 4 1) (4 3 2) (1 2 4) (0 1 6)

Rozdział 5

Algorytmy tekstowe

5.1 Algorytm KMP

Algorytm KMP (Knutha-Morissa-Pratta) w swojej podstawowej formie polega na wyznaczeniu tzw. tablicy prefikso-sufiksów, czyli tablicy, która dla każdego prefiksu podanego słowa zawiera informację o długości najdłuższego prefikso-sufisku właściwego (podsłowa, które występuje zarówno na początku jak i na końcu słowa). Z takiej tablicy można uzyskać wiele informacji m.in. ilość wystąpień wzorca w tekście, długość najkrótszego okresu słowa, długość najdłuższego szablonu itp.

Listing 5.1: "Algorytm KMP"

```
1  VI kmp(string s) {
2      VI res(s.size()+1, 0);
3      int x = 0;
4      LOOP(i, 2, res.size()) {
5          while (x && s[i-1] != s[x]) x = res[x];
6          if (s[i-1] == s[x]) x++;
7          res[i] = x;
8      }
9      return res;
10 }
```

Argumenty

- **string s**: dowolne słowo, dla którego chcemy wyznaczyć tablicę prefikso sufiksów, $n := |s| \in [0, \sim 10^7]$

Wartość zwracana

- **vector<int>result**: result[i] := długość najdłuższego prefikso-sufiksu właściwego podsłowa s[0..i-1], $i \in [0, n]$, $|\text{result}| = n + 1$

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

Listing 5.2: "Przykład użycia"

```
1 int main() {
2     string s;
3     while (cin >> s) {
4         VI result = kmp(s);
5         printf("Tablica KMP: ");
6         printTab(result);
7     }
8 }
```

Wejście	Wyjście
abacabab	Tablica KMP: [0, 0, 0, 1, 0, 1, 2, 3, 2]
abcdefghij	Tablica KMP: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
xxxxxxxx	Tablica KMP: [0, 0, 1, 2, 3, 4, 5, 6, 7]
pies#pies_i_kot	Tablica KMP: [0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0]
slowoslowoslowo	Tablica KMP: [0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

5.2 Wyszukiwanie wzorca w tekście

Do znalezienia wszystkich wystąpień określonego wzorca w tekście, można użyć opisanego w poprzednim rozdziale algorytmu KMP. Wystarczy obliczyć tablicę prefikso-sufiksów na słowie `wzorzec#tekst` (gdzie znak `#` jest dowolnym znakiem, który nie występuje ani we wzorcu ani w tekście). Dzięki tej sztuczce, każde występienie w tablicy prefikso-sufiksów wartości |wzorzec| jest równoznaczne z występowaniem wzorca w badanym tekście.

Listing 5.3: "Wyszukiwanie wzorca w tekście"

```
1 VI findPattern(string p, string t) {
2     VI res;
3     VI tab = kmp(p+"#"+t);
4     int ps = p.size();
5     REP(i, tab.size())
6         if (tab[i] == ps)
7             res.PB(i-2*ps-1);
```

```
8       return res;
9   }
```

Argumenty

- **string p**[attern]: wzorzec, który chcemy wyszukać w tekście (bez wystąpień znaku #), $n := |\text{p}| \in [1, \sim 10^7]$
- **string t**[ext]: tekst, w którym szukamy wzorca (bez wystąpień znaku #), $m := |\text{t}| \in [0, \sim 10^7]$

Wartość zwracana

- **vector<int>result**: posortowana tablica wszystkich wystąpień wzorca w tekście (indeksy pierwszych znaków), $|\text{result}| \leqslant m - n + 1$

Złożoność

- czasowa: $O(n + m)$
- pamięciowa: $O(n + m)$

Wymagania

- algorytm KMP listing 5.1, strona 46

Uwagi

- W przypadku, gdy znak # może wystąpić we wzorcu lub w tekście, należy w 3 linii listingu 5.3 zastąpić go innym znakiem, który nie występuje w tych słowach

Listing 5.4: "Przykład użycia"

```
1   int main() {
2       string pattern, text;
3       while (cin >> pattern >> text) {
4           VI result = findPattern(pattern, text);
5           printf("Wystapienia slowa %s: ", pattern.c_str());
6           printTab(result);
7       }
8   }
```

Wejście	Wyjście
xyx xyxyxyx	Wystapienia slowa xyx: [0, 2, 4]
ab abaab	Wystapienia slowa ab: [0, 3]
xxx xxxxxx	Wystapienia slowa xxx: [0, 1, 2, 3]
kot Ala_ma_kota_a_kot_ma_Ale	Wystapienia slowa kot: [7, 14]
a aAaAaAa	Wystapienia slowa a: [0, 2, 4, 6]

5.3 Minimalny okres słowa

Minimalny okres słowa jest to najkrótszy prefiks właściwy p pewnego słowa wejściowego s, taki, że s jest prefiksem słowa p^k, $k \in n$ (k-krotna konkatenacja słowa p). Potocznie mówiąc, okres słowa jest to podsłowo, z którego możemy uzyskać słowo wejściowe za pomocą operacji konkatenacji.

Listing 5.5: "Minimalny okres słowa"

```
1 int minPeriod(string &s) {
2     return s.size() - kmp(s).back();
3 }
```

Argumenty

- **string s**: dowolne słowo, dla którego chcemy wyznaczyć minimalny okres, $n := |s| \in [0, \sim 10^7]$

Wartość zwracana

- **int result**: długość minimalnego okresu słowa s, result $\in [0, n]$

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

Wymagania

- algorytm KMP listing 5.1, strona 46

Listing 5.6: "Przykład użycia"

```
1 int main() {
2     string s;
3     while (cin >> s) {
4         printf("Minimalny okres slowa %s: %s\n",
5             s.c_str(), s.substr(0, minPeriod(s)).c_str());
6     }
7 }
```

Wejście	Wyjście
slowoslowoslowoslowo	Minimalny okres slowa slowoslowoslowoslowo: slowo
slowoslowoslowoslo	Minimalny okres slowa slowoslowoslowoslo: slowo
abcdefghij	Minimalny okres slowa abcdefghij: abcdefghij
xxxxxxxxxx	Minimalny okres slowa xxxxxxxxxx: x
aba	Minimalny okres slowa aba: ab

5.4 Aho-Corasick

Algorytm Aho-Corasick umożliwia znalezienie wszystkich wystąpień wielu wzorców w jednym tekście za pomocą struktury Trie.

5.4.1 Struktura pomocnicza

Listing 5.7: "Struktura pomocnicza"

```
1  struct TNode {
2      vector<TNode*> cld;
3      TNode* fail = NP;
4      TNode* suf = NP;
5      VI P;
6
7      TNode(int ch){
8          cld.resize(ch, NP);
9      }
10 };
```

Atrybuty

- **vector<TNode*> cld**[child]: tablica wskaźników do węzłów potomnych
- **TNode *fail**: wskaźnik do węzła reprezentującego najdłuższy sufiks lokalnego słowa (krawędź powrotu)
- **TNode *suf**[ix]: wskaźnik do węzła, reprezentującego najdłuższy sufiks kolalnego słowa, który należy do zbioru wzorców
- **vector<int>P**[atterns]: indeksy wzorców, które kończą się w danym węźle

TNode(int) - konstruktor

Argumenty

- **int ch**[ars]: ilość różnych znaków w alfabecie

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

5.4.2 Drzewo Trie

Listing 5.8: "Drzewo Trie"

```
1  struct Trie {
2      TNode *root;
3      VS P;
4      int ch;
5
6      Trie(VS P, int ch = 62): ch(ch) {
7          root = new TNode(ch);
8          FOREACH(p, P) insert(p);
9          setFails();
10     }
11
12     int id(char c) {
13         if ('a' <= c && c <= 'z') return c-'a';
14         else if ('A' <= c && c <= 'Z') return c-'A'+26;
15         else return c-'0'+52;
16     }
17
18     void insert(string s) {
19         TNode *node = root;
20         FOREACH(c, s) {
21             if (node->cld[id(c)] == NP)
22                 node->cld[id(c)] = new TNode(ch);
23             node = node->cld[id(c)];
24         }
25         node->P.PB(P.size());
26         P.PB(s);
27     }
28
29     void setFails() {
30         queue <TNode*> q;
31         root->fail = root;
```

```
32          q.push(root);
33          while (!q.empty()) {
34              TNode* node = q.front();
35              q.pop();
36              REP(i, ch) {
37                  TNode *cld = node->cld[i];
38                  if (cld != NP) {
39                      if (node != root) {
40                          TNode *tmp = node->fail;
41                          while (tmp != root && tmp->cld[i] == NP)
42                              tmp = tmp->fail;
43                          cld->fail = (tmp->cld[i] == NP) ?
44                              root : tmp->cld[i];
45                          cld->suf = cld->fail->P.empty() ?
46                              cld->fail->suf : cld->fail;
47                          q.push(cld);
48                      }
49                      else {
50                          cld->fail = root;
51                          q.push(cld);
52                      }
53                  }
54              }
55          }
56      }
57
58      VVI search(string s) {
59          VVI res(P.size());
60          TNode *node = root;
61          REP(i, s.size()) {
62              char c = s[i];
63              while (node != root && node->cld[id(c)] == NP)
64                  node = node->fail;
65              if (node->cld[id(c)] != NP)
66                  node = node->cld[id(c)];
67              TNode *tmp = node;
68              do {
69                  FOREACH(p, tmp->P)
70                      res[p].PB(i-P[p].size()+1);
71                  tmp = tmp->suf;
72              } while (tmp != NP);
73          }
74          return res;
75      }
76  };
```

Atrybuty

- **TNode* root**: wskaźnik do korzenia struktury
- **vector<string>P**[atterns]: lista wzorców
- **int ch**[ars]: ilość różnych znaków w alfabecie

Trie(vector<string>, int) - konstruktor

Argumenty

- **vector<string>P**[atterns]: lista (niepustych) wzorców (wzorce mogą się powtarzać), $n := \sum_{p \in P} |p|$, $n \in [0, \sim 10^6]$
- **int ch**[ars]: ilość znaków w alfabecie (domyślnie 62: 26 małych liter + 26 wielkich liter + 10 cyfr)

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

int id(char)

Argumenty

- **char c**: znak z alfabetu

Wartość zwracana

- **int res**[ult]: unikalna liczba przypisana do danego znaku, res $\in [0, ch)$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Uwagi

- Metoda ta przystosowana jest dla alfabetu złożonego z małych i dużych liter oraz cyfr. W przypadku niestandardowych alfabetów, zaleca się samodzielnie zaimplementować ciało tej funkcji, tak aby każdemu znaku z alfabetu przyporządkowywała unikalną liczbę z przedziału $[0, ch)$.

void insert(string)

Argumenty

- **string s** słowo, które chcemy dodać do listy wzorców

Złożoność

- czasowa: O(|s|)
- pamięciowa: O(|s|)

Uwagi

- Jest to raczej funkcja wewnętrzna (używana z poziomu konstruktora). Nie zaleca się używania tej metody w sposób bezpośredni.
- W przypadku bezpośredniego używania tej metody, należy pamiętać o zaaktualizowaniu wskaźników `TNode::fail` i `TNode::suf` za pomocą metody `setFails()`.

void setFails()

Opis
Aktualizuje wskaźniki `TNode::fail` i `TNode::suf` dla całej struktury.

Złożoność

- czasowa: O(n)
- pamięciowa: O(1)

vector<vector<int>>search(string)

Argumenty

- **string s**: tekst, w którym chcemy znaleźć wystąpienia wzorców, $m := |s| \in [0, \sim 10^6]$

Wartość zwracana

- **vector<vector<int>>result**: `result[i]` := posortowana tablica wszystkich wystąpień i-tego wzorca w tekście s (indeksy pierwszych znaków), $i \in [0, |P|)$, $|result| = |P|$

Złożoność

- czasowa: $O(n+m)$
- pamięciowa: $O(1)$

Listing 5.9: "Przykład użycia"

```
1  int main() {
2      VS patterns;
3      readTab(patterns);
4      string text;
5      cin >> text;
6      Trie trie(patterns);
7      VVI result = trie.search(text);
8      REP(i, result.size()) {
9          printf("Wystapienia slowa %s: ", patterns[i].c_str());
10         printTab(result[i]);
11     }
12 }
```

Wejście	**Wyjście**
4 she he hers his hishers	Wystapienia slowa she: [2] Wystapienia slowa he: [3] Wystapienia slowa hers: [3] Wystapienia slowa his: [0]

Rozdział 6

Geometria 2D

Poniższe struktury zawierają reprezentację obiektów geometrycznych takich jak punkt, wektor, prosta, półprosta, odcinek, okrąg, koło, wielokąt. Struktury posiadają metody umożliwiające wykonywanie podstawowych operacji na tych obiektach.

6.0.1 Punkt

Listing 6.1: "Punkt"

```
1  struct Point {
2      double x, y;
3
4      Point(): x(0), y(0) {}
5
6      Point(double x, double y): x(x), y(y) {}
7
8      double dist(Point p = Point()) {
9          p = p - *this;
10         return sqrt(p.x*p.x + p.y*p.y);
11     }
12
13     Point rotate(double a, Point c = Point()) {
14         Point p = *this - c;
15         return c + Point(p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a));
16     }
17
18     Point operator +(Point p) {
19         return Point(x + p.x, y + p.y);
20     }
21
22     Point operator -(Point p) {
```

```
23          return Point(x − p.x, y − p.y);
24      }
25
26      Point operator ∗(double a) {
27          return Point(a∗x, a∗y);
28      }
29
30      bool operator <(Point p) {
31          return MP(x, y) < MP(p.x, p.y);
32      }
33
34      bool operator ==(Point p) {
35          p = p − ∗this;
36          return ISZERO(p.x) && ISZERO(p.y);
37      }
38  };
```

Atrybuty

- **double x**: pierwsza współrzędna punktu
- **double y**: druga współrzędna punktu

Point(double, double) - konstruktor

Argumenty

- **double x**: pierwsza współrzędna punktu
- **double y**: druga współrzędna punktu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double dist(Point)

Argumenty

- **Point p**: punkt, od którego chcemy obliczyć rzeczywistą odległość

Wartość zwracana

- **double result**: rzeczywista odległość między dwoma punktami

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Point rotate(double, Point)

Argumenty

- **double a**[ngle] - kąt o który chcemy obrócić punkt
- **Point c**[enter] - punkt względem którego chcemy dokonać obrotu

Wartość zwracana

- **Point result**: punkt obrócony o kąt a względem puntku c

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Point operator +(Point)

Argumenty

- **Point p** - punkt, którego współrzędne chcemy dodać

Wartość zwracana

- **Point res** punkt o współrzędnych równych sumie współrzędnych dodawanych punktów

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Point operator -(Point)

Argumenty

- **Point p** - punkt, którego współrzędne chcemy odjąć

Wartość zwracana

- **Point result**: punkt o współrzędnych równych różnicy współrzędnych odejmowanych punktów

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Point operator *(double)

Argumenty

- **double a** - liczba, przez którą chcemy przemnożyć współrzędne punktu

Wartość zwracana

- **Point result** punkt o współrzędnych przemnożonych przez podaną liczbę

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

bool operator <(Point)

Argumenty

- **Point p** - punkt, który chcemy porównać z bazowym punktem

Wartość zwracana

- wartość `true`, gdy bazowy punkt ma mniejszą pierwszą współrzędną, a w przypadku równych wartości rzędnych, gdy bazowy punkt ma mniejszą drugą współrzędną
- wartość `false`, wpw.

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

bool operator ==(Point)

Argumenty

- **Point p** - punkt, który chcemy porównać z bazowym punktem

Wartość zwracana

- wartość `true`, gdy porównywane punkty mają równe współrzędne
- wartość `false`, wpw.

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Listing 6.2: "Przykład użycia"

```
int main() {
    int x, y;
    cin >> x >> y;
    Point A(x, y);
    cin >> x >> y;
    Point B(x, y);
    double angle;
    cin >> angle;
    printf("Odleglosc punktow: %.2f\n", A.dist(B));
    Point P = A.rotate(angle*M_PI/180, B);
    printf("Punkt A obrocony o kat %.2f wzgledem punktu B: (%.2f, %.2f)\n",
        angle, P.x, P.y);
    P = A + B;
    printf("A + B = (%.2f, %.2f)\n", P.x, P.y);
    P = A - B;
    printf("A - B = (%.2f, %.2f)\n", P.x, P.y);
    P = A * 2;
    printf("A * 2 = (%.2f, %.2f)\n", P.x, P.y);
    if (A < B) printf("A < B\n");
    if (A == B) printf("A == B\n");
    if (B < A) printf("A > B\n");
}
```

Wejście	Wyjście
1 1 3 2 90	Odleglosc punktow: 2.24 Punkt A obrocony o kat 90.00 wzgledem punktu B: (4.00, 0.00) A + B = (4.00, 3.00) A - B = (-2.00, -1.00) A * 2 = (2.00, 2.00) A < B

6.0.2 Wektor

Wymagania

- struktura `Point` listing 6.1, strona 56

Listing 6.3: "Wektor"

```
1  struct Vector {
2      double a, b;
3      Point s;
4
5      Vector(): a(0), b(0) {}
6
7      Vector(double a, double b): a(a), b(b) {}
8
9      Vector(Point p1, Point p2): s(p1) {
10         p1 = p2 - p1;
11         a = p1.x;
12         b = p1.y;
13     }
14
15     double length() {
16         return Point(a, b).dist();
17     }
18
19     double scalarProduct(Vector v) {
20         return a*v.a + b*v.b;
21     }
22
23     double det(Vector v) {
24         return a*v.b - b*v.a;
25     }
26
27     double alpha(Vector v) {
28         double res = acos(scalarProduct(v) / (length() * v.length()));
29         if (det(v) >= 0) {
30             return res;
```

```
31          }
32          else {
33              return 2*M_PI - res;
34          }
35      }
36  };
```

Atrybuty

- **double a**: pierwsza współrzędna wektora
- **double b**: druga współrzędna wektora
- **Point s**: początek wektora

Vector (double, double) - konstruktor

Argumenty

- **double a**: pierwsza współrzędna wektora
- **double b**: druga współrzędna wektora

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Vector (Point, Point) - konstruktor

Argumenty

- **Point p1**: początek wektora
- **Point p2**: koniec wektora

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double length()

Wartość zwracana

- **double result**: rzeczywista długość wektora

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double scalarProduct(Vector)

Argumenty

- **Vector v**: wektor, dla którego chcemy iloczyn skalarny wektorów

Wartość zwracana

- **double result**: wartość iloczynu skaralnego 2-óch wektorów

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double det(Vector)

Argumenty

- **Vector v**: wektor, dla którego chcemy wyliczyć wyznacznik (dlugość iloczynu wektorowego)

Wartość zwracana

- **double result**: wyznacznik (długość iloczynu wektorowego) dwóch wektorów

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double alpha(Vector)

Argumenty

- **Vector v**: wektor, dla którego chcemy kąt między wektorami

Wartość zwracana

- **double result**: kąt między wektorami wyrażony w radianach, result $\in [0, 2\pi)$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Listing 6.4: "Przykład użycia"

```
1  int main() {
2      int x, y;
3      cin >> x >> y;
4      Point p1(x, y);
5      cin >> x >> y;
6      Point p2(x, y);
7      Vector v1(p1, p2);
8      cin >> x >> y;
9      Vector v2(x, y);
10     printf("Wspolrzedne wektorow: [%.2f, %.2f] [%.2f, %.2f]\n",
11         v1.a, v1.b, v2.a, v2.b);
12     printf("Dlugosc wektorow: %.2f %.2f\n", v1.length(), v2.length());
13     printf("Iloczyn skalarny wektorow: %.2f\n", v1.scalarProduct(v2));
14     printf("Wyznacznik wektorow: %.2f\n", v1.det(v2));
15     printf("Kat pomiedzy wektorami: %.2f\n", v1.alpha(v2)*180/M_PI);
16 }
```

Wejście	**Wyjście**
3	Wspolrzedne wektorow: [-3.00, 0.00] [4.00, 0.00]
0 0	Dlugosc wektorow: 3.00 4.00
0 4	Iloczyn skalarny wektorow: -12.00
0 3	Wyznacznik wektorow: -0.00
0 0 1 100	Kat pomiedzy wektorami: 180.00

6.0.3 Prosta, półprosta, odcinek

Wymagania

- struktura `Point` listing 6.1, strona 56
- struktura `Vector` listing 6.3, strona 61

Uwaga: Poniższa struktura może reprezentować prostą, półprostą lub odcinek. W dokumentacji zastosowałem pojęcie **linii**, które określa dowolny wyżej wymieniony obiekt.

Listing 6.5: "Linia"

```
1  struct Line {
2      double a, b, c;
3      int pts;
4      VP P;
5
6      Line(): a(0), b(0), c(0), pts(0) {}
7
8      Line(double a, double b, double c): a(a), b(b), c(c), pts(0) {}
9
10     Line(Point p1, Point p2, int pts = 0): pts(pts), P({p1, p2}){
11         double dx = p1.x - p2.x;
12         double dy = p1.y - p2.y;
13         a = -dy;
14         b = dx;
15         c = -(a*p1.x + b*p1.y);
16         if (pts == 1) {
17             P[1].x = p1.x < p2.x ? INF : -INF;
18             P[1].y = p1.y < p2.y ? INF : -INF;
19         }
20     }
21
22     double dist(Point p) {
23         if (pts) {
24             REP(i, 2) {
25                 Vector v1(P[i], P[1-i]), v2(P[i], p);
26                 if (min(v1.alpha(v2), v2.alpha(v1)) > M_PI_2) {
27                     return min(p.dist(P[0]), p.dist(P[1]));
28                 }
29             }
30         }
31         return abs(a*p.x + b*p.y + c) / sqrt(a*a + b*b);
32     }
33
34     double length() {
35         return (pts == 2) ? P[0].dist(P[1]) : INF;
36     }
37
38     Line perpendicular(Point p) {
39         return Line(b, -a, -b*p.x + a*p.y);
40     }
```

```
41
42      Line parallel(Point p) {
43          return Line(a, b, -a*p.x - b*p.y);
44      }
45
46      bool hasPoint(Point p) {
47          return ISZERO(dist(p));
48      }
49
50      VP getPointsWithX(double x) {
51          if (ISZERO(b)) return {};
52          Point p = Point(x, -(a*x+c)/b);
53          return hasPoint(p) ? VP(1, p) : VP();
54      }
55
56      VP getPointsWithY(double y) {
57          if (ISZERO(a)) return {};
58          Point p = Point(-(b*y+c)/a, y);
59          return hasPoint(p) ? VP(1, p) : VP();
60      }
61  };
```

Atrybuty

- **double a, b, c**: parametry równania linii
- **int pts**[points]: ilość punktów ograniczających tę linię (0 - prosta, 1 - półprosta, 2 - odcinek)
- **vector<Point>P**[oints]: zbiór punktów ograniczających tę linię (w przypadku półprostej jeden z punktów ma postać (∞, ∞), $(\infty, -\infty)$, $(-\infty, \infty)$ lub $(-\infty, -\infty)$,)

Point(double, double, double) - konstruktor

Argumenty

- **double a**: pierwszy parametr równania linii
- **double b**: drugi parametr równania linii
- **double c**: trzeci parametr równania linii

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Point(Point, Point) - konstruktor

Argumenty

- **Point p1**: pierwszy punkt leżacy na tworzonej linii
- **Point p2**: drugi punkt (różny od pierwszego) leżący na tworzonej prostej

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double dist(Point p)

Argumenty

- **Point p**: punkt, dla którego chcemy wyliczyć odległość od linii

Wartość zwracana

- **double result**: odległość podanego punktu od prostej (długość najkrótszego odcinka łączącego podany punkt z dowolnym punktem leżącym na linii)

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double length()

Wartość zwracana

- **double result**: długość linii (∞, w przypadku prostej lub półprostej)

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Line perpendicular(Point)

Wartość zwracana

- **Point p**: punkt, który ma leżeć na nowej prostej prostopadłej

Wartość zwracana

- **Line result**: prosta prostopadła do danej przechodząca przez punkt p

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Line parallel(Point)

Wartość zwracana

- **Point p**: punkt, który ma leżeć na nowej prostej równoległej

Wartość zwracana

- **Line result**: prosta równoległa do danej przechodząca przez punkt p

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

bool hasPoint(Point)

Wartość zwracana

- **Point p**: punkt, który chcemy sprawdzić czy leży na danej linii

Wartość zwracana

- wartość `true`, jeżeli punkt p leży na danej linii
- wartość `false`, wpw.

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

vector<Point>getPointsWithX(double)

Argumenty

- **double x**: wartość rzędnej punktów

Wartość zwracana

- **vector<Point>result**: zbiór punktów leżących na linii o pierwszej współrzędnej równej x, $|\text{res}| \in 0, 1$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

vector<Point>getPointsWithY(double)

Argumenty

- **double y**: wartość odciętej punktów

Wartość zwracana

- **vector<Point>res**[ult]: zbiór punktów leżących na linii o drugiej współrzędnej równej y, $|\text{res}| \in 0, 1$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Listing 6.6: "Przykład użycia"

```
int main() {
    double x, y;
    Point A, B, C;
    cin >> x >> y;
    A = Point(x, y);
    cin >> x >> y;
    B = Point(x, y);
    cin >> x >> y;
    C = Point(x, y);
    Line l(A, B, 2);
    printf("Rownanie odcinka AB: %.2fx %.2fy + %.2f = 0\n", l.a, l.b, l.c);
    printf("Dlugosc odcinka AB: %.2f\n", l.length());
    printf("Odleglosc punktu C od odcinka AB: %.2f\n", l.dist(C));
    printf("Czy punkt C lezy na odcinku AB?: %s\n",
        l.hasPoint(C) ? "TAK" : "NIE");
    Line k = l.perpendicular(C);
    printf("Rownanie prostej prostopadlej: %.2fx %.2fy + %.2f = 0\n",
        k.a, k.b, k.c);
```

```
19      Line m = l.parallel(C);
20      printf("Rownanie prostej rownoleglej: %.2fx %.2fy + %.2f = 0\n",
21          m.a, m.b, m.c);
22      printf("Punkt lezacy na odcinku AB dla x=%.2f: ", C.x);
23      VP P = l.getPointsWithX(C.x);
24      if (P.empty()) printf("brak\n");
25      else printf("(%.2f, %.2f)\n", P[0].x, P[0].y);
26      printf("Punkt lezacy na odcinku AB dla y=%.2f: ", C.y);
27      P = l.getPointsWithY(C.y);
28      if (P.empty()) printf("brak\n");
29      else printf("(%.2f, %.2f)\n", P[0].x, P[0].y);
30  }
```

Wejście	**Wyjście**
1 1 3	Rownanie odcinka AB: -0.00x -2.00y + 2.00 = 0 Dlugosc odcinka AB: 2.00 Odleglosc punktu C od odcinka AB: 0.00 Czy punkt C lezy na odcinku AB?: TAK Rownanie prostej prostopadlej: -2.00x 0.00y + 6.00 = 0 Rownanie prostej rownoleglej: -0.00x -2.00y + 2.00 = 0 Punkt lezacy na odcinku AB dla x=3.00: (3.00, 1.00) Punkt lezacy na odcinku AB dla y=1.00: brak

6.0.4 Okrąg

Wymagania

- struktura `Point` listing 6.1, strona 56

Listing 6.7: "Okrąg"

```
1   struct Circle {
2       double a, b, c;
3       double r;
4       Point s;
5
6       Circle(): a(0), b(0), c(0), r(0) {}
7
8       Circle(double a, double b, double c): a(a), b(b), c(c) {
```

```
9           s = Point(a, b) * (-0.5);
10          r = sqrt(s.x*s.x + s.y*s.y - c);
11      }
12
13      Circle(Point s, double r): r(r), s(s){
14          a = -2*s.x;
15          b = -2*s.y;
16          c = s.x*s.x + s.y*s.y - r*r;
17      }
18
19      Circle(Point s, Point p): s(s) {
20          *this = Circle(s, s.dist(p));
21      }
22
23      double area() {
24          return M_PI * r * r;
25      }
26
27      double circuit() {
28          return 2 * M_PI * r;
29      }
30
31      VP getPointsWithX(double x) {
32          VP res;
33          FOREACH(y, rqe(1, b, x*x + a*x + c)) {
34              res.PB(Point(x, y));
35          }
36          return res;
37      }
38
39      VP getPointsWithY(double y) {
40          VP res;
41          FOREACH(x, rqe(1, a, y*y + b*y + c)) {
42              res.PB(Point(x, y));
43          }
44          return res;
45      }
46  };
```

Atrybuty

- **double a, b, c**: parametry równania okręgu
- **double r**: długość promienia okręgu
- **Point s**: środek okręgu

Circle(double, double, double) - konstruktor

Argumenty

- **double a**: pierwszy parametry równania okręgu
- **double b**: drugi parametr równania okręgu
- **double c**: trzeci parametr równania okręgu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Circle(Point, double) - konstruktor

Argumenty

- **Point s**: środek okręgu
- **double r**: dlugość promienia okręgu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Circle(Point, Point) - konstruktor

Argumenty

- **Point s**: środek okręgu
- **Point p**: dowolny punkt leżący na okręgu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double area()

Wartość zwracana

- **double result**: pole koła

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

double circuit()

Wartość zwracana

- **double res**: obwód okręgu

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

vector<Point>getPointsWithX(double)

Argumenty

- **double x**: wartość rzędnej punktów

Wartość zwracana

- **vector<Point>result**: zbiór punktów należacych do okręgu o pierwszej współrzędnej równej x, $|\text{res}| \in 0, 1, 2$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Wymagania

- funkcja `rqe(double, double, double)` listing 2.13, strona 20

vector<Point>getPointsWithY(double)

Argumenty

- **double y**: wartość odciętej punktów

Wartość zwracana

- **vector<Point>result**: zbiór punktów należacych do okręgu o drugiej współrzędnej równej y, $|\text{res}| \in 0, 1, 2$

Złożoność

- czasowa: O(1)
- pamięciowa: O(1)

Wymagania

- funkcja `rqe(double, double, double)` listing 2.13, strona 20

Listing 6.8: "Przykład użycia"

```
1  int main() {
2      double x, y, r;
3      cin >> x >> y >> r;
4      Circle c = Circle(Point(x, y), r);
5      printf("Rownanie okregu: x ^ 2 + y ^ 2 + %.2fx + %.2fy + %.2f = 0\n",
6          c.a, c.b, c.c);
7      printf("Pole kola: %.2f\n", c.area());
8      printf("Obwod kola: %.2f\n", c.circuit());
9      FOREACH(x, VD({c.s.x, c.s.x + c.r, c.s.x + c.r + 1})) {
10         printf("Punkty lezace na okregu dla x=%.2f: ", x);
11         VP P = c.getPointsWithX(x);
12         if (P.empty()) printf("brak\n");
13         else FOREACH(p, P) printf("(%.2f, %.2f) ", p.x, p.y);
14         printf("\n");
15     }
16 }
```

Wejście	Wyjście
2 3 4	Rownanie okregu: x ^ 2 + y ^ 2 + -4.00x + -6.00y + -3.00 = 0 Pole kola: 50.27 Obwod kola: 25.13 Punkty lezace na okregu dla x=2.00: (2.00, -1.00) (2.00, 7.00) Punkty lezace na okregu dla x=6.00: (6.00, 3.00) Punkty lezace na okregu dla x=7.00: brak

6.0.5 Wielokąt

Wymagania

- struktura `Point` listing 6.1, strona 56

- struktura `Vector` listing 6.3, strona 61
- struktura `Line` listing 6.5, strona 65

Listing 6.9: "Wielokąt"

```
1  struct Polygon {
2      VP P;
3      vector <Line> E;
4      int n;
5
6      Polygon(VP P): P(P), n(P.size()) {
7          REP(i, n) {
8              E.PB(Line(P[i], P[(i+1)%n], 2));
9          }
10     }
11
12     double area() {
13         double res = 0;
14         REP(i, n-2) {
15             Vector v1 = Vector(P[n-1], P[i]);
16             Vector v2 = Vector(P[n-1], P[i+1]);
17             res += v1.det(v2);
18         }
19         return abs(res)/2;
20     }
21
22     double circuit() {
23         double res = 0;
24         FOREACH(e, E) {
25             res += e.length();
26         }
27         return res;
28     }
29 };
```

Atrybuty

- **vector<Point>P**[oints]: lista kolejnych wierzchołków wielokąta
- **vector<Line> E**[dges]: lista krawędzi wielokąta
- **int n**: ilość wierzchołków/krawędzi wielokąta

Polygon(vector<Point>) - konstruktor

Argumenty

- **vector<Point>P**[oints]: lista kolejnych wierzchołków wielokąta

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

double area()

Wartość zwracana

- **double result**: pole wielokąta

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

double circuit()

Wartość zwracana

- **double result**: obwód wielokąta

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

Listing 6.10: "Przykład użycia"

```
1 int main() {
2     VP P;
3     double x, y;
4     while (cin >> x >> y) P.PB(Point(x, y));
5     Polygon polygon(P);
6     printf("Pole wielokata: %.2f\n", polygon.area());
7     printf("Obwod wielokata: %.2f\n", polygon.circuit());
8 }
```

Wejście	Wyjście
0 0 0 4 3 0	Pole wielokata: 6.00 Obwod wielokata: 12.00

Rozdział 7

Inne

7.1 Find & Union

Struktura **Find & Union** pozwala na szybkie łączenie ze sobą pewnych zbiorów rozłącznych oraz na określanie czy dane dwa elementy znajdują się w tej samej grupie.

Listing 7.1: "Find & Union"

```
1  struct FU {
2      VI P, R;
3      int n;
4
5      FU(int n = 0): n(n) {
6          P.resize(n);
7          R.resize(n, 0);
8          REP(i, n) P[i] = i;
9      }
10
11     int find(int a) {
12         return P[a] == a ? a : (P[a] = find(P[a]));
13     }
14
15     bool Union(int a, int b) {
16         a = find(a);
17         b = find(b);
18         if (a == b) return false;
19         if (R[a] < R[b]) P[a] = b;
20         else P[b] = a;
21         if (R[a] == R[b]) R[a]++;
22         return true;
23     }
24
```

```
25      bool isConnected(int a, int b) {
26          return find(a) == find(b);
27      }
28  };
```

Atrybuty

- **vector<int>P**: P[i] - przedstawiciel elementu o indeksie i
- **vector<int>R**: R[i] - głębokość drzewa zakorzenionego w elemencie i
- **int n**: ilość elementów w strukturze

FU(int) - konstruktor

Argumenty

- **int n**: ilość elementów w zbiorze, $n \in [0, \sim 10^7]$

Złożoność

- czasowa: O(n)
- pamięciowa: O(n)

int find(int)

Argumenty

- **int a**: indeks elementu, dla którego chcemy wyznaczyć indeks przedstawiciela, $a \in [0, n)$

Wartość zwracana

- **int result**: indeks przedstawiciela danego elementu, result $\in [0, n)$

Złożoność

- czasowa: O($\alpha(n)$)*
- pamięciowa: O($\alpha(n)$)*

*α - odwrotność funkcji Ackermanna, funkcja bardzo wolno rosnąca

bool Union(int a, int b)

Argumenty

- **int a**: indeks pierwszego elementu, $a \in [0, n)$
- **int b**: indeks drugiego elementu, $b \in [0, n)$

Wartość zwracana

- wartość `true`, gdy podane elementy były w różnych grupach przed rozpoczęciem operacji łączenia
- wartość `false`, wpw.

Złożoność

- czasowa: $O(\alpha(n))$*
- pamięciowa: $O(\alpha(n))$*

*α - odwrotność funkcji Ackermanna, funkcja bardzo wolno rosnąca

Uwagi

- W odróżnieniu od innych metod, nazwa `Union` jest pisana z wielkiej litery, gdyż zapis `union` jest w języku C++ zarezerowanym słowem kluczowym.

bool isConnected(int, int)

Argumenty

- **int a**: indeks pierwszego elementu, $a \in [0, n)$
- **int b**: indeks drugiego elementu, $b \in [0, n)$

Wartość zwracana

- wartość `true`, gdy podane elementy należą do tej samej grupy
- wartość `false`, wpw.

Złożoność

- czasowa: $O(\alpha(n))$*
- pamięciowa: $O(\alpha(n))$*

*α - odwrotność funkcji Ackermanna, funkcja bardzo wolno rosnąca

Listing 7.2: "Przykład użycia"

```
1  int main() {
2      int n, a, b;
3      string s;
4      cin >> n;
5      FU findAndUnion(n);
6      while (cin >> s >> a >> b) {
7          if (s == "union") findAndUnion.Union(a, b);
8          if (s == "conn") printf("Elementy %d i %d %s polaczone\n",
9              a, b, findAndUnion.isConnected(a, b) ? "sa" : "nie sa");
10     }
11 }
```

Wejście	**Wyjście**
5	Elementy 0 i 1 nie sa polaczone
conn 0 1	Elementy 0 i 1 sa polaczone
union 0 1	Elementy 2 i 4 sa polaczone
conn 0 1	Elementy 0 i 4 nie sa polaczone
union 2 3	Elementy 0 i 4 sa polaczone
union 3 4	
conn 2 4	
conn 0 4	
union 1 2	
conn 0 4	

Lista zadań

Jest to lista zadań, na których testowane były implementacje programów zamieszczonych w tej pracy. Zadania te pochodzą z serwisu SPOJ, który umożliwia automatyczne sprawdzanie poprawności programów rozwiązujących różnego typu problemów algorytmicznych. Istnieje wiele tego typu serwisów, m.in. UVa Online Judge, Szkopuł, CodeChef. Są to dobre miejsca do treningu algorytmiki, gdyż zawierają zadania z wielu zawodów i konkursów.

- NWD: `https://pl.spoj.com/problems/PP0501A/`
- NWW: `https://pl.spoj.com/problems/NWW/`
- Odwrotność modularna: `https://pl.spoj.com/problems/ODWROTNO/`
- Potęgowanie modularne: `https://pl.spoj.com/problems/MWP2_2B/`
- Symbol Newtona: `https://pl.spoj.com/problems/BINOMS/`
- Sito Eratostenesa: `https://pl.spoj.com/problems/DYZIO2/`
- Pierwiastki równania kwadratowego: `https://pl.spoj.com/problems/TSQRL/`
- Sumy prefiksowe: `https://pl.spoj.com/WSDOCPP/problems/TABSWP/`
- Ekstremum na przedziale: `https://www.spoj.com/problems/MAXINARR/`
- Przeszukiwanie grafu w głąb: `https://www.spoj.com/problems/PT07Y/`
- Przeszukiwanie grafu wszerz: `https://www.spoj.com/problems/NAKANJ/`
- Algorytm Dijkstry: `https://pl.spoj.com/problems/DIJKSTRA/`
- Sortowanie topologiczne: `https://www.spoj.com/problems/TOPOSORT/`
- Minimalne drzewo rozpinające: `https://www.spoj.com/problems/MST/`
- Wyszukiwanie wzorca w tekście: `https://pl.spoj.com/problems/KMP/`
- Minimalny okres słowa: `https://www.spoj.com/problems/PERIOD/`
- Aho-Corasick: `https://pl.spoj.com/problems/FR_09_19/`

- Punkt: `https://pl.spoj.com/problems/FR_01_M2/`
- Wektor: `https://pl.spoj.com/WSDOCPP/problems/ROWWEK/`
- Prosta, półprosta, odcinek: `https://pl.spoj.com/problems/AL_09_03/`
- Okrąg: `https://pl.spoj.com/problems/ETI06F1/`
- Wielokąt: `https://www.spoj.com/problems/CALCAREA/`
- Find & Union: `https://www.spoj.com/problems/SOCNETC/`

Uwaga: Czas dostępu wyszystkich linków zamieszczonych w tym rozdziale to: 24 stycznia 2021

Bibliografia

- [WPR] "Wprowadzenie do algorytmów", Thomson H. Cormen, Charles E. Leiserson, Ronald L. Rivest, WNT, 2004
- [COMP] "Competetive Programming 3", Steven Halim, Felix Halim, 2013
- [APR] "Algorytmika praktyczna: nie tylko dla mistrzów", Piotr Stańczyk, PWN, 2009
- [ALG] "Algorytmy", Maciej Sysło, Helion, 2016
- [ASD] "Algorytmy i struktury danych", L. Banachowski, K. Diks, W. Rytter, WNT, 2003
- [UVA] Sprawdzarka UVa Online Judge (`https://onlinejudge.org/`)
- [SPOJ] Sprawdzarka SPOJ (`https://pl.spoj.com/`)
- [WIKI] Wikipedia (`https://pl.wikipedia.org`)
- [MIT] Kurs MeetIT (`http://kompendium.meetit.pl/kurs`)
- [EDU] Serwis edukacyjny I-LO w Tarnowie (`https://eduinf.waw.pl/inf/alg/001_search`)
- [WAZ] Serwis edukacyjny Ważniak (`http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych`)
- [ALG2] Serwis edukacyjny Algorytm (`http://www.algorytm.edu.pl`)
- [KHAN] Serwis edukacyjny Khan Academy `https://pl.khanacademy.org/computing/computer-science/algorithms`
- [GFG] Serwis edukacyjny GeekForGeeks (`https://www.geeksforgeeks.org`)
- [PROG] Serwis edukacyjny Programiz (`https://www.programiz.com`)
- [MAT] Serwis eduacyjny Mattomatti (`https://mattomatti.com/pl`)
- [8LO] Serwis edukacyjny VIII LO im. Marii Skłodowskiej-Curie w Katowicach. (`http://io.8lo.pl/articles.php`)

- [SAN] Serwis edukacyjny Sanfoundry (https://www.sanfoundry.com)
- [SMU] Serwis edukacyjny Smurf (http://smurf.mimuw.edu.pl)
- [GIT] Serwis GitHub (//https://github.com/)

Źródła wiedzy

- NWD
  - [WIKI] https://pl.wikipedia.org/wiki/Najwi%C4%99kszy_wsp%C3%B3lny_dzielnik
  - [APR] rozdział 4.2
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0006.php
  - [GFG] https://www.geeksforgeeks.org/c-program-find-gcd-hcf-two-numbers/ (implementacje)
- NWW
  - [WIKI] https://pl.wikipedia.org/wiki/Najmniejsza_wsp%C3%B3lna_wielokrotno%C5%9B%C4%87
  - [COMP] rozdział 5.5.2
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0008.php
  - [PROG]https://www.programiz.com/cpp-programming/examples/lcm (implementacje)
- Odwrotność modularna
  - [WIKI] https://pl.wikipedia.org/wiki/Liczba_odwrotna
  - [APR] rozdział 4.3
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0009.php
  - [GFG]https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/ (implementacje)
- Potęgowanie modularne
  - [WIKI] https://pl.wikipedia.org/wiki/Algorytm_szybkiego_pot%C4%99gowania
  - [APR] rozdział 4.5
  - [KHAN] https://pl.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation
  - [MAT] https://mattomatti.com/pl/a0037 (implementacje)
- Symbol Newtona
  - [WIKI] https://pl.wikipedia.org/wiki/Symbol_Newtona

  - [APR] rozdział 4.1
  - [WAZ] http://wazniak.mimuw.edu.pl/index.php?title=Matematyka_dyskretna_1/Wyk%C5%82ad_5:_Wsp%C3%B3%C5%82czynniki_dwumianowe
  - [MAT] https://mattomatti.com/pl/a0024 (implementacje)
- Sito Eratostenesa
  - [WIKI] https://pl.wikipedia.org/wiki/Sito_Eratostenesa
  - [ALG] rozdział 7.6.2
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0011.php
  - [MAT] https://mattomatti.com/pl/a64 (implementacje)
- Pierwiastki równania kwadratowego
  - [WIKI] https://pl.wikipedia.org/wiki/R%C3%B3wnanie_kwadratowe
  - [ALG] rozdział 3.1
  - [ALG2] http://www.algorytm.edu.pl/instrukcjawarunkowa/104-pierwiastki-rownania-kwadratowego.html
  - [MAT] https://mattomatti.com/pl/a41 (implementacje)
- Sumy prefiksowe
  - [8LO] http://io.8lo.pl/articles/SumyPrefiksowe.htm
  - [MIT] http://kompendium.meetit.pl/kurs#basic1
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0042b.php (implementacje)
- Ekstremum na przedziale
  - [8LO] http://io.8lo.pl/articles/rmq_1_1.htm
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0034.php (implementacje)
- Reprezentacja grafów
  - [WIKI] https://pl.wikipedia.org/wiki/Graf_(matematyka)
  - [WPR] rozdział 23.1
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0124.php
- Przeszukiwanie grafu w głąb
  - [WIKI] https://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b
  - [COMP] rozdział 4.2.1
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0125.php

  - [MAT] https://mattomatti.com/pl/a0119 (implementacje)

- Przeszukiwanie grafu wszerz
  - [WIKI] https://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz
  - [COMP] rozdział 4.2.2
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0126.php
  - [MAT] https://mattomatti.com/pl/a0114 (implementacje)
- Algorytm Dijkstry
  - [WIKI] https://pl.wikipedia.org/wiki/Algorytm_Dijkstry
  - [APR] rozdział 1.10
  - [MIT] http://kompendium.meetit.pl/kurs#graph2
  - [GFG] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/ (implementacje)
- Sortowanie topologiczne
  - [WIKI] https://pl.wikipedia.org/wiki/Sortowanie_topologiczne
  - [WPR] rozdział 23.4
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0137.php
  - [MAT] https://mattomatti.com/pl/fs27 (implementacje)
- Minimalne drzewo rozpinające
  - [WIKI] https://pl.wikipedia.org/wiki/Minimalne_drzewo_rozpinaj%C4%85ce
  - [ASD] rozdział 7.6
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0141.php
  - [GFG] https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/ (implementacje)
- Algorytm KMP
  - [WIKI] https://pl.wikipedia.org/wiki/Algorytm_Knutha-Morrisa-Pratta
  - [WPR] rozdział 34.4
  - [WAZ] http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Algorytmy_tekstowe_I
  - [SAN] https://www.sanfoundry.com/cpp-program-implement-kruth-morris-patt-algorithm-kmp/ (implementacje)
- Wyszukiwanie wzorca w tekście

  - [ALG2] http://www.algorytm.edu.pl/algorytmy-maturalne/wyszukiwanie-wzorca-w-tekscie.html
  - [COMP] rozdział 6.4
  - [EDU] https://eduinf.waw.pl/inf/alg/001_search/0046.php
  - [MAT] https://mattomatti.com/pl/a0011 (implementacje)
- Minimalny okres słowa
  - [APR] rozdział 6.2
  - [SMU] http://smurf.mimuw.edu.pl/node/572
- Aho-Corasick
  - [WIKI] https://pl.wikipedia.org/wiki/Algorytm_Aho-Corasick
  - [APR] rozdział 6.3
  - [MIT] http://kompendium.meetit.pl/kurs#text4
  - [GIT] https://gist.github.com/ashpriom/b8231c806edeef50afe1 (implementacje)
- Punkt
  - [WIKI] https://pl.wikipedia.org/wiki/Punkt_(geometria)
  - [COMP] rozdział 7.2.1
- Wektor
  - [WIKI] https://pl.wikipedia.org/wiki/Wektor
- Prosta, półprosta, odcinek
  - [WIKI] https://pl.wikipedia.org/wiki/Prosta
  - [COMP] rozdział 7.2.2
- Okrąg
  - [WIKI] https://pl.wikipedia.org/wiki/Okr%C4%85g
  - [COMP] rozdział 7.2.3
- Wielokąt
  - [WIKI] https://pl.wikipedia.org/wiki/Wielok%C4%85t
  - [COMP] rozdział 7.3.1
- Find & Union
  - [WIKI] https://pl.wikipedia.org/wiki/Struktura_zbior%C3%B3w_roz%C5%82%C4%85cznych
  - [COMP] rozdział 2.4.2

– [MIT] http://kompendium.meetit.pl/kurs#struct5
– [GFG] https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/ (implementacje)

Uwaga: Czas dostępu wyszystkich linków zamieszczonych w bilbiografii to: 22 stycznia 2021