informatyka+

Algorytmika i programowanie

Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

informatyka+

Kuźnia Talentów Informatycznych: Algorytmika i programowanie

Struktury danych i ich zastosowania

Marcin Andrychowicz,

Bolesław Kulbabiński, Tomasz Kulczyński,

Jakub Łącki, Błażej Osiński,

Wojciech Śmietanka

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Struktury danych i ich zastosowania

Rodzaj zajęć: Kuźnia Talentów Informatycznych
Tytuł: Struktury danych i ich zastosowania
Autor: Marcin Andrychowicz, Bolesław Kulbabiński, Tomasz Kulczyński, Jakub Łącki, Błażej Osiński, Wojciech Śmietanka

Redaktor merytoryczny: prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego **Informatyka+** — ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych w zakresie technologii informacyjno-komunikacyjnych (ICT).
www.informatykaplus.edu.pl
kontakt@informatykaplus.edu.pl

Wydawca: Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
www.wwsi.edu.pl
rektorat@wwsi.edu.pl

Projekt graficzny okładki: FRYCZ I WICHA

Warszawa 2010

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Struktury danych i ich zastosowania

Marcin Andrychowicz, Bolesław Kulbabiński, Tomasz Kulczyński, Jakub Łącki, Błażej Osiński, Wojciech Śmietanka

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Streszczenie

Celem kursu jest zapoznanie uczestnika z szeregiem różnych struktur danych. Prezentowane jest szerokie spektrum zagadnień: od podstawowych struktur wskaźnikowych jak stosy i kolejki, poprzez zbiory rozłączne, drzewa przedziałowe i wyszukiwań binarnych, aż do masek bitowych. Przydatność wymienionych struktur danych ilustrują liczne przykłady zastosowań w algorytmach optymalizacyjnych, grafowych czy też geometrycznych, a także w rozwiązaniach zadań olimpijskich. Uczestnik zapoznawany jest także pobieżnie z kontenerami z biblioteki STL, które są prostą w użyciu implementacją niektórych spośród omawianych struktur.

Zakładana jest znajomość jakiegoś języka programowania, najlepiej C++, gdyż w nim napisane są fragmenty przykładowych programów. Znajomość podstaw algorytmiki (wyniesiona choćby z kursu „Przegląd podstawowych algorytmów") będzie dla uczestnika sporą pomocą.

Spis treści

Streszczenie **4**

1 Stos **5**

2 Kolejka **6**

3 Lista **7**

4 Kopiec **9**

4.1 Zastosowanie kopca w implementacji algorytmu Dijkstry 12

5 Drzewa rozpinające **14**

6 Zbiory rozłączne **16**

7 Drzewa wyszukiwań binarnych (BST) **19**

7.1 Zrównoważone drzewa poszukiwań 22

8 Drzewa przedziałowe **24**

8.1 Drzewo potęgowe 25

8.2 Drzewa przedziałowe 27

9 Technika zamiatania **32**

9.1 Zamiatanie kątowe 34

9.2 Sortowanie kątowe 35

10 Drzewa TRIE **36**

11 Algorytm Aho-Corasick **39**

11.1 Algorytm Bakera 40

12 Maski bitowe **41**

12.1 Programowanie dynamiczne na maskach 43

12.2 Meet in the middle 44

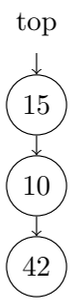
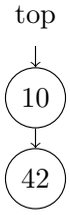
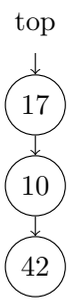
Literatura **45**

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

1 Stos

Definicja 1. *Stos* to struktura danych, w której dokładamy nowe elementy na szczycie stosu i zdejmujemy elementy począwszy od szczytu stosu.

Będziemy chcieli, żeby stos w pamięci wyglądał jak następuje:

stos	stos po zdjęciu elementu ze szczytu	stos po dodaniu nowego elementu
		

Przyjrzyjmy się teraz implementacji stosu w języku C++ za pomocą operatorów `new` i `delete`.

```
struct stack_element {                    // klasa element stosu
        int val;                          // wartość w bieżącym elemencie
        stack_element *prev;              // wskaźnik na poprzedni element
        stack_element(int _val, stack_element *_prev) { // konstruktor
            val = _val;                   // ustawiamy wartość
            prev = _prev;                 // ustawiamy wskaźnik na poprzedni

        }
} ;

struct my_stack {
        stack_element *_top;
        int _size;
        my_stack() {                        // konstruktor
            _top = NULL;                    // ustawiamy szczyt stosu na NULL
            _size = 0;                      // rozmiar na 0
        }
        void push(int a) {                      // dodawanie elementu do stosu
            _top = new stack_element(a, _top); // tworzymy nowy element, którego
                                               // poprzednikiem będzie _top stosu
            ++_size;                           // zwiększamy rozmiar
        }
        void pop() {                    // usuwanie elementu ze stosu
            stack_element *tmp = _top; // w zmiennej pomocniczej pamiętamy szczyt
            _top = tmp->prev;          // obniżamy szczyt
            delete tmp;                // usuwamy stary szczyt
            --_size;                   // zmniejszamy rozmiar
        }
} ;
```

Ćwiczenie 1. Zaimplementuj dodatkowe metody:

- `front` — zwracającą pierwszy element ze stosu
- `size` — zwracającą liczbę elementów na stosie
- `empty` — zwracającą wartość logiczną, czy stos jest pusty

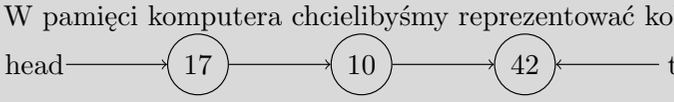
Ćwiczenie 2. Uzasadnij, dlaczego stos jest odpowiedni do implementacji przeszukiwania metodą DFS, ale nie nadaje się do implementacji przeszukiwania metodą BFS.

Ćwiczenie 3. Przetestuj na komputerze działanie procedury `pop`, gdy wywołuje się ją na pustym stosie.

Ćwiczenie 4. Za pomocą kontenera `stack` z biblioteki STL zaimplementuj przeszukiwanie grafu metodą DFS.

2 Kolejka

Definicja 2. *Kolejka* to struktura danych, w której dokładamy elementy na koniec i pobieramy elementy z przodu kolejki. W realnym świecie występuje jako kolejka w sklepie: najpierw obsługiwany jest klient z przodu kolejki, zaś nowi klienci ustawiają się na samym końcu kolejki.
W pamięci komputera chcielibyśmy reprezentować kolejkę w taki sposób:



Przyjrzyjmy się teraz implementacji kolejki w języku C++ za pomocą operatorów `new` i `delete`.

```
struct queue_element {                                  // element kolejki
        int val;
        queue_element *next;
        queue_element(int _val, queue_element *_next) { // konstruktor
            val = _val;
            next = _next;
        }
} ;

struct my_queue {
    queue_element *head, *tail;                         // głowa i ogon kolejki
    int _size;
    my_queue() {
        head = tail = NULL;
        _size = 0;
    }
    void push(int a) {                          // dodawanie elementu na koniec
        ++_size;                                // zwiększamy rozmiar kolejki
        if (head == NULL) {                     // jeśli kolejka jest pusta
            head = new queue_element(a, NULL); // ustawienie wartości głowy
            tail = head;         // ustawienie wartości ogona
        }
        else {                                      // jeśli kolejka ma
                                                    // już jakieś elementy
```

```
            tail->next = new queue_element(a, NULL); // następnik ogona,
                                                     // to nowy element
            tail = tail->next;                       // przechodzimy
                                                     // do ostatniego elementu
        }
    }
} ;
```

Idea działanie tego kodu jest prosta: cały czas trzymamy wskaźniki na głowę i ogon kolejki. Każdy element z kolejki ma wskaźnika na swojego następnika. Wyjmujemy elementy wskazywane przez głowę. Dodajemy elementy na koniec, zaraz za ogonem.

Ćwiczenie 5. Uzupełnij strukturę kolejki o następujące metody:

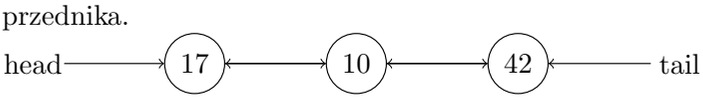
- `front` — zwracającą pierwszy element z kolejki
- `size` — zwracającą liczbę elementów w kolejce
- `empty` — zwracającą wartość logiczną, czy kolejka jest pusta
- `pop` — zdejmująca pierwszy element z przodu kolejki, warto tutaj poszukać podobieństw z `pop` ze stosu

Ćwiczenie 6. Uzasadnij, dlaczego kolejka jest odpowiednią strukturą do przeszukiwania metodą BFS, a nie nadaje się do przeszukiwania metodą DFS.

Ćwiczenie 7. Zaimplementuj przeszukiwanie grafu metodą BFS korzystając z kontenera `queue` z biblioteki STL.

3 Lista

Lista dwukierunkowa to struktura danych, w której każdy element ma swojego następnika i poprzednika.



Taka lista umożliwia wstawianie i usuwanie elementów z dowolnego miejsca. Spróbujmy napisać szkielet struktury danych lista. Zacznijmy od pojedynczego elementu listy:

```
struct list_element {    // element listy
    int val;
    list_element *prev, *next;
    list_element(int _val, list_element *_prev, list_element *_next) {
                      // konstruktor, ustawia wartości początkowe
        val = _val;   // wartość elementu
        prev = _prev; // poprzednik
        next = _next; // następnik
    }
} ;
```

Jak widać, potrzebujemy wskaźniki na poprzedni i następny element listy. Poza tym mamy pole `val`, w którym jest przechowywana wartość elementu. Pojedynczy element struktury `my_list` ma dwa wskaźniki, `head` i `tail` wskazujące na pierwszy i ostatni element listy. Pośrednie elementy są połączone ze sobą.

```
struct my_list {
    list_element *head, *tail;        // wskaźniki na głowę i ogon listy
```

```
    int _size;                          // rozmiar
    my_list() {                         // konstruktor
        head = tail = NULL;             // początkowo głowa = ogon = nic
        _size = 0;
    }
    void push_back(int a) {             // dodanie elementu na koniec
        ++_size;
        if (head == NULL) {             // jeśli kolejka jest pusta
            head = tail = new list_element(a, NULL, NULL);
        }
        else {                          // kolejka ma już jakieś elementy
            tail->next = new list_element(a, tail, NULL);
            tail->next->prev = tail;
            tail = tail->next;
        }
    }
    void pop_back() {                   // usunięcie elementu z końca kolejki
        --_size;
        list_element *tmp = tail;       // zapamiętujemy stary ogon
                                        // w zmiennej tymczasowej
        tail = tail->prev;              // poprawienie ogona
        tail->next = NULL;              // poprawienie następnika ogona
        delete tmp;                     // usunięcie starego ogona
    }
    list_element* search(int x) {       // znalezienie pierwszego elementu
                                        //  o podanym kluczu
        list_element* e = head;
        while (e != NULL) {             // idziemy po liście aż znajdziemy
            if (e->val == x) return e;  // Hurra! Znaleźliśmy!
            e = e->next;                // przechodzimy dalej
        }
        return NULL;                    // nie znaleźliśmy
    }
    void remove(list_element *ptr) {              // usuwanie elementu
        --_size;
        if (ptr->prev == NULL) head = ptr->next;  // jeśli nie ma poprzednika
        else ptr->prev->next = ptr->next;         // jeśli jest
        if (ptr->next == NULL) tail = ptr->prev;  // jeśli nie ma następnika
        else ptr->next->prev = ptr->prev;         // jeśli on jest
        delete ptr;                               // usunięcie wskaźnika
    }
    void write() {                                // wypisanie elementów listy
        list_element *e = head;
        while (e != NULL) {
            cout<<e->val<<" ";
            e = e->next;
        }
        cout<<endl;
    }
} ;
```

Obserwując ten kod należy zwrócić uwagę, że nie wszystkie metody wykonywane są w czasie stałym. Metoda `search` działa w czasie liniowym ze względu na rozmiar listy.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Ćwiczenie 8. Jako ćwiczenie warto zaimplementować dodatkowe metody w strukturze `my_list`. Np.:

- `pop_front` — usunięcie elementu z przodu listy
- `push_front` — dodanie elementu z przodu listy
- `insert` — dodanie elementu do listy za wskazanym elementem

Ćwiczenie 9. Spróbuj przerobić strukturę `my_queue`, żeby działała jak lista jednokierunkowa. W tym celu należy dodać funkcjonalności: wyszukiwanie elementów, usuwanie elementów ze środka, wstawianie elementów w środek listy.

Ćwiczenie 10. Korzystając z listy dwukierunkowej zaimplementuj strukturę pozwalającą na edycję tekstu. W edycji tekstu dozwolone są następujące operacje:

- napisanie jednej litery zaraz za kursorem
- usunięcie jednej litery z miejsca bezpośrednio za kursorem
- przesunięcie kursora o jedno pole w lewo/prawo
- wypisanie całego tekstu na ekran

Podsumowanie struktur

Warto uważnie prześledzić działanie tych trzech struktur danych. Szczęśliwie są one wszystkie zaimplementowane w bibliotece standardowej C++ STL. Są to odpowiednio kontenery `stack`, `queue`, `list`. Autorzy tego dokumentu starali się zachować oryginalne nazwy metod tak, aby przesiadka na kontenery z biblioteki STL była możliwie bezbolesna. Drobne różnice są jedynie w implementacji listy.

4 Kopiec

Wstęp

Kolejny fragment jest poświęcony strukturze danych, zwanej **kopcem**. W poniższej tabeli porównano złożoności operacji na kopcu oraz tablicy (nie-)posortowanej.

	wstawienie	usunięcie	minimum
tablica	$O(1)$	$O(1)$	$O(n)$
posortowana tablica	$O(n)$	$O(n)$	$O(1)$
kopiec	$O(\log n)$	$O(\log n)$	$O(\log n)$

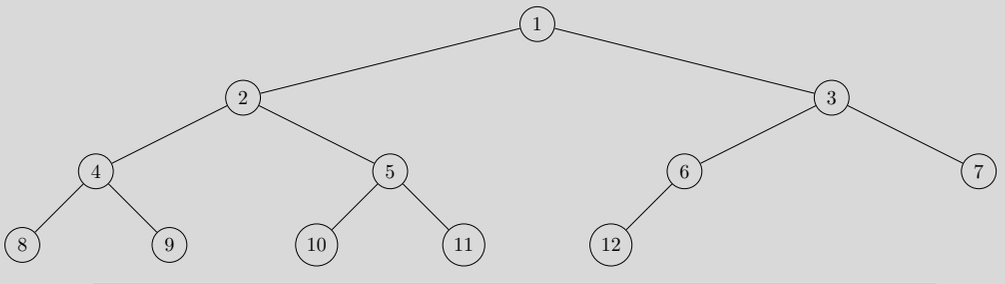
Oznaczenia:

- **wstawienie** — wstawienie elementu
- **usunięcie** — usunięcie elementu na podanej pozycji (a nie o danej wartości)
- **minimum** — znalezienie minimum

Implementacja pełnego drzewa binarnego

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Definicja 3. Pełne drzewo binarne to drzewo, którego wierzchołki mają co najwyżej dwóch synów (binarne) a liście[a] znajdują się tylko na 2 poziomach, przy czym te na niższym poziomie są „z jednej strony" (patrz rys. 7.1).



Rys. 1: Numeracja wierzchołków pełnego drzewa binarnego

[a] **liść** — wierzchołek nie posiadający synów

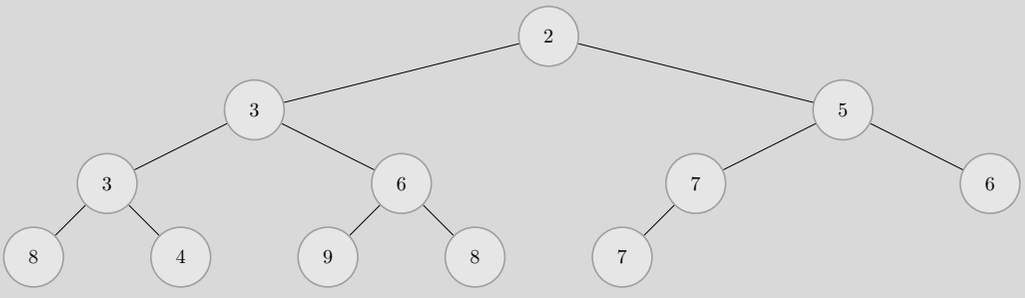
Wierzchołki pełnego drzewa binarnego możemy ponumerować tak, jak to zostało przedstawione na rys. 7.1. Taka numeracja jest wygodna, gdyż:

- lewy syn x to $2x$
- prawy syn x to $2x + 1$
- ojciec x to $\lfloor x/2 \rfloor$

W wierzchołkach pełnego drzewa binarnego będziemy przechowywali pewne wartości (np. liczby). Drzewo binarne będziemy reprezentowali w postaci tablicy `heap`, przy czym `heap[i]` oznacza wartość w i-tym wierzchołku.

Własność kopca

Definicja 4. Kopcem nazywamy pełne drzewo binarne z wartościami w wierzchołkach, które ma własność kopca, tzn. każdy wierzchołek ma przypisaną wartość nie większą niż wartości jego synów.
Innymi słowy, dla wszystkich x zachodzi $heap[\lfloor x/2 \rfloor] \leqslant heap[x]$.



Rys.2: Przykładowy kopiec

Operacje na kopcu

Niektóre operacje na kopcu oprócz opisu będą zawierały implementację. Na potrzeby implementacji zakładamy, że nasz kopiec ma następującą deklarację:

```
#define MAXN 1000000

int heap[MAXN+1];
```

```
int size=0; //liczba elementów w kopcu
//elementy w kopcu to heap[1],heap[2],...,heap[size]
```

Wstawianie elementu Aby wstawić element do kopca:

1. dodajemy nowy element na koniec tablicy (własność kopca może być zaburzona)
2. dopóki ojciec nowego elementu jest od niego większy, to zamieniamy wartości w obu wierzchołkach (proces ten nazywamy **kopcowaniem w górę (HeapUp)**)

```
void heapUp(int x){ //kopcuje w górę element na pozycji x
    while(heap[x/2] > heap[x]){
        swap(heap[x], heap[x/2]); //zamienia wartości w obu węzłach
        x/=2;
    }
}

void insert(int x){ //dodaje x do kopca
    heap[++size]=x;
    heapUp(size);
}
```

Znajdowanie minimum Minimum znajduje się oczywiście z korzeniu, czyli w `heap[1]`.

```
int minimum(){ //zwraca minimum
    return heap[1];
}
```

Usuwanie korzenia Aby usunąć korzeń z kopca:

1. w miejsce korzenia wstawiamy ostatni element z tablicy (własność kopca może być zaburzona)
2. dopóki wstawiony element jest większy od któregoś ze swoich synów to zamieniamy go z synem o mniejszej wartości (proces ten nazywamy **kopcowaniem w dół (HeapDown)**)

Usuwanie korzenia dobrze wizualizuje prezentacja dołączona do tego tematu.

```
void heapDown(int x){ //kopcuje w dół element na pozycji x
    while(2*x <= size){ //dopóki x ma chociaż jednego syna
        int son=2*x;
        if(2*x+1 <= size && heap[2*x+1] < heap[2*x])
            son=2*x+1;
        //zmienna son zawiera teraz indeks syna x o mniejszej wartości
        if(heap[son] >= heap[x]) break;
        swap(heap[x], heap[son]); //zamienia wartości w obu węzłach
        x=son;
    }
}

void eraseRoot(){ //usuwa korzeń (czyli minimum)
    heap[1]=heap[size--];
    heapDown(1);
}
```

Usuwanie dowolnego elementu Aby usunąć dowolny element (niekoniecznie korzeń) z kopca:

1. w miejsce usuwanego elementu wstawiamy ostatni element
2. kopcujemy nowy element w górę
3. kopcujemy nowy element w dół

Analiza złożoności Wszystkie opisane operacje działają w czasie proporcjonalnym do wysokości kopca, która jest logarytmiczna względem liczby elementów w kopcu.

Sortowanie przez kopcowanie — HeapSort

Kopca można użyć do szybkiego — $O(n \log n)$ — sortowania. Pomysł jest dość prosty — elementy tablicy, którą chcemy posortować wrzucamy do kopca a następnie dopóki kopiec nie jest pusty, to wypisujemy minimum i je usuwamy.

4.1 Zastosowanie kopca w implementacji algorytmu Dijkstry

Przypomnienie algorytmu

Przypomnijmy, że algorytm Dijkstry służył do znajdowania najkrótszych ścieżek ze źródła w sieciach, w których wagi krawędzi są nieujemne.[1] Oto schemat tego algorytmu:

1. oznacz wszystkie wierzchołki jako nieodwiedzone
2. dla każdego $v \in V$ przyjmij $dis[v] = \infty$
3. przyjmij $dis[s] = 0$
4. dopóki istnieje nieodwiedzony wierzchołek o skończonej odległości:
   (a) niech v będzie wierzchołkiem nieodwiedzonym o najmniejszej odległości
   (b) oznacz v jako odwiedzony
   (c) zrelaksuj wszystkie krawędzie wychodzące z v

Kluczowa jest tutaj operacja 4.a, którą wykonujemy $O(|V|)$ razy a dotychczas zabierała ona czas $O(|V|)$, co dawało całkowitą złożoność algorytmu $O(|V|^2)$

Zastosowanie kopca

Będziemy przechowywać w kopcu numery wierzchołków nieodwiedzonych. Chcielibyśmy znajdować szybko w kopcu wierzchołek o najmniejszej wartości w tablicy dis. Przy kopcowaniu musimy więc porównywać odpowiednie wartości w tablicy dis, a nie bezpośrednio wartości trzymane w kopcu ($dis[x]$ będziemy nazywać priorytetem wartości x).

Priorytety mogą jednak ulegać zmianie w trakcie działania algorytmu (na skutek relaksacji krawędzi), co może zaburzyć własność kopca. Zauważmy jednak, że mogą one tylko maleć.

[1] Dokładniejszy opis znajduje się w notatkach do kursu „Przegląd podstawowych algorytmów"

Rozwiązanie problemu — I metoda

Za każdym razem, gdy priorytet jakiegoś elementu się zmniejszy, musimy wykonać kopcowanie tego elementu w górę. Umiemy jednak wykonywać kopcowanie jedynie elementu na danej pozycji (a nie o danej wartości). Problem ten da się rozwiązać rozbudowując nieznacznie strukturę naszego kopca.

Oprócz tablicy `heap`, będziemy trzymali tablicę `where`, gdzie `where[x]` oznacza pozycję w kopcu wartości x, tzn. $heap[where[x]] = x$. Przyjmujemy $where[x] = 0$, jeśli w kopcu nie ma elementu x. Tablicę tę musimy uaktualniać za każdym razem, gdy przemieszczamy elementy w kopcu. Rozwiązanie to nie jest perfekcyjne. Musimy zaimplementować kopiec z operacją zmiany priorytetu, co może zająć dość dużo czasu, nie są bowiem dostępne żadne gotowe implementacje kopca dysponujące taką operacją.

Rozwiązanie problemu — II metoda

Przedstawimy teraz rozwiązanie, które korzysta ze zwykłego kopca (bez operacji zmiany priorytetu). Do kopca będziemy wrzucać pary postaci $(dis[x], x)$ — a nie pojedyncze liczby jak dotychczas, przy czym przyjmujemy porządek leksykograficzny na parach, tzn.

$$(x, y) < (a, b) \iff x < a \vee (x = a \wedge y < b)$$

Najmniejszy element w kopcu, odpowiada wówczas wierzchołkowi o najmniejszej odległości.

Co zrobić, jeśli wartość $dis[x]$ się zmieni? Wrzucamy wówczas do kopca nową parę $(dis[x], x)$. Przy takim podejściu w kopcu będą znajdowały się śmieci — nieaktualne pary postaci (d, x), gdzie $d > dis[x]$. Pary takie po prostu pomijamy przy wyciąganiu ich z kopca.

A oto dokładniejszy schemat takiego algorytmu:

1. dla każdego $v \in V$ ustaw $dis[v] = \infty$
2. ustaw $dis[s] = 0$
3. wrzuć do kopca parę $(0, s)$
4. dopóki kopiec nie jest pusty
   (a) wyciągnij najmniejszą parę z kopca, oznaczmy ją przez (d, v)
   (b) jeśli $d > dis[v]$, to powróć do pkt. 4
   (c) zrelaksuj wszystkie krawędzie wychodzące z v, jeśli poprawia to odległość do wierzchołka x, to dodaj parę $(dis[x], x)$ do kopca

Złożoność i implementacja

Oba przedstawione rozwiązania mają złożoność $O(|E| \log |V|)$, czyli dużo lepszą niż algorytm Dijkstry nie korzystający z kopca, o ile tylko ilość krawędzi w grafie jest dużo mniejsza od $|V|^2$ (grafy takie nazywamy rzadkimi). W zadaniach olimpijskich często mamy do czynienia z takimi grafami a nawet z grafami dla których maksymalne liczby krawędzi i wierzchołków są tego samego rzędu.

Implementację algorytmu Dijkstry z użyciem kopca pozostawiam jako wartościowe ćwiczenia. Być może zastanawiasz się, jak reprezentować w programie pary. Najwygodniejszą opcją jest skorzystanie z klasy `pair`. Poniższy kod prezentuje sposób jej używania:

```
#include <stdio.h>
//poniższe dwa wiersze są konieczne do korzystania z klasy pair
#include <iostream>
using namespace std;

int main(){
```

```
    pair<int, int> para; //tworzy zmienną para, której obie składowe są typu int
    para.first=7; //ustawia pierwszą składową na 7
    para.second=5; //ustawia drugą składową na 5
    printf("%d %d\n", para.first, para.second); //wypisze "7 5"
    para=make_pair(2, 3); //szybsza opcja na przypisanie obu wartości
    pair<int, int> cos(2, 3); //przypisanie wartości przy tworzeniu
    if(para < cos){ //pary są porównywane w porządku leksykograficznym!
        cos = para;
    }
    return 0;
}
```

5 Drzewa rozpinające

Rozważmy następujący problem. Jesteśmy na obozie informatycznym i mamy za zadanie utworzyć sieć przewodową pomiędzy n hubami. Dodatkowo chcemy zużyć do tego możliwie mało kabla, który w końcu też kosztuje. Nie wszystkie pary huby dają się połączyć, gdyż np. są za daleko od siebie. Pomiędzy parami hubów, które możemy połączyć, znamy wymagane długości kabla. Nie można tworzyć nowych rozgałęzień przy tworzeniu hubów. W jaki sposób znaleźć sieć łączącą huby tak, aby możliwy był transport danych między wszystkimi parami (być może z użyciem hubów pośredniczących)? Proponujemy zastanowić się chwilę nad tym problemem.

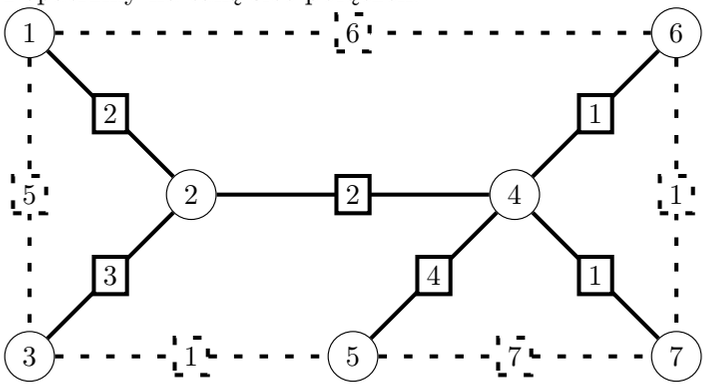
Rozwiązanie

Zacznijmy od obserwacji, że szukana sieć jest drzewem. Gdyby w optymalnej sieci były cykle, to można by wyrzucić dowolną krawędź z cyklu otrzymując sieć, która nadal łączy wszystkie huby, ale jest krótsza o jedno połączenie (zużywamy mniej kabla).

Kolejna obserwacja jest taka: zawsze istnieje optymalne drzewo rozpinające (czyli takie, które łączy wszystkie wierzchołki do jednej spójnej składowej), które zawiera najkrótszą krawędź. Powód jest prosty: gdybyśmy znaleźli drzewo o najmniejszej sumarycznej długości, które nie zawierałoby tej najkrótszej krawędzi, moglibyśmy dodać tę krawędź do tego drzewa i uzyskalibyśmy jakiś cykl. Jak wiemy, możemy wyrzucić z tego cyklu dowolną krawędź (inną od tej dopiero co dodanej) i nadal mieć spójną sieć. Ponieważ pewną krawędź zastąpiliśmy inną, o nie większej długości, to w sumie nowa długość kabli jest nie większa niż poprzednia.

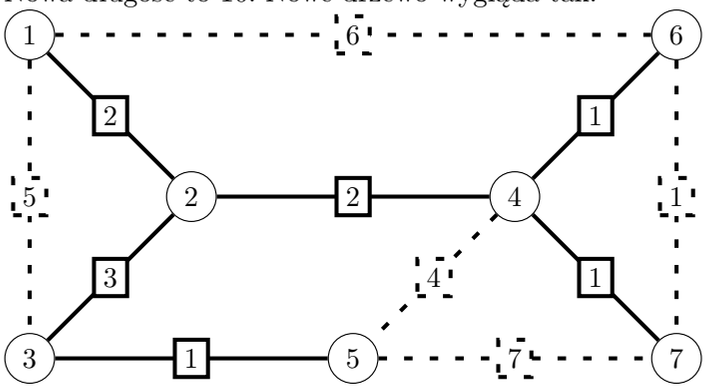
Przykład

Popatrzmy na taką sieć połączeń:



Ktoś wybrał pogrubione krawędzie, uzyskując według siebie najkrótsze drzewo rozpinające. Łączna długość: 13. Spróbuj samemu znaleźć błąd w wyborze krawędzi!

Rozwiązanie: Jest cykl 2—3—5—4. W tym cyklu nie została wybrana tylko krawędź 3—5, mimo, że ta krawędź jest najkrótsza. Podmieniamy ją za najdłuższą krawędź z tego, czyli 4—5. Nowa długość to 10. Nowe drzewo wygląda tak:



Ponieważ nas interesuje dowolne drzewo rozpinające, którego długość jest minimalna, to możemy z góry wziąć najkrótszą krawędź i mieć pewność, że nie zamkniemy sobie drogi do optymalnego rozwiązania.

Pomysł 1

Teraz w dość naturalny sposób nasuwa się pomysł, aby powtórzyć sposób wyboru kolejnej krawędzi. Wybierając nową krawędź omijamy te, których dodanie spowodowałoby powstanie cyklu. Zastanówmy się, dlaczego to rozwiązanie działa.

Krok algorytmu

Popatrzmy trochę inaczej na ten problem. Znajdujemy najpierw najkrótszą krawędź spośród wszystkich dozwolonych. Dodajemy ją do wynikowego drzewa. Przypuśćmy, że łączyła ona wierzchołki v i u. Możemy złączyć te wierzchołki do jednego wierzchołka. W końcu, od momentu dodania krawędzi $v-u$ nie rozróżniamy już tych wierzchołków. Jest nam obojętne, czy jakaś nowa krawędź połączy w z v, czy w z u. Z połączonego wierzchołka uv wychodzą wszystkie te same krawędzie, które wychodziły wcześniej. Po połączeniu może się pojawić jakiś wierzchołek, do którego idą dwie krawędzie z wierzchołka uv. Oczywiste jest, że tę o większej długości usuwamy. Nie będzie już ona potrzebna.

Krok i co dalej?

Po jednokrotnym wykonaniu operacji wyboru krawędzi i scaleniu dwóch wierzchołków dostajemy nowy problem. Uzyskaliśmy graf o $n-1$ wierzchołkach, które trzeba połączyć drzewem rozpinającym. Postępujemy znów tak samo. Czyli wykonujemy **krok algorytmu**. Znajdujemy najkrótszą krawędź i łączymy. Robimy tak, aż nie osiągniemy grafu z jednym wierzchołkiem. Stan z jednym wierzchołkiem oznacza, że wszystkie wierzchołki są w jednej spójnej składowej.

Poprawność

Poprawność tego algorytmu wynika stąd, że przed każdym krokiem mamy problem znalezienia drzewa rozpinającego w zwyczajnym grafie, bez żadnych wybranych wcześniej krawędzi. Wiemy natomiast z wcześniejszych rozważań, że w takim grafie zawsze istnieje rozwiązanie, w którym bierzemy najkrótszą krawędź. Wynika stąd, że algorytm, który za każdym razem wybiera najkrótszą krawędź i scala końce tej krawędzi, poprawnie znajdzie drzewo rozpinające o minimalnej sumarycznej długości.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

A jednak Pomysł 1 działa!

Zauważmy teraz, że początkowy algorytm robi dokładnie to samo. Zawsze bierzemy najkrótszą niewykorzystaną jeszcze krawędź i dodajemy ją do szukanego drzewa, jeśli jej końce w drzewie leżą w różnych składowych. Wniosek? Ten algorytm także jest poprawny.

Ile to nas kosztuje?

Pozostaje kwestia, jak szybko można zaimplementować ten algorytm. Niech n oznacza ilość wierzchołków grafu, zaś m oznacza liczbę krawędzi. Na początku warto posortować krawędzie względem rosnącej długości, aby potem wybór kolejnych najkrótszych był prosty. Ta faza kosztuje nas $O(m \log m)$. Trudniejsza jest kwestia, jak szybko sprawdzać, czy dwa końce krawędzi leżą w budowanym drzewie rozpinającym w jednej spójnej składowej. Najprostsze rozwiązanie, to sprawdzanie przy każdej dodawanej krawędzi algorytmem przeszukiwania grafu, np. metodą DFS, czy końce krawędzi są w tej samej spójnej składowej. Takie rozwiązanie prowadzi do złożoności tej fazy wynoszącej $O(n \cdot m)$. Jest to bardzo dużo i spróbujemy zredukować ten koszt. Można to nieco ulepszyć spostrzeżeniem, że jeśli nie dodaliśmy nowej krawędzi, to podział na składowe jest cały czas ten sam. Oznacza to, że wystarczy, żebyśmy po każdym dodaniu krawędzi policzyli podział na składowe. Możemy to zrobić przypisując wszystkim wierzchołkom w danej składowej jakąś liczbę charakterystyczną. Później łatwo już odpowiadać, czy dwa wierzchołki są w jednej spójnej składowej. Wystarczy spojrzenie, czy przyporządkowane im liczby charakterystyczne są sobie równe. Takie rozwiązanie w tej fazie kosztuje $O(n^2)$ czasu, co jest zauważalnie lepsze niż rozwiązanie poprzednie, szczególnie, gdy graf jest gęsty.

Znany jest jednak znacznie szybszy sposób sprawdzania czy wierzchołki są połączone. Metoda ta korzysta ze struktury zbiorów rozłącznych.

6 Zbiory rozłączne

Dana jest pewna rodzina zbiorów. Każde dwa zbiory w tej rodzinie są parami rozłączne. Na tej rodzinie zbiorów chcemy wykonywać dwie operacje. Jedna - to zapytanie: czy element a i element b należą do tego samego zbioru. Druga - to operacja połączenia dwóch podzbiorów w jeden.

Interesuje nas przede wszystkim to, jak szybko działa nasza struktura, jeśli na n elementach wykonamy n operacji złączenia zbiorów i m operacji sprawdzenia, czy jakieś dwa elementy są w tym samym zbiorze.

We wszystkich prezentowanych poniżej podejściach będziemy wykonywać dwie operacje:

- `FIND(x)` — znalezienie reprezentanta zbioru, do którego należy `x`. Jeśli x i y należą do tego samego zbioru, to `FIND(x)` zwraca to samo, co `FIND(y)`.
- `UNION(x,y)` — złączenie zbioru zawierającego x ze zbiorem zawierającym y.

Podejście pierwsze

Każdy zbiór reprezentujemy w postaci listy elementów. Dodatkowo każdy element ma wskaźnik do pierwszego elementu na liście. Wtedy:

- Operację `FIND` wykonujemy w czasie $O(1)$, wystarczy odwołać się do pierwszego elementu.
- Jeśli chcemy złączyć dwie listy, to możemy jedną zostawić taką jaką jest, a drugą dołączyć na koniec. Na razie kosztowało nas to czas $O(1)$. Niestety musimy jeszcze poprawić wartości wskaźników drugiej listy tak, aby pokazywały na pierwszy element pierwszej listy. To kosztuje znacznie więcej, bo $O(d)$, gdzie d jest długością drugiej listy. Rozważmy najgorszy scenariusz: mamy jeden duży zbiór, który za każdym razem dołączamy do listy jednoelementowej na

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

koniec. Wtedy i - ta taka operacja kosztuje i modyfikacji wskaźnika. Oznacza to, że wykonanie n takich operacji wykonamy będzie trwało:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

czyli $O(n^2)$.

Wszystkie zapytania i operacje połączenia zajmują razem $O(n^2 + m)$.

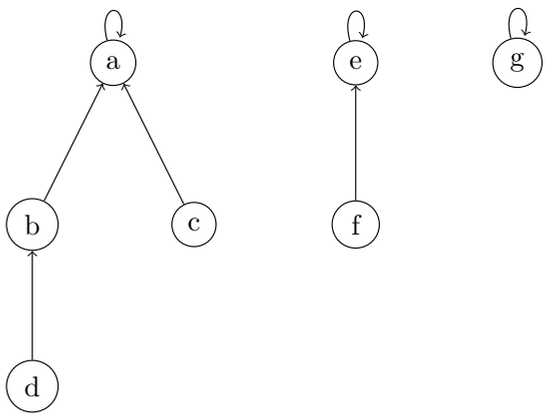
Podejście drugie

Zamiast doklejać drugą listę na koniec pierwszej, można wybrać kolejność doklejania tak, aby było potem możliwie mało operacji modyfikowania wskaźnika na pierwszy element. W tym celu zawsze do dłuższej listy doklejamy krótszą. Powoduje to, że każdy element, gdy zmieniamy mu wskaźnik do reprezentanta, przynajmniej podwaja długość listy, w której się znajduje. Wniosek jest taki, że każdy element będziemy poprawiać maksymalnie $O(\log n)$ razy. Oznacza to, że wszystkie operacje połączenia kosztują nas $n \cdot O(\log n) = O(n \log n)$ plus $O(n)$ operacji łączenia samych list. Razem daje to $O(n \log n)$. Zapytania `FIND` kosztują nas razem $O(m)$.

W sumie działanie naszej struktury będzie kosztować $O(m + n \log n)$. Niby niewielka zmiana, a złożoność znacznie lepsza.

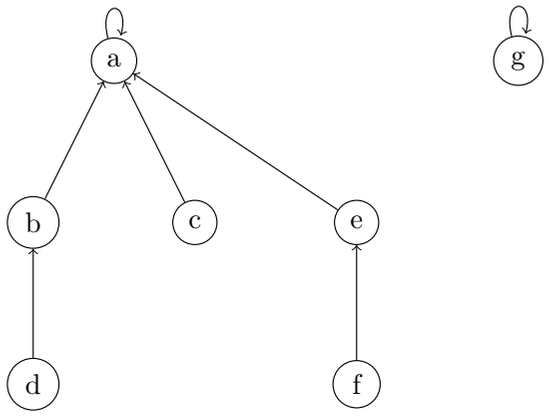
Do trzech razy sztuka

Ostanie podejście będzie podobne. Tym razem reprezentujemy zbiór jako drzewo. W takim drzewie każdy element będzie miał wskaźnik na swojego ojca. Dodatkowo każdy będzie pamiętał wysokość, na której się aktualnie znajduje. Zasadniczo będzie to wyglądać tak:



Jeśli wywołujemy `FIND`, to idziemy po wskaźnikach do góry. Jeśli chcemy połączyć zbiór zawierający x ze zbiorem zawierającym y, to musimy najpierw znaleźć reprezentantów, czyli r_x =`FIND(x)`, r_y =`FIND(y)`, a potem do elementu o wyższej randze spośród r_x i r_y podpinamy ten o niższej randze. Jeśli przypadkiem rangi były równe, to zwiększamy rangę temu elementowi, do którego został podpięty ten drugi. Efekt przykładowej operacji `UNION(b, f)`:

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego



Ćwiczenie 11. Wykaż, że wysokość tak skonstruowanego drzewa jest co najwyżej logarytmiczna ze względu na liczbę elementów tego poddrzewa.

Ćwiczenie 12. Napisz procedurę `UNION` i funkcję `FIND`. Przyjmij, że elementy mają numery od 1 do n, tablica `father` reprezentuje ojca w drzewie reprezentującym zbiór. Tablica `rank` reprezentuje rangę/wysokość zbioru podczepionego w danym elemencie.

Na razie złożoność czasowa jest bardzo podobna do tej z podejścia drugiego. Może trochę lepsza stała, ale nic poza tym. Teraz posłużymy się kolejną sztuczką, aby znacząco przyśpieszyć algorytm.

Sztuczka

Od tej pory nazywajmy pamiętaną wysokość drzewa rangą. Na razie napiszmy funkcję FIND następująco:

```
int FIND(int x) {
   if (father[x] == x) return x;
   return FIND(father[x]);
}
```

Ten kod jest poprawny, ale nieoptymalny. Przy każdym zapytaniu o element x musimy pokonać długą ścieżkę do korzenia. Z pomocą przychodzi nam sztuczka: każdy odwiedzany element podpinamy do reprezentanta. Zatem jeśli przechodzimy ścieżkę, to podepnijmy wszystkie odwiedzane elementy do reprezentanta. Popatrzmy na nowy kod:

```
int FIND(int x) {
   if (father[x] == x) return x;
   father[x] = FIND(father[x]);
   return father[x];
}
```

Oszacowanie złożoności czasowej tego algorytmu jest dość trudne. Zostało to opisane w wyczerpujący sposób we „Wprowadzeniu do algorytmów". Warto zacytować oszacowanie złożoności za tą książką: n operacji łączenia i m operacji FIND kosztują $O(m \log^* n)$, gdzie $\log^* n$ oznacza wysokość stosu potęg dwójek potrzebnych do zbudowania n. Np. $\log^* 2 = 1$, $\log^* 2^2 = 2$, $\log^* 2^{2^2} = 3$, $\log^* 2^{2^{2^2}} = 4$, itd. Wartość $2^{2^{2^{2^2}}} = 2^{65536}$ wielokrotnie przewyższa rozmiarem dane, które są przetwarzane przez dzisiejsze komputery. Można więc przyjąć, że czas działania tego algorytmu jest liniowy.

Cała ta sztuczka nosi nazwę **kompresji ścieżek**.

Podsumowanie

Korzystając z przedstawionego algorytmu na znajdowanie drzewa rozpinającego i szybkiej struktury danych rozwiązującej problem FIND-UNION możemy zbudować drzewo rozpinające o najmniejszej sumarycznej długości w czasie $O(m \log m)$.

Cały przedstawiony algorytm nosi nazwę algorytmu Kruskala z użyciem struktury danych dla zbiorów rozłącznych.

7 Drzewa wyszukiwań binarnych (BST)

Przy rozwiązywaniu problemów algorytmicznych natrafiamy często na konieczność przechowywania zbioru (czy raczej multizbioru), którego zawartość będzie się często zmieniała. Zwykle chcemy wtedy móc stwierdzać, czy element o określonej wartości w nim występuje. Jaka jest odpowiednia struktura danych do przechowywania takiego multizbioru?

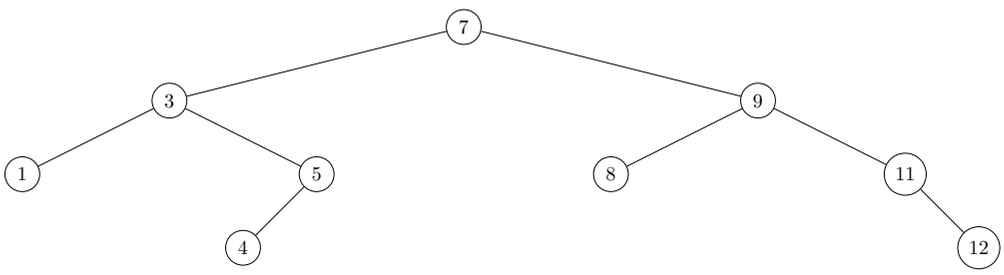
Pierwsze, co przychodzi do głowy, to zwykła lista. Wówczas jednak szukanie określonej wartości, może wymagać przejrzenia wszystkich elementów multizbioru, co nie jest zbyt wydajne. Innym rozwiązaniem jest trzymanie wartości w posortowanej tablicy. Wówczas wyszukiwanie binarne elementu będzie działało szybko, ale wstawianie nowego elementu wymagać będzie bądź ponownego sortowania, bądź kopiowania pewnej liczby elementów, być może dużej. Istnieją jednak struktury, które wszystkie te operacje mogą wykonywać szybko.

Drzewo wyszukiwań binarnych

Struktura ta służy do przechowywania elementów uporządkowanych (np. rosnąco), czym różni się od innych struktur, jak choćby stosu. Po angielsku nazywa się ona *binary search tree*, stąd powszechnie używany skrót BST.

Drzewa BST są ukorzenione (tzn. mają jeden wybrany wierzchołek będący korzeniem całego drzewa) oraz binarne (czyli każdy wierzchołek ma co najwyżej dwóch synów). W każdym wierzchołku przechowywane są wartości, nazywane również **kluczami**. Tym, co odróżnia drzewa BST od, na przykład, kopca jest to, że klucze wierzchołków spełniają dodatkowy warunek:

> *Porządek symetryczny:* dla każdego wierzchołka o kluczu v wierzchołki w jego lewym poddrzewie mają wartości mniejsze bądź równe v, a w prawym większe bądź równe.



Rys. 1: Przykładowe drzewo BST

Warunek porządku symetrycznego pokazuje już, jak będziemy wyszukiwali elementów w drzewach BST: jeżeli szukany element jest mniejszy od klucza aktualnego wierzchołka to szukamy w lewym poddrzewie, w przeciwnym przypadku w prawym.

Struktura implementacji

Implementacja drzew BST jest trochę bardziej skomplikowana od kopca, gdyż nie można używać po prostu jednej tablicy. Zamiast niej, podobnie jak w przypadku stosów, kolejek i list, będziemy korzystać ze struktur dla wierzchołków i łączyć je wskaźnikami.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

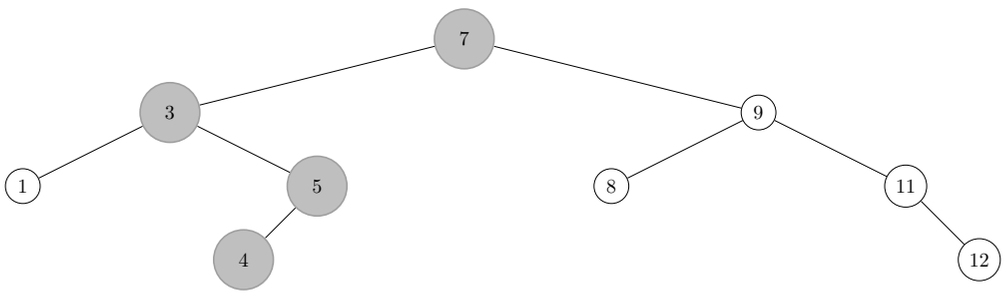
```
struct Node
{
    // Wskaźniki do lewego i prawego syna, oraz ojca.
    // wartość NULL oznacza brak odpowiedniego wierzchołka.
    Node *left, *right, *parent;
    int key; // klucz, czyli wartość przechowywana w wierzchołku
};
```

Całe drzewo będziemy przechowywali w programie jako obiekt:

```
struct BST
{
    Node * root; // wskaźnik na korzeń drzewa
    BST()        // konstruktor pustego drzewa
    {
        root = NULL;
    }
    /* tu należy umieszczać kolejne funkcje */
};
```

Wyszukiwanie elementu

Skoro znamy już strukturę drzewa BST możemy zaimplementować wyszukiwanie elementu w drzewie. Algorytm oparty na metodzie dziel i zwyciężaj, został już pobieżnie opisany: poczynając od całego drzewa sprawdzamy korzenie kolejnych poddrzew: jeżeli klucz takiego korzenia jest większy od poszukiwanego to kontynuujemy w lewym poddrzewie, w przeciwnym przypadku w prawym.



Rys. 2: Poszukiwanie elementu o kluczu 4

```
Node * search(int v)
{
    Node *s = root;
    // gdy s == NULL, oznacza to, że wyszliśmy ,,poza drzewo''
    while (s != NULL && s->key != v){
        if (s->key > v)
            s = s->left;
        else
            s = s->right;
    }
    return s;
    // funkcja zwraca NULL, gdy brak wierzchołka o kluczu v
}
```

Nie trudno jest zauważyć, że metoda ta tak na prawdę przechodzi po jakiejś ścieżce w drzewie, odwiedzając każdy wierzchołek raz. Jej złożoność czasowa to zatem $O(h)$, gdzie h jest wysokością drzewa (czyli długością najdłuższej ścieżki z korzenia do liścia).

Ćwiczenie 13. Wymyśl i zaimplementuj algorytm znajdujący najmniejszy klucz w drzewie BST.

Ćwiczenie 14. W jakiej kolejności należałoby odwiedzać, wierzchołki drzewa, by ich klucze przeglądać w kolejności niemalejącej? Napisz metodę klasy BST wypisującą wszystkie klucze z drzewa w kolejności niemalejącej.

Ćwiczenie 15. Następnikiem wierzchołka s nazywamy wierzchołek, który zostanie wypisany tuż po s, w powyższym algorytmie przeglądania drzewa. Wykaż, że jeżeli s ma prawego syna, to następnikiem s jest wierzchołkiem o najmniejszym kluczu w jego prawym poddrzewie. Zaimplementuj funkcję znajdującą następnik w takim przypadku. Który wierzchołek jest następnikiem s, gdy nie ma on prawego poddrzewa? Czy następnik będzie zawsze istniał?

Dodawanie elementu

Jeżeli chcemy by nasze drzewa okazały się użyteczne musimy nauczyć się dodawać do nich nowe wierzchołki tak, by został zachowany porządek symetryczny. W tym celu należy, naśladując algorytm wyszukiwania elementu, przejść przez drzewo, aż do wierzchołka, który nie ma odpowiedniego (prawego lub lewego) syna i dodać tam nowy wierzchołek.

```
void insert(int v)
{
    Node *n = new Node(), *s = root, *prev = NULL;
    // inicjalizacja nowego wierzchołka
    n->left = NULL;
    n->right = NULL;
    n->key = v;

    while (s != NULL){  // naśladowanie wyszukiwania
        prev = s;
        if (s->key >= v)
            s = s->left;
        else
            s = s->right;
        // Niezmiennik: zmienna prev wskazuje
        // na poprzednio odwiedzony wierzchołek
    }
    // pod wierzchołek prev dowiązujemy n z odpowiedniej strony
    if (prev->key >= v)
        prev->left = n;
    else
        prev->right = n;
    n->parent = prev;
}
```

Ćwiczenie 16. Powyższy kod działa jedynie dla drzewa o co najmniej jednym elemencie. Dlaczego? Co się stanie gdy zostanie uruchomiona na pustym drzewie? Dopisz do metody obsługę tego specjalnego przypadku.

Usuwanie wierzchołka

Usuwanie wierzchołka jest trochę bardziej skomplikowane od dodawania, gdyż trzeba uważać na więcej przypadków szczególnych. Jeżeli chcemy usunąć wierzchołek s, to musimy zadbać o poprawne działanie w następujących przypadkach:

1. s nie ma synów — możemy po prostu skasować ten wierzchołek;
2. s ma jednego syna (lewego lub prawego). Wówczas ojcu s jako bezpośredniego potomka (i to po odpowiedniej stronie!) ustawiamy jedynego syna s;
3. s ma dwóch synów. Należy wówczas znaleźć następnika s, jego klucz zapisać w wierzchołku s i usunąć następnika. Zauważmy, że operacja ta nie zaburza porządku symetrycznego w drzewie, gdyż następnik ma najmniejszy klucz, nie większy niż ten usuwany.

Napisanie bezbłędnie metody z taką liczbą możliwych przypadków stanowi pewne wyzwanie, do którego podjęcia zachęcamy!

Ćwiczenie 17. Uzupełnij poniższą metodę tak, by poprawnie działała we wszystkich przypadkach. Nie zapomnij o poprawieniu wskaźników `parent` oraz o tym, że usuwany wierzchołek może być korzeniem drzewa!

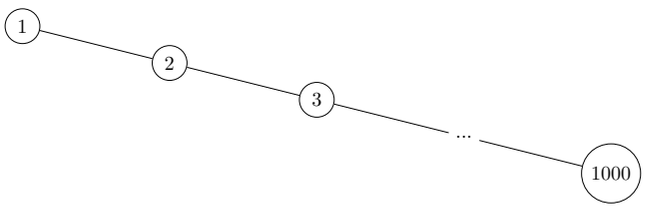
```
void erase(Node * s) {
    Node *next;
    if (s->left == NULL || s->right==NULL)
    {
        // przypadki 1 i 2: s nie ma co najmniej jednego syna
        // poprawnie rozpatrz te dwa przypadki!

        delete s;
    }
    else
    {
        // przypadek 3, szukanie następnika
        next = s->right;
        while (next->left != NULL)
            next = next->left;
        s->key = next->key;
        erase(next);
        // dzięki powyższemu odwołaniu trzeba sprawdzić mniej przypadków
    }
}
```

7.1 Zrównoważone drzewa poszukiwań

Problem z efektywnością drzew BST

Operacje dodawania, usuwania i wyszukania na drzewach BST działają w czasie proporcjonalnym do wysokości drzewa. W przypadku danych losowych, wysokość ta jest rzędu $O(\log n)$, co jest zupełnie zadowalające. Niestety, może się zdarzyć, że ciąg elementów dodawanych do drzewa spowoduje, że będzie miało ono postać jednej długiej ścieżki bez rozgałęzień, jak na rysunku 3.



Rys. 3: Złośliwy przypadek drzewa BST

Wówczas czas działania wszystkich operacji na drzewie będzie liniowo zależny od liczby elementów, a zatem nic nie zyskujemy na korzystaniu z drzew BST w porównaniu do zwykłej listy!

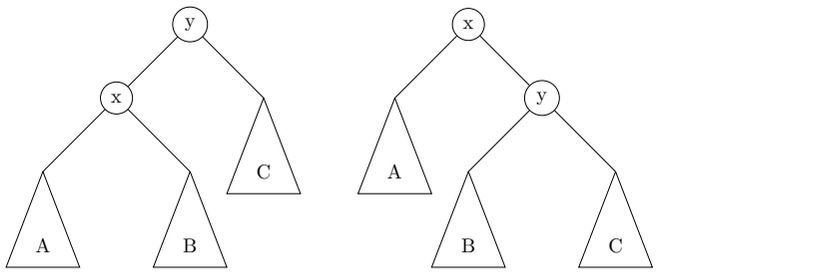
Drzewa AVL

Aby uporać się z powyższym problemem stosuje się różne techniki. Chyba najprostszą do zrozumienia są drzewa AVL, zaproponowane przez Gieorgija Adelson-Wielskija i Jewgienija Łandisa. Pomysł polega na dodaniu drzewom BST następującego warunku:

> dla każdego wierzchołka, wysokości jego prawego i lewego poddrzewa mogą różnić się o co najwyżej 1

Można udowodnić, że wysokość drzewa o n wierzchołkach spełniającego ten warunek jest $O(\log n)$.

Operacje na drzewach AVL są podobne do tych na BST. Różnica jest tylko taka, że po operacjach dodawania, bądź usuwania wierzchołków, które mogą zmienić strukturę drzewa, należy przywrócić warunek zrównoważania. Można to zrobić za pomocą tzw. rotacji, którym przyjrzymy się na przykładzie z rys. 4.



Rys 4. Pojedyncza rotacja

Załóżmy, że poddrzewo A ma wysokość $h+1$, a poddrzewa B i C mają wysokości h. Wówczas w wierzchołku x poddrzewa różnią się wysokością o 1, co jest dopuszczalne, ale już w wierzchołku y wysokości poddrzew różnią się o 2. Pojedyncza rotacja, przedstawiona na obrazku, może to jednak poprawić: po jej wykonaniu poddrzewa będą miały równe wysokości, zarówno x jak i y.

Jest to oczywiście bardzo ogólny opis działania drzew AVL. Zainteresowanych odsyłamy do pozycji [2] i [3]. Istnieją też inne sposoby równoważenia drzew binarnych. Jako szczególnie proste do implementacji polecamy samoorganizujące się drzewa BST (nazywane niekiedy drzewami *splay*), o których też można poczytać w wymienionych materiałach.

Kontenery z biblioteki standardowej

Jak się okazuje, w standardowej bibliotece szablonów (STL) języka $C++$ znajdują się struktury danych oparte na zrównoważonych drzewach binarnych. Można z nich korzystać w bardzo wielu przypadkach, ale niestety nie wszystkich: czasami trzeba zaimplementować swoje własne drzewa.

Dwa często stosowane kontenery tego typu to `set` (reprezentuje zbiór, zatem nie mogą się w nim powtarzać wartości) oraz `multiset`. Kilka wskazówek, jak się nimi posługiwać:

- W programie trzeba umieścić:

```
#include<set>
using namespace std;
```

- Deklaracja zbioru i multizbioru elementów typu `int`:

```
set<int> zb;
multiset<int> mzb;
```

- Dodawanie, usuwanie i szukanie elementu:

```
zb.insert(6);
zb.erase(7);
if (zb.find(4) != zb.end())
{
    // 4 jest w zbiorze zb
}
```

- Wypisanie elementów zbioru w kolejności rosnącej:

```
for(set<int>::iterator it = zb.begin(); it != zb.end(); ++it)
    printf("%d\n", *it);
```

Więcej o korzystaniu z tych oraz innych kontenerów biblioteki STL będzie można dowiedzieć się na innych kursach. Można też korzystać z dokumentacji (w języku angielskim) dostępnej pod adresem: `http://www.sgi.com/tech/stl/`

Zadanie

A. **Przedziały**
Napisz program przechowujący informacje o zbiorze przedziałów na prostej. Będzie on otrzymywał kolejne polecenia do wykonania, w jednej z poniższych postaci:

- $1\ p\ k$ — dodanie do zbioru przedziału o początku w p i końcu w k (p i k będą liczbami całkowitymi, $-10^9 \leqslant p < k \leqslant 10^9$).
- 0 — zapytanie o liczbę *rozłącznych* przedziałów w zbiorze,
- -1 — koniec listy poleceń.

Przykładowe wejście:
1 1 2
1 3 4
0
1 2 3
0
−1

Przykładowe wyjście:
2
1

8 Drzewa przedziałowe

Czym są drzewa przedziałowe?

Drzewami przedziałowymi nazywamy struktury danych, umożliwiające szybkie wykonywanie operacji na zbiorze przedziałów, takich jak:

- wstawienie przedziału do zbioru (być może z pewną wagą),
- usunięcie przedziału,
- sprawdzenie w ilu przedziałach zawiera się dany punkt,
- odczytanie sumarycznej wagi punktów z danego przedziału, etc.

Na wykładzie przedstawiamy dwa podejścia do implementacji drzewa przedziałowego. Pierwsze z nich nazywane jest również **drzewem potęgowym**.

Ustalenia wstępne

Zakładamy, że wszystkie rozważane przedziały mają końce w punktach całkowitoliczbowych z ustalonego zakresu będącego potęgą dwójki (zakres ten oznaczać będziemy przez N). Inaczej niż w geometrii, przez długość przedziału rozumiemy liczbę punktów o współrzędnych całkowitych w nim zawartych. Przykładowo przedział $[3, 3]$ ma długość 1 gdyż zawiera dokładnie jeden punkt. Łatwiej jest zatem myśleć o punktach jak o komórkach tablicy.

Zakres współrzędnych przedziałów przechowywanych w drzewie nie może być zbyt duży, w szczególności tablica liczb całkowitych o rozmiarze $2N$ (a najczęściej kilka takich tablic) musi swobodnie mieścić się w limicie pamięciowym. W niektórych przypadkach możemy poradzić sobie z bardzo dużym zakresem przedziałów stosując pewną sztuczkę. Jeśli interesuje nas jedynie ułożenie przedziałów względem siebie (a nie ich faktyczna długość) możemy wstępnie wczytać dane wejściowe, posortować wszystkie występujące w nich punkty i nadać im nowe współrzędne będące kolejnymi liczbami naturalnymi. Po takim zabiegu zakres współrzędnych będzie wielkości danych wejściowych.

Każda z omawianych struktur danych będzie udostępniać dwie operacje:

- **wstawienie** (funkcja `insert()`) — wykonanie pewnej akcji (dodanie obciążenia lub wyciągnięcie maksimum) na przedziale lub w punkcie,
- **zapytanie** (funkcja `query()`) — odczytanie aktualnego stanu (sumy lub maksimum) z przedziału lub punktu, będącego wynikiem wykonanych operacji `insert`.

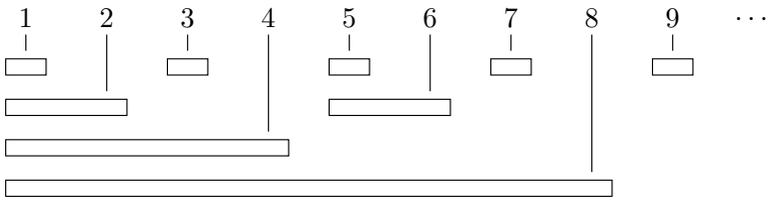
8.1 Drzewo potęgowe

Załóżmy, że potrzebujemy struktury danych umożliwiającej wykonywanie następujących operacji:

- `insert(x,v)` — dodanie wartości (inaczej obciążenia) v do punktu x
- `query(a,b)` — zsumowanie obciążeń punktów zawartych w przedziale $[a, b]$

Trywialną realizacją powyższych wymagań może być tablica, w której po prostu zapisujemy wstawione wartości. Niestety, pomimo tego, iż funkcja `insert` działa w czasie stałym, zliczenie sumy liczb w przedziale wiąże się z kosztem proporcjonalnym do jego długości, co w przypadku dużej liczby wywołań `query` dla długich przedziałów jest zdecydowanie zbyt wolne.

Drzewo potęgowe opiera się na pomyśle, aby w tablicy (nazwijmy ją `load`) nie przechowywać obciążenia jednego punktu, lecz sumę obciążeń trochę większego obszaru. Dokładniej, w polu `load[x]` będziemy pamiętać sumę wszystkich operacji `insert` dla punktów z przedziału $[x - p + 1, x]$, gdzie p jest największą potęgą dwójki dzielącą x.



Co nam daje taka modyfikacja? Na pewno komplikuje operację `insert`, gdyż dodając wartość do pewnego punktu musimy uaktualnić potencjalnie wiele pól tablicy. Przykładowo, wykonując `insert` w punkcie nr 3 musimy uaktualnić pola `load[3]`, `load[4]`, `load[8]`, `load[16]`, itd. Okazuje się jednak, że liczba niezbędnych zmian jest niewielka.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Ćwiczenie 18. Uzasadnij, że liczba pól tablicy wymagających aktualizacji przy operacji `insert` jest rzędu $O(\log N)$.

Skąd wiadomo, które pola w tablicy uaktualnić? Zauważmy, że jeśli uaktualniliśmy pole o indeksie x, a p jest największą potęgą dwójki dzielącą x, to następnym polem, które "obejmie swoim zasięgiem" pole x, jest $x+p$. Wystarczy zatem zacząć od ustawienia wartości pola `load[x]`, a następnie przesuwać się do kolejnych pól, za każdym razem zwiększając indeks o największą potęgę dwójki dzielącą indeks pola aktualnego.

No dobrze, a jak w takim razie wyznaczyć tę potęgę dwójki? Można to zrobić łatwo w złożoności logarytmicznej, my jednak chcielibyśmy umieć wykonać to w czasie stałym. Z pomocą przychodzą operatory bitowe, dzięki którym szukaną liczbę p wyznaczymy w sposób następujący:

```
p = ( (x ^ (x - 1)) + 1 ) / 2
```

Aby lepiej zrozumieć powyższy wiersz, przeanalizujmy jego działanie na przykładzie.
Niech $x = 20$, czyli w zapisie binarnym $10100_{(2)}$:

`x`	=	$10100_{(2)}$
`x - 1`	=	$10011_{(2)}$
`x ^ (x - 1)`	=	$00111_{(2)}$
`(x ^ (x - 1)) + 1`	=	$01000_{(2)}$
`((x ^ (x - 1)) + 1) / 2`	=	$00100_{(2)}$

Skoro umiemy wyznaczyć największą potęgę dwójki dzielącą daną liczbę w czasie stałym, to umiemy też zaimplementować operację `insert` tak, aby działała w czasie O(log n):

```
void insert(int x, int v)
{
    while(x < N)
    {
        load[x] += v;
        x += ( (x ^ (x - 1)) + 1) / 2;
    }
}
```

Jak dotąd udało nam się jedynie pogorszyć złożoność operacji `insert`. Okazuje się jednak, że dzięki poczynionym modyfikacjom uda nam się zredukować złożoność operacji `query` do $O(\log N)$! Aby uprościć sobie trochę implementację, zauważmy, że zamiast pytać o sumaryczne obciążenie na przedziale $[a, b]$ wystarczy, że poznamy sumaryczne obciążenia na przedziałach $[1, a-1]$ i $[1, b]$. Oto kod funkcji `query` wraz z pomocniczą funkcją `sum`:

```
int sum(int x)
{
    int res = 0;
    while(x > 0)
    {
        res += load[x];
        x -= ( (x ^ (x - 1)) + 1) / 2;
    }
    return res;
}

int query(int a, int b)
{
    return sum(b) - sum(a-1);
}
```

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

8.2 Drzewa przedziałowe

Drzewo potęgowe sprawdza się bardzo dobrze w przypadku gdy operacja `insert` dotyczy punktów. Jeśli jednak musimy dodawać obciążenia na całe przedziały, potrzebujemy struktury nieco bardziej rozbudowanej. Jako że zapytanie o punkt możemy traktować jak zapytanie o przedział długości 1, skupimy się od razu na najogólniejszym przypadku, w którym wszystkie operacje dotyczą przedziałów.

Rozkład na przedziały bazowe

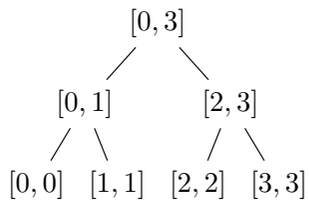
Definicja 5. Przedziałem bazowym nazywamy przedział o długości będącej potęgą dwójki i początku w punkcie będącym wielokrotnością swojej długości.

Interesuje nas rozkład danego przedziału na *minimalną* liczbę rozłącznych przedziałów bazowych, które pokrywają dany przedział. Np. przedział $[3, 9]$ ma rozkład na sumę przedziałów $[3, 3]$, $[4, 7]$ i $[8, 9]$.

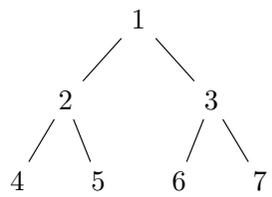
Kluczowy tutaj jest następujący fakt:

Ćwiczenie 19. Uzasadnij, że liczba przedziałów bazowych w rozkładzie jest rzędu $O(\log k)$, gdzie k jest długością rozkładanego przedziału.

Tym razem nasze drzewo przedziałowe będzie pełnym drzewem binarnym, którego węzły odpowiadać będą przedziałom bazowym.



Obciążenia poszczególnych przedziałów bazowych zapisywać będziemy w tablicy `load[]` o rozmiarze $2N$, numerując węzły drzewa w następujący sposób:



Drzewo z powyższą numeracją węzłów ma następujące własności:

- ojcem węzła x jest $x/2$,
- lewy syn węzła x ma indeks $2x$ a prawy $2x + 1$,
- węzły parzyste są zawsze lewymi synami swoich ojców, natomiast nieparzyste prawymi,
- drzewo przedziałowe o zakresie od 0 do $N-1$ ma $2N-1$ węzłów a jego wysokość to $\log N+1$.

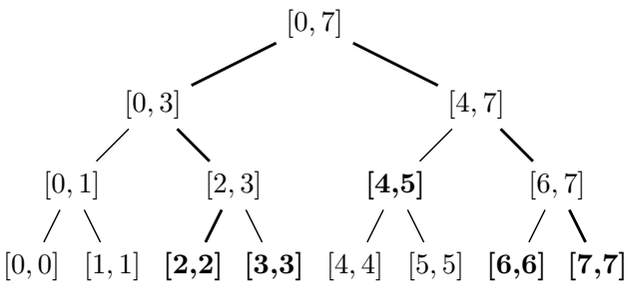
KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

8.2.1 Drzewo typu (+,+)

Zacznijmy od omówienia struktury udostępniającej następujące operacje:

- `insert(a,b,v)` — dodanie obciążenia v do punktów z przedziału $[a, b]$
- `query(a,b)` — zsumowanie obciążeń punktów zawartych w przedziale $[a, b]$

Jak zrealizować metodę `insert`? Wiemy, że każdy przedział długości k można rozłożyć na sumę $O(\log k)$ podprzedziałów bazowych. Wystarczy więc, że znajdziemy rozkład przedziału $[a, b]$ a następnie dodamy wartość v do każdego węzła drzewa odpowiadającego podprzedziałowi z rozkładu. Aby uprościć implementację, zaczniemy od znalezienia liści odpowiadających przedziałom $[a, a]$ i $[b, b]$, a następnie, startując od nich, będziemy poruszać się w górę drzewa, uaktualniając napotkane węzły odpowiadające przedziałom zawartym w $[a, b]$.



Przedziały pogrubione tworzą rozkład przedziału $[2, 7]$. Warto zauważyć, że nie zawsze otrzymamy rozkład na *minimalną* liczbę podprzedziałów. Łatwo jednak uzasadnić, że liczba odwiedzonych węzłów będzie rzędu $O(\log N)$.

Jeżeli będziemy jednocześnie przechodzić ścieżkami od lewego i od prawego końca przedziału to nasze ścieżki w pewnym momencie się spotkają (być może dopiero w korzeniu). Jeśli nasze ścieżki jeszcze się nie spotkały i w pewnym momencie znajdziemy się w węźle leżącym na *prawej* ścieżce, który jest *prawym* synem swojego ojca, to kandydatem na przedział należący do szukanego rozkładu jest brat obecnego węzła. Analogicznie w sytuacji gdy znajdziemy się w węźle leżącym na *lewej* ścieżce, który jest *lewym* synem swojego ojca.

Czy to wystarczy aby móc zaimplementować operację `query`? Niestety nie, ponieważ może się zdarzyć, że rozkłady dwóch zachodzących na siebie przedziałów nie będą mieć wspólnych elementów.

Ćwiczenie 20. Podaj przykład takich przedziałów.

Musimy zatem przechowywać w węzłach drzewa dodatkowe informacje. W dodatkowej tablicy `sub[x]` pamiętać będziemy sumę obciążeń wszystkich węzłów poddrzewa o korzeniu x, które są zapisane w tym poddrzewie. Przechodząc ścieżkami w górę drzewa musimy pamiętać o aktualizowaniu wartości `sub` we wszystkich napotkanych węzłach, korzystając z zależności rekurencyjnej:

```
sub[x] = sub[left(x)] + sub[right(x)] + load[x] * length(x)
```

gdzie `left(x)`, `right(x)` to potomkowie węzła x, a `length(x)` to długość przedziału reprezentowanego przez węzeł x.

Oto kod funkcji `insert`:

```
void insert(int a, int b, int v)
{
    int l = N + a, r = N + b;
    int length = 1; // długość przedziałów na aktualnie odwiedzanym poziomie
```

```
    load[l] += v;
    sub[l] += v;
    // jeśli a==b to nie dodajemy obciążenia dwukrotnie
    if(r != l)
    {
        load[r] += v;
        sub[r] += v;
    }

    while(l >= 1)
    {
        // jeśli l i r nie są sąsiadami w drzewie, to sprawdzamy czy
        // nie trzeba uaktualnić węzłów wewnętrznych
        if(l < r - 1)
        {
            if(l % 2 == 0) // l jest lewym synem swego ojca
            {
                load[l + 1] += v;
                sub[l + 1] += v * length;
            }
            if(r % 2 == 1) // r jest prawym synem swego ojca
            {
                load[r - 1] += v;
                sub[r - 1] += v * length;
            }
        }

        // jeśli l i r nie są liśćmi, to uaktualniamy ich wartości sub
        if(r < N)
        {
            sub[l] = sub[2 * l] + sub[2 * l + 1] + load[l] * length;
            sub[r] = sub[2 * r] + sub[2 * r + 1] + load[r] * length;
        }

        // przechodzimy poziom wyżej
        l /= 2; r /= 2; length *= 2;
    }
}
```

Implementacja funkcji `query` jest podobna:

```
int query(int a, int b)
{
    int l = N + a, r = N + b;
    int length = 1; // długość przedziałów na aktualnie odwiedzanym poziomie

    // w llen i rlen pamiętamy ile punktów przedziału [a,b] zawiera
    // się w poddrzewie o korzeniu l i r odpowiednio
    int llen = 1, rlen = (a != b ? 1 : 0);
    int res = 0;

    while(l >= 1)
    {
```

```
            // sumujemy obciążenia z węzłów l i r
            res += llen * load[l] + rlen * load[r];

            // jeśli l i r nie są sąsiadami w drzewie to sprawdzamy czy
            // istnieją węzły wewnętrzne z obciążeniem
            if(l < r - 1)
            {
                if(l % 2 == 0) // l jest lewym synem swego ojca
                {
                    res += sub[l + 1];
                    llen += length;
                }
                if(r % 2 == 1) // r jest prawym synem swego ojca
                {
                    res += sub[r - 1];
                    rlen += length;
                }
            }

            // przechodzimy poziom wyżej
            l /= 2; r /= 2; length *= 2;
        }
        return res;
    }
```

8.2.2 Drzewo typu (+,max)

Drzewo typu (+,max) udostępnia następujące metody:

- `insert(a,b,v)` — dodanie obciążenia v do punktów z przedziału $[a, b]$
- `query(a,b)` — znalezienie maksymalnego obciążenia punktu należącego do $[a, b]$

Konstrukcja drzewa jest niemal identyczna jak w przypadku (+,+), z tym, że w tablicy `sub[x]` będziemy przechowywać *maksymalne obciążenie wierzchołka z poddrzewa o korzeniu w* x.

Poniżej znajduje się kod metody `insert`. Implementacja `query` jest nietrudnym ćwiczeniem.

```
 void insert(int a, int b, int v)
 {
     int l = N + a, r = N + b;

     load[l] += v;
     sub[l] += v;
     // jeśli a==b to nie dodajemy obciążenia dwukrotnie
     if(r != l)
     {
         load[r] += v;
         sub[r] += v;
     }

     while(l >= 1)
     {
         // jeśli l i r nie są sąsiadami w drzewie to sprawdzamy czy
         // nie trzeba uaktualnić węzłów wewnętrznych
```

```
            if(l < r - 1)
            {
                if(l % 2 == 0) // l jest lewym synem swego ojca
                {
                    load[l + 1] += v;
                    sub[l + 1] += v;
                }
                if(r % 2 == 1) // r jest prawym synem swego ojca
                {
                    load[r - 1] += v;
                    sub[r - 1] += v;
                }
            }

            // jeśli l i r nie są liśćmi to uaktualniamy ich wartości sub
            if(r < N)
            {
                sub[l] = max(sub[2 * l], sub[2 * l + 1]) + load[l];
                sub[r] = max(sub[2 * r], sub[2 * r + 1]) + load[r];
            }

            // przechodzimy poziom wyżej
            l /= 2; r /= 2;
        }
    }
```

8.2.3 Drzewo typu (max,max)

Tym razem nasze drzewo udostępnia następujące metody:

- `insert(a,b,v)` — dla każdego punktu x należącego do przedziału $[a, b]$ wykonanie podstawienia `load[x] = max(load[x],v)`
- `query(a,b)` — znalezienie maksymalnego obciążenia punktu należącego do $[a, b]$

Implementacja nieznacznie różni się od implementacji drzewa (+,max) — pozostawiamy ją jako ćwiczenie.

Zadania

1. Zaimplementuj rekurencyjne odpowiedniki metod `insert` i `query` dla dowolnego z omawianych drzew przedziałowych. Porównaj czasy działania obu wersji dla dużych danych wejściowych.
2. Jeżeli x jest zmienną typu `int`, to wyrażenie `((x ^ (x - 1)) + 1) / 2` możemy zastąpić przez `(x & -x)`. Upewnij się, że rozumiesz dlaczego wyrażenia te zwracają tę samą wartość.
3. Do których drzew można wstawiać za pomocą `insert` wartości ujemne? Które wymagają modyfikacji?
4. Zadanie *Koleje* z IX Olimpiady Informatycznej (dostępne w [5]).

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

9 Technika zamiatania

Wprowadzenie

Technika zamiatania jest jednym z podstawowych podejść do rozwiązywania problemów geometrycznych. Na tych zajęciach pokażemy przykłady kilku problemów, które można rozwiązać tym sposobem.

Aby przeprowadzić zamiatanie potrzebujemy **miotły**. Zwykle jest to prosta, która przesuwa się nad całą płaszczyzną i przegląda kolejno napotykane obiekty (np. punkty, czy też figury). Umówmy się na potrzeby tych notatek, że miotła jest pionowa i przesuwa się od lewej do prawej, czyli zgodnie ze wzrostem współrzędnej x. W miarę napotykania kolejnych obiektów miotła zapamiętuje informacje potrzebne do rozwiązania problemu. W typowym przypadku wykorzystuje do tego strukturę danych taką, jak na przykład drzewo przedziałowe, o którym mówiliśmy na poprzednich zajęciach.

Pary przecinających się odcinków

Zamiatanie najlepiej pokazać na konkretnym przykładzie:

> Na płaszczyźnie znajduje się n pionowych i poziomych odcinków. Chcemy policzyć wszystkie przecięcia odcinków pionowych z poziomymi.

Zadanie to rozwiązać można prostym algorytmem w czasie $O(n^2)$ — wystarczy dla każdej pary prostopadłych odcinków stwierdzić, czy przecinają się ze sobą.

Ćwiczenie 21. Jak sprawdzałbyś, czy dwa odcinki są prostopadłe? Jak można użyć w tym celu iloczynu skalarnego wektorów?

My jednak pokażemy znacznie szybsze rozwiązanie, które działa w czasie $O(n \log n)$, oparte na technice zamiatania. Nasza miotła, czyli pionowa prosta przesuwająca się w prawo, będzie się zatrzymywać, gdy napotka jedno z następujących **zdarzeń**:

- początek poziomego odcinka,
- koniec poziomego odcinka,
- odcinek pionowy.

Chcemy, by w każdym momencie swojej wędrówki miotła znała odcinki poziome, które się pod nią znajdują (przecinają się z nią). W rzeczywistości wystarczy przechowywać współrzędne igrekowe tych odcinków. W tym celu, gdy miotła napotka początek poziomego odcinka, musi zapamiętać jego współrzędne, aż do momentu osiągnięcia jego końca.

W momencie, gdy pod miotłą pojawi się odcinek pionowy, będziemy zliczali ile odcinków poziomych się z nim przecina. Załóżmy, że współrzędne igrekowe jego dolnego i górnego końca to odpowiednio y_1 i y_2. Zapytamy więc miotłę ile odcinków poziomych, które aktualnie znajdują się w miotle, ma współrzędne z przedziału $[y_1, y_2]$. Po zsumowaniu wyników dla wszystkich napotkanych pionowych odcinków otrzymamy ostateczny rezultat.

Pozostaje jeszcze powiedzieć, jak zrealizować miotłę, by można było szybko dodawać i usuwać współrzędne odcinków, a także zliczać współrzędne z określonych przedziałów. Wykorzystamy w tym celu drzewo potęgowe! Każdy jego punkt będzie reprezentował jedną współrzędną igrekową, zaś obciążeniem punktu jest liczba odcinków, które są aktualnie pod miotłą na tej współrzędnej. Operacje, które wykonuje miotła odpowiadają dokładnie operacjom udostępnianym przez drzewo potęgowe — dokonuje ona zmiany obciążenia w danym punkcie oraz sumowania obciążeń z określonego przedziału.

Przyjrzyjmy się jeszcze raz strukturze całego rozwiązania. Na początku musimy stworzyć i posortować po pierwszej współrzędnej wszystkie zdarzenia, które napotka miotła. Można to zrobić

w czasie $O(n \log n)$. Później, każde zdarzenie obsługujemy w czasie $O(\log n)$. W ten sposób udało nam się rozwiązać pierwszy problem, tak jak obiecywaliśmy, w czasie $O(n \log n)$.

Ćwiczenie 22. Co należy zrobić gdy nie można po prostu zastosować drzewa potęgowego, gdyż przedział, w jakim znajdują się współrzędne końców odcinków jest zbyt duży?

Implementacja

Na początku napiszmy strukturę opisującą zdarzenie.

```
struct event
{
    int x; /* współrzędna, na której występuje zdarzenie */
    /* Typ zdarzenia:                    *
    /*  1 - początek odcinka poziomego   *
     * -1 - koniec odcinka poziomego     *
     *  0 - odcinek pionowy              */
    int type;

    int y1, y2; /* dolna i górna współrzędna y      *
                 * (dla odcinków poziomych y1 = y2) */
    int operator<(const event& e) const
    {
        if(x == e.x)
            return type > e.type;
        else
            return x < e.x;
    }
};
```

Funkcja `operator<` definiuje **operator porównujący** obiekty danej struktury. Powinna ona zwracać wartość `true` wtedy i tylko wtedy, gdy obiekt lokalny jest mniejszy od obiektu przesłanego jako parametr. Dzięki niej możemy porównywać dwie zmienne typu `event` jak zwykłe liczby. Gdy mamy zdefiniowany taki operator, możemy też posortować tablicę, lub wektor obiektów klasy `event` przez wywołanie funkcji `sort` z biblioteki STL.

W wyniku tego sortowania chcemy mieć zdarzenia ułożone w kolejności, w jakiej ma je napotykać miotła. Dlatego operator ten porządkuje zdarzenia w pierwszej kolejności po współrzędnej x, a w przypadku remisu porównuje również typy zdarzeń. Kolejność, w jakiej należy przetwarzać zdarzenia o tej samej pierwszej współrzędnej, należy dokładnie przemyśleć. Wyobraźmy sobie, że na pewnej współrzędnej x zachodzą zdarzenia wszystkich trzech typów: jeden odcinek poziomy się rozpoczyna, inny się kończy, a pewien odcinek pionowy dotyka obydwóch odcinków poziomych. Sprawdzenie, ilu odcinków poziomych dotyka ten pionowy trzeba więc wykonać po wstawieniu odcinków, które zaczynają się na danej współrzędnej, ale przed usunięciem tych, które właśnie się kończą. Stąd, poszczególne typy zdarzeń, w ramach tej samej pierwszej współrzędnej, są sortowane właśnie w tej kolejności. Pamiętaj, że za każdym razem gdy projektujemy algorytm używający techniki zamiatania, trzeba zastanowić się w jakiej kolejności należy przeglądać zdarzenia o tej samej pierwszej współrzędnej. Czasami odpowiednia kolejność jest jednoznacznie wyznaczona (jak w powyższym przykładzie), ale zdarza się też, że pewne obliczenia chcemy wykonać dopiero po obsłużeniu wszystkich zdarzeń na danej współrzędnej.

Przejdźmy teraz do kodu realizującego zamiatanie:

```
// events - kontener typu vector<event> przechowujący zdarzenia
// sortujemy zdarzenia zgodnie ze zdefiniowanym porządkiem
sort(events.begin(), events.end());
```

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

```
// zmienna zliczająca znalezione przecięcia
long long inter = 0;
for(int i=0; i<(int)events.size(); i++)
{
    if(events[i].type == 0) //odcinek pionowy
        inter += query(events[i].y1, events[i].y2);
    else
        insert(events[i].y1, events[i].type);
}
```

Korzystamy z drzewa potęgowego, omawianego na poprzednich zajęciach. Przypomnijmy, że `insert(x, v)` dodaje obciążenie v do punktu x, zaś `query(a,b)` zwraca sumę obciążeń na przedziale $[a, b]$.

Co jeszcze potrafi zamiatanie?

Wachlarz problemów, które potrafimy rozwiązać przez zamiatanie, zależy w dużej mierze od znanych nam struktur danych. Na przykład, korzystając z drzewa przedziałowego typu (+, max), rozwiązać możemy następujący problem: na płaszczyźnie znajdują się prostokąty o bokach równoległych do osi układu współrzędnych; znajdź punkt, który jest przykryty największą liczbą prostokątów.

Jako zdarzenia rozpatrujemy tu lewe i prawe boki prostokąta, a miotła pamięta, ile prostokątów znajduje się pod nią na każdej współrzędnej. Gdy napotykamy lewy bok prostokąta dodajemy jedynkę na przedziale wyznaczonym przez współrzędne igrekowe jego poziomych boków. Po każdym dodaniu prostokąta, szukamy punktu w miotle, który jest przykryty największą liczbą prostokątów.

Ćwiczenie 23. W tym problemie też należy uważać na sytuację, w której wiele zdarzeń ma tą samą pierwszą współrzędną. W jakiej kolejności należy je przeglądać, by algorytm działał poprawnie?

9.1 Zamiatanie kątowe

Zajmiemy się teraz trochę innym sposobem przeglądania obiektów na płaszczyźnie. Nie ma on dobrze rozpowszechnionej nazwy, jednak z powodu pewnych podobieństw do zamiatania, będziemy go nazywać **zamiataniem kątowym**.

W tym przypadku, miotła jest półprostą zaczepioną w ustalonym punkcie. Półprosta ta przegląda płaszczyznę wykonując obrót o 360°. Tym razem umówmy się, że półprosta obracać się będzie przeciwnie do ruchu wskazówek zegara. Rozważmy następujący przykładowy problem:

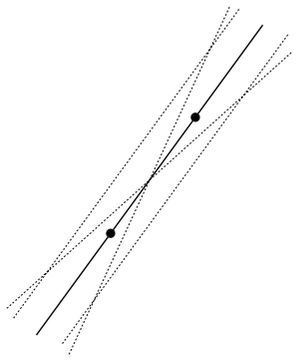
> Na płaszczyźnie znajduje się n punktów, spośród których żadne trzy nie leżą na jednej prostej. Z każdym punktem skojarzona jest waga, określona dodatnią liczbą całkowitą. Chcemy narysować na płaszczyźnie prostą, tak, by sumy wag punktów po obydwóch stronach prostej były sobie jak najbliższe.

Zastanówmy się najpierw jak napisać rozwiązanie, które rozpatrzy wszystkie możliwe proste. Po pierwsze, zauważmy, że możemy ograniczyć się do rozważania takich prostych, które przechodzą przez dokładnie dwa punkty. Nawet jeśli pewna optymalna prosta nie przechodzi przez żaden punkt, możemy ją tak przesunąć i obrócić, by oparła się na dwóch punktach.

Z drugiej strony, ponieważ nie ma trzech punktów współliniowych, jeśli mamy prostą przechodzącą przez dwa punkty, to możemy ją minimalnie przesunąć lub obrócić, tak by każdy z tych dwóch punktów znalazł się po jednej lub drugiej stronie prostej (patrz rysunek).

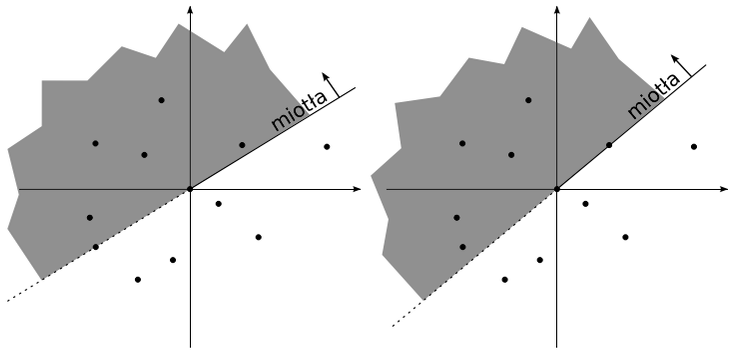
Rozwiązanie naszego problemu wygląda zatem tak: dla każdej pary punktów (takich par jest $O(n^2)$) prowadzimy przez nie prostą, liczymy sumy wag punktów po jednej i drugiej stronie (w czasie $O(n)$) a następnie rozpatrujemy 4 możliwości przydzielenia punktów z prostej do jednej lub drugiej części płaszczyzny. W ten sposób dostajemy algorytm w złożoności $O(n^3)$.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego



Rys. 1: Prosta przechodząca przez dwa punkty i cztery możliwości jej przesunięcia i obrotu

Korzystając z zamiatania kątowego poprawimy czas działania do $O(n^2 \log n)$. Dla każdego punktu z płaszczyzny, będziemy wykonywali dookoła niego zamiatanie kątowe. Chcemy w każdym momencie wykonania algorytmu znać sumę wag punktów na półpłaszczyźnie po lewej stronie miotły (tj. po stronie przeciwnej do ruchu wskazówek zegara). Gdy miotła napotka punkt, należy go z miotły usunąć. Jeśli zaś punkt znajdzie się na przedłużeniu miotły, jest on do niej dodawany. Sytuację wyjaśniają rysunki.



Rys. 2: Moment dodania punktu do miotły (po lewej) oraz usunięcia punktu (po prawej).

W momencie, gdy napotykamy punkt, sprawdzamy cztery możliwości narysowania prostej, odpowiadające możliwościom przydzielenia punktów do jednej z dwóch półpłaszczyzn. Można sobie zaoszczędzić nieco pracy i zauważyć, że wystarczy sprawdzać jedynie dwie opcje: punkt wokół którego zamiatamy przydzielamy do jednej lub drugiej strony płaszczyzny. W ten sposób również rozważymy wszystkie dostępne podziały płaszczyzny.

9.2 Sortowanie kątowe

Implementację zamiatania kątowego pozostawiamy jako ćwiczenie, opowiemy jedynie pokrótce, jak w prosty sposób posortować punkty kątowo. Z pomocą przychodzi nam funkcja `atan2` z biblioteki standardowej C++.

```
double atan2(double y, double x);
```

Tak, to nie pomyłka, pierwszym argumentem tej funkcji jest współrzędna y, zaś drugim — współrzędna x. Funkcja ta zwraca kąt, jaki tworzy odcinek łączący punkt (x, y) z dodatnią półosią X wyrażony w radianach (zwracana wartość jest liczbą z przedziału $[-\pi, \pi]$). Jeżeli zamiast na

radianach, wolimy operować na stopniach z przedziału $[-180, 180]$ to możemy otrzymaną wartość pomnożyć przez $\frac{180}{\pi}$. Aby użyć funkcji `atan2`, należy do programu dołączyć plik nagłówkowy `cmath`.

Trzeba być jednak ostrożnym używając funkcji `atan2`, ponieważ zapisuje ona wynik w liczbie zmiennoprzecinkowej (pamiętanej w komputerze z ograniczoną dokładnością). Jeśli kąty wyznaczone przez dwa punkty różnią się bardzo nieznacznie, `atan2` może zwrócić dla nich tę samą wartość. Zwykle uważa się, że jeżeli współrzędne punktów mają wartości bezwzględne większe niż 10^6, bezpieczniej jest użyć funkcji

```
long double atan2l(long double y, long double x);
```

i zmiennych typu `long double`. Jeśli współrzędne mogą być większe niż 10^9, nawet wyniki `atan2l` mogą być niewystarczająco dokładne.

Ćwiczenie 24. *Ambitne:* Jak wykorzystać iloczyn wektorowy do kątowego sortowania? Czy też trzeba się wtedy martwic o dokładność wyniku?

Zadania

1. Zadanie *Wyspy* z XI Olimpiady Informatycznej (dostępne w [5]).
2. Na płaszczyźnie zamalowujemy na czarno wnętrza n prostokątów o bokach równoległych do osi układu współrzędnych. Jakie jest pole zamalowanej części płaszczyzny?
3. Zaimplementuj rozwiązanie zadania z rozdziału o zamiataniu kątowym.
4. Na płaszczyźnie dane są punkty. Znajdź takie trzy spośród nich, które tworzą kąt o jak największej mierze.

10 Drzewa TRIE

Problem

Wiemy już, że za pomocą algorytmu KMP jesteśmy w stanie znaleźć wszystkie wystąpienia wzorca w tekście w czasie liniowym. Problem wyszukiwania wzorca można nieco uogólnić i zapytać jak szybko jesteśmy w stanie znaleźć wszystkie wystąpienia *wielu* wzorców w zadanym tekście.

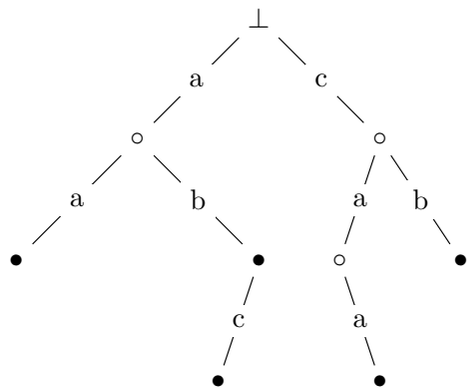
Możemy użyć algorytmu KMP do każdego wzorca z osobna a następnie scalić wyniki. Jeżeli wszystkich wzorców jest n a tekst ma długość m to rozwiązanie to ma złożoność $O(n \cdot m)$. Algorytm Aho-Corasick pozwala znacznie polepszyć ten wynik.

Drzewo TRIE

Będziemy korzystali ze struktury danych o nazwie TRIE. Jest to drzewo ukorzenione, którego krawędzie są etykietowane symbolami danego alfabetu. W dalszej części będziemy zakładali, że nasz alfabet to zbiór liter alfabetu angielskiego.

Drzewo TRIE reprezentuje pewien zbiór słów nad ustalonym alfabetem. Inaczej niż w dotychczas omawianych drzewach, elementy zbioru nie są przechowywane w węzłach lecz można je odtworzyć na podstawie pozycji węzłów w drzewie. Słowo reprezentowane przez węzeł x otrzymamy przechodząc po ścieżce od korzenia do x i odczytując litery na kolejnych krawędziach. Nie wszystkie węzły będą reprezentować elementy znajdujący się w zbiorze, dlatego w każdym węźle przechowujemy dodatkowo wartość logiczną, wskazującą czy węzeł ten odpowiada istniejącemu elementowi.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego



Na rysunku powyżej zamalowane węzły odpowiadają słowom przechowywanym w drzewie. Korzeń (oznaczony symbolem ⊥) odpowiada słowu pustemu. Powyższe drzewo reprezentuje zbiór słów {„aa”, „ab”, „abc”, „caa”, „cb”}, natomiast słowa „a”, „c” czy „ca” to tego zbioru nie należą.

Drzewo TRIE udostępniać będzie następujące metody:

- *insert* — wstawienie słowa do zbioru,
- *search* — sprawdzenie, czy słowo należy do zbioru,
- *erase* — usunięcie słowa ze zbioru.

W każdym węźle musimy przechowywać zbiór wskaźników do potomków. Jeśli nasz alfabet nie jest zbyt duży (a najczęściej będzie to alfabet angielski, czyli 26 liter) możemy pokusić się o dodanie do każdego węzła 26-elementowej tablicy wskaźników, co nieco uprości implementację. Wiąże się to jednak z trochę większym narzutem pamięciowym oraz koniecznością zerowania tablic w konstruktorze. Alternatywnym podejściem jest zastosowanie kontenera `map` z biblioteki STL, w którym przechowywane są pary (litera, odpowiadający jej wskaźnik do potomka). Aby nie komplikować nadmiernie implementacji, zastosujemy pierwszą metodę.
Oto (niepełna) implementacja drzewa TRIE:

```
#define ALPH 26    // rozmiar alfabetu

struct TrieNode    // węzeł drzewa TRIE
{
    // konstruktor ustawiający ojca i znak na krawędzi prowadzącej od ojca
    TrieNode(TrieNode *_parent, char _last)
    {
        for(int i = 0; i < ALPH; ++i)
            links[i] = NULL;
        parent = _parent;
        last = _last;
        in_dict = false;
    }

    TrieNode *links[ALPH];   // tablica potomków
    TrieNode *parent;        // wskaźnik do ojca
    char last;               // znak na krawędzi prowadzącej od ojca
    bool in_dict;            // czy słowo jest w drzewie
};
```

```
struct Trie
{
    Trie()   // konstruktor tworzący korzeń drzewa
    {
        root = new TrieNode(NULL, '-');
    }

    void insert(string &word)
    {
        TrieNode *v = root;   // wskaźnik na aktualny węzeł
        for(int i = 0; i < (int)word.size(); ++i)
        {
            if(v->links[word[i] - 'a'] == NULL)
                // jeśli nie ma jeszcze krawędzi z daną literą
                // to utwórz nowego potomka
                v->links[word[i] - 'a'] = new TrieNode(v, word[i]);
            v = v->links[word[i] - 'a'];
        }
        v->in_dict = true;
    }

    bool search(string &word)
    {
        TrieNode *v = root;   // wskaźnik na aktualny węzeł
        for(int i = 0; i < (int)word.size(); ++i)
        {
            if(v->links[word[i] - 'a'] == NULL)
                   // jeśli nie ma krawędzi z daną literą to zwróć fałsz
                return false;
            else   // jeśli jest to przejdź po niej
                v = v->links[word[i] - 'a'];
        }
        return v->in_dict;
    }

    TrieNode *root;
};
```

Ćwiczenie 25. Uzupełnij powyższą implementację o brakującą metodę `erase` oraz destruktory zwalniające zaalokowaną pamięć.

Przy założeniu, że nasz alfabet jest 26-elementowy, możemy przyjąć, że wszystkie operacje na zbiorze `links` wykonywane są w czasie stałym. Przy takim założeniu, złożoność wszystkich operacji na drzewie TRIE jest liniowa względem długości wstawianego (lub usuwanego czy szukanego) słowa.

Jeżeli nasz alfabet byłby duży i miał rozmiar k to złożoności poszczególnych operacji na słowie o długości n zależałyby od implementacji:

	utworzenie węzła	insert	search	erase
TRIE z mapami	$O(1)$	$O(n \log k)$	$O(n \log k)$	$O(n \log k)$
TRIE z tablicami	$O(k)$	$O(n \cdot k)$	$O(n)$	$O(n)$

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

11 Algorytm Aho-Corasick

Działanie algorytmu Aho-Corasick (w skrócie AC) jest analogiczne do algorytmu KMP. Dlatego też potrzebować będziemy odpowiednika tablicy prefiksowej w postaci drzewa TRIE, do którego początkowo wstawimy wszystkie wzorce.

W każdym węźle naszego drzewa chcemy pamiętać dodatkową wartość `suf_link`. Jeśli węzeł x reprezentuje słowo s to chcemy aby `x.suf_link` był wskaźnikiem do węzła reprezentującego najdłuższy właściwy sufiks słowa s, który jest jednocześnie prefiksem pewnego wzorca. Różnica względem algorytmu KMP jest taka, że sufiks ten nie musi być prefiksem tego samego wzorca (nie musi być jego prefikso-sufiksem).

Wartości pól `suf_link` obliczymy podobnie jak w algorytmie KMP, przy czym drzewo przechodzić będziemy algorytmem BFS (można tutaj równie dobrze użyć algorytmu DFS):

```
void create_links(Trie &trie)
{
    // przechodzimy drzewo algorytmem BFS
    queue<TrieNode*> Q;

    // wstawiamy korzeń do kolejki
    Q.push(trie.root);
    trie.root->suf_link = trie.root->dict_link = NULL;

    while(!Q.empty())
    {
        TrieNode *v = Q.front();
        Q.pop();
        if(v != trie.root)    // węzeł nie jest korzeniem
        {
            // postępujemy tak jak w KMP - cofamy się po kolejnych suf_linkach
            // dopóki nie natrafimy na krawędź o odpowiedniej etykiecie
            TrieNode *j = v->parent->suf_link;
            while(j && j->links[v->last - 'a'] == NULL)
                j = j->suf_link;
            if(j == NULL)
                v->suf_link = trie.root;
            else v->suf_link = j->links[v->last - 'a'];

            if(v->suf_link->in_dict)         // ustawiamy dict_link
                v->dict_link = v->suf_link;
            else v->dict_link = v->suf_link->dict_link;
        }

        // wstawiamy wszystkich potomków v do kolejki
        for(int i = 0; i < ALPH; ++i)
            if(v->links[i] != NULL)
                Q.push(v->links[i]);
    }
}
```

W powyższym kodzie, dla każdego węzła v ustawiana jest dodatkowo wartość `v->dict_link`, która jest wskaźnikiem do najdłuższego właściwego sufiksu węzła v, który jest jednym z wzorców.

Może się bowiem zdarzyć, że w jednym miejscu tekstu kończy się kilka dopasowań wzorców (na przykład wyszukiwanie wzorców ze zbioru {„abc”, „bc”, „c”} w tekście „abcd”). W takim przypadku, wskaźnik `dict_link` będzie potrzebny do odtworzenia wszystkich dopasowanych wzorców.

Zasada działania algorytmu AC jest taka, jak algorytmu KMP:

```
void aho_corasick(Trie &trie, string &text)
{
    TrieNode *v = trie.root, *tmp;
    for(int i = 0; i < (int)text.size(); ++i)
    {
        while(v && v->links[text[i] - 'a'] == NULL)
            v = v->suf_link;
        if(v == NULL)
            v = trie.root;
        else v = v->links[text[i] - 'a'];
        // sprawdzamy, czy są jakieś dopasowania
        if(v->in_dict)
            tmp = v;
        else tmp = v->dict_link;
        while(tmp)
        {
            printf("Znaleziono wystapienie wzorca numer %d.\n", tmp->id);
            tmp = tmp->dict_link;
        }
    }
}
```

Jeżeli po przetworzeniu kolejnej litery z tekstu znajdziemy się w węźle drzewa TRIE różnym od korzenia, musimy sprawdzić, czy ten węzeł nie reprezentuje wzorca, a także wypisać wszystkie dopasowania osiągalne poprzez `dict_link`.

Ćwiczenie 26. Ważnym przypadkiem szczególnym jest sytuacja, w której wszystkie wzorce są jednakowej długości. Zakładając, że takie właśnie są dane wejściowe, zaimplementuj alternatywną wersję algorytmu AC, nie korzystającą z `dict_link`ów.

Jaką złożoność ma algorytm Aho-Corasick? Wstawienie wzorców do drzewa odbywa się w czasie liniowym względem sumy ich długości. Funkcja `create_links` również działa w czasie liniowym względem sumy długości wzorców, co można uzasadnić analogicznie jak w oszacowaniu złożoności algorytmu KMP. Wreszcie funkcja `aho_corasick`, która, pomijając przeglądanie wskaźników `dict_link`, działa liniowo względem sumy długości wzorców i tekstu. Może się jednak zdarzyć, że ilość dopasowań wzorców będzie rzędu kwadratowego (na przykład wyszukiwanie wzorców ze zbioru {„a”, „aa”, „aaa”, „aaaa”} w tekście „aaaa”. A zatem złożoność algorytmu AC to $O(n+S_w+d)$, gdzie n jest długością tekstu, S_w — sumą długości wzorców a d — sumą wystąpień wzorców w tekście.

11.1 Algorytm Bakera

Ważnym zastosowaniem algorytmu AC jest algorytm Bakera, który służy do wyszukiwania dwuwymiarowego wzorca w dwuwymiarowym tekście.

Załóżmy, że nasz wzorzec jest macierzą o w_y wierszach i w_x kolumnach oraz elementach należących do alfabetu. Analogicznie tekst jest macierzą o t_y wierszach i t_x kolumnach.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Algorytm Bakera działa w następujący sposób:
Wyszukujemy kolumny wzorca w kolumnach tekstu za pomocą algorytmu AC. Znalezione dopasowania zapisujemy w dodatkowej tablicy dwuwymiarowej `hits[`t_x`][`t_y`]`. Kolumnom wzorca nadajemy identyfikatory będące dodatnimi liczbami naturalnymi, przy czym istotne jest, aby przystające kolumny otrzymały ten sam identyfikator.

Przykład:

wzorzec:	aba bab	tekst:	abad baba cbab	tablica `hits`:	1210 0121 0000
identyfikatory kolumn:	121				

Liczba znajdująca się na pozycji `hits[i][j]` jest identyfikatorem kolumny wzorca, która występuje w tekście w kolumnie i-tej, w wierszach od j do $j+w_y-1$. Liczba 0 oznacza brak dopasowania.

Mając wyznaczoną tablicę `hits` wystarczy, że za pomocą algorytmu KMP znajdziemy wszystkie wystąpienia słowa złożonego z kolejnych identyfikatorów kolumn wzorca (na powyższym przykładzie "121") w wierszach tablicy `hits`. Warto zauważyć, że ostatnie w_y-1 wierszy tablicy `hits` jest zawsze wypełnionych zerami i można je zignorować.

Ćwiczenie 27. Zaimplementuj algorytm Bakera.

Jako, że wszystkie wzorce wyszukiwane algorytmem AC są równej długości, złożoność algorytmu Bakera jest liniowa względem sumy wielkości tekstu i wzorca.

12 Maski bitowe

Binarna reprezentacja zbiorów

Wiemy, że dane w pamięci komputera zapisane są w postaci kodu binarnego. Okazuje się, że możemy to wykorzystać w celu wygodnego przechowywania i przetwarzania informacji o zbiorach.

Zajmiemy się 32-bitowym typami całkowitymi dodatnimi np. `unsigned int`. Liczba 75 w pamięci ma postać:

numer bitu	...	7	6	5	4	3	2	1	0
wartość	...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
wartość bitu	...	0	1	0	0	1	0	1	1

Definicja 6. Maska bitowa to forma kodowania informacji o całym zbiorze w jednej zmiennej.
Ustawienie bitu o numerze i na 1 lub 0 (mówimy czasem o „zapaleniu bitu") informuje o występowaniu lub braku elementu i w zbiorze.

Przykładem może być przechowywanie informacji o podzbiorach zbioru liter alfabetu angielskiego. Literom od a do h nadajemy numerki od 0 do 7. Zbiorowi $\{a,b,d,h\}$ odpowiada zatem maska $10001011_2 = 2^7+2^3+2^1+2^0 = 139$.

Ćwiczenie 28. Znajdź maskę dla zbioru $\{a,d,f,g\}$ i zbiór reprezentowany przez maskę 83.

Jak w prosty sposób budować maski bitowe? Służą do tego operatory bitowe, m.in:

- `a << b` przesuwa bity zmiennej a o b bitów w lewo (z prawej dopełniając zerami). Maska reprezentująca $\{a,g,h\}$ to zatem `(1<<0) + (1<<6) + (1<<7)` (nawiasy są istotne, operacje bitowe mają niski priorytet).
  Przykład: 10011011_2 `<<` $2 = 01101100_2$

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

- `a >> b` przesuwa bity zmiennej a o b bitów w prawo (z lewej dopełniając zerami). Zachowuje się tak samo jak dzielenie (całkowitoliczbowe) przez 2^b.
  Przykład: 10011011_2 `>>` $2 = 00100110_2$

- `~a` neguje, zamienia wszystkich bitów na przeciwne. Ze zbioru robi jego dopełnienie.
  Przykład: $10011011_2 = 01100100_2$

Operatory bitowe

Działanie operatorów bitowych na pojedynczych bitach (i odpowiadające im nazwy z logiki):

Zmienne		Koniunkcja	Alternatywa	Alternatywa wykluczająca
`a`	`b`	`a & b`	`a | b`	`a ^ b`
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Wszelkie operatory bitowe są wykonywane niezwykle szybko, mniej więcej tak, jak dodawanie dwóch liczb. Ich działanie na całych zmiennych polega na wykonaniu powyższych operacji na każdej parze bitów. Możemy więc szybko wykonywać operacje na zbiorach, dzięki następującym operatorom:

- **Koniunkcja:** `a & b`
  Przykład: 10011011_2 `&` $11001111_2 = 10001011_2$
  Koniunkcja jest prawdziwa tylko jeżeli oba argumenty są prawdziwe. W związku z tym zastosowana do dwóch zbiorów daje ich część wspólną, bo zachowane są te bity, które były zapalone w obu maskach.
  Trik: `(a & (1<<nr)) != 0` sprawdza czy bit o numerze nr jest jedynką

- **Alternatywa:** `a | b`
  Przykład: 10011011_2 `|` $11001111_2 = 11011111_2$
  Alternatywa sprawdza czy choć jeden z jej argumentów jest równy 1. Zatem wynikiem alternatywy dwóch masek jest ich suma (teoriomnogościowa).

- **Alternatywa wykluczająca:** `a ^ b`
  Przykład: 10011011_2 `^` $11001111_2 = 01010100_2$
  Operację tę często nazywamy zaczerpniętym z angielskiego skrótem `xor`. Sprawdza ona, czy dokładnie jeden z jej argumentów jest równy 1.
  Trik: `(a ^ b) & a` to maska będąca różnicą zbiorów reprezentowanych przez maski a i b.

Ćwiczenie 29. Mając dane maski reprezentujące zbiory a, b i c znajdź za pomocą powyższych operatorów:

- część wspólną tych trzech zbiorów
- zbiór elementów, które występują w dokładnie dwóch spośród zbiorów a, b, c
- różnicę zbiorów a i b, bez użycia operatora `^`

Wszystkie podzbiory

Maski bitowe bardzo zgrabnie wykorzystujemy w rozwiązaniach polegających na przejrzeniu wszyst kich podzbiorów. Przykładowy problem:

Mamy dane liczby $a_1, a_2, \ldots, a_n$. Czy da się z nich wybrać podzbiór o sumie S?

```
bool odp = false;
/* Maski odpowiadające wszystkim podzbiorom zbioru        *
 * n-elementowego to po prostu liczby od 0 do (1 << n)-1  */
int ogr = 1 << n;
for (int maska = 0; maska < ogr; maska++)
{
  int suma = 0;
  /* sprawdzamy, które elementy są w sprawdzanym podzbiorze */
  for (int i = 0; i < n; i++)
    if ((maska & (1<<i)) != 0)
      suma += a[i];
  if (suma == S)
  {
    odp = true;
    break;
  }
}
/* Wynik: w zmiennej odp */
```

Ćwiczenie 30. Określ złożoność powyższego rozwiązania.

Ćwiczenie 31. *(Ambitne)* Masz daną maskę bitową reprezentującą pewien zbiór. W jaki prosty sposób możesz wygenerować wszystkie podzbiory tego zbioru?
Wskazówka: Należy je generować w odwrotnej kolejności niż w powyższym programie.

12.1 Programowanie dynamiczne na maskach

Czasami maski bitowe stają się stanami w rozwiązaniach opartych na programowaniu dynamicznym. Dzieje się tak gdy musimy zapamiętać nie tylko ile obiektów już wykorzystaliśmy, ale też dokładnie które z nich. Przyjrzyjmy się następującemu problemowi:

Chcemy ułożyć w szereg n przedszkolaków. Henio i Kazio uwielbiają stać koło siebie, ale niestety Adaś i Michaś są wyraźnie wrogo do siebie nastawieni. Ogólniej, dla każdej pary przedszkolaków wiemy, jak bardzo będą zadowoleni z bycia sąsiadami (dla dzieci o numerach i i j liczbę tę oznaczmy przez $w_{ij} \geqslant 0$). Należy wyznaczyć maksymalną sumę zadowolenia, jaka może być osiągnięta.

Można ten problem rozwiązać sprawdzając wszystkie możliwe permutacje przedszkolaków, w czasie $O(n \cdot n!)$. Istnieje jednak szybszy algorytm z wykorzystaniem programowania dynamicznego. Polega na rozwiązaniu wszystkich możliwych podproblemów, tzn. znalezienia maksymalnego zadowolenia każdego podzbioru dzieci, z ustalonym pewnym przedszkolakiem na końcu szeregu. Wyniki tych podproblemów będziemy przechowywać w tablicy:

- `t[mask][a]` - maksymalne zadowolenie zbioru przedszkolaków (przechowywanego w masce `mask`) w ustawieniu gdzie dziecko o numerze a jest na końcu.

Zauważmy, że jeżeli maski będziemy przeglądać po prostu w kolejności rosnącej, to w momencie obliczania wyniku dla pewnego zbioru, wszystkie jego podzbiory będą już przetworzone (będziemy znali ich wyniki).

```
for (int maska = 1; maska < (1 << n); maska++)
  for (int i = 0, a = 1; i < n; i++, a <<= 1)
    /* ustalamy ostatnie dziecko z danego zbioru */
    if (a & maska)
    {
      if ((a ^ maska) == 0)
      {
        /* przypadek samotnego dziecka */
```

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

```
        t[maska][i] = 0;
        continue;
      }
      int nw = 0;
      for (int j = 0, b = 1; j < n; j++, b <<= 1)
      {
        /* sprawdzamy wszystkie możliwe przedostatnie przedszkolaki, *
         * różne od ostatniego i będące w przetwarzanym zbiorze      */
        if (j != i && (b & maska))
          /* t[maska ^ a][j] to wynik dla zbioru bez ostatniego dziecka *
           * z j-tym dzieckiem na końcu                                 */
          if (t[maska ^ a][j] + w[i][j] > nw)
            nw = t[maska ^ a][j] + w[i][j];
      }
      t[maska][i] = nw;
    }
```

Ćwiczenie 32. Jak z wypełnionej tablicy `t` uzyskać wynik dla całego zadania?

Złożoność tego algorytmu to $O(2^n \cdot n^2)$. Jest to wynik istotnie lepszy niż $O(n \cdot n!)$. Przykładowo, dla $n = 15$ nasz algorytm potrzebuje około 7 milionów operacji dominujących, a rozwiązanie „brutalne" ponad 10 *bilionów*!

Ćwiczenie 33. Rozszerz powyższy algorytm by otrzymać nie tylko wartość, ale także optymalne rozmieszczenie dzieci.

12.2 Meet in the middle

Meet in the middle, dosłownie „spotkajmy się w połowie", nie ma póki co ogólnie przyjętej polskiej nazwy. Zachęcam do wymyślenia czegoś!

Technika ta to próba przyśpieszenia rozwiązań brutalnych: dzielimy wejściowy zbiór na dwa, w każdym z nich robimy pełne przeszukiwanie, a następnie staramy się złączyć wyniki. Spróbujemy teraz ponownie zmierzyć się z problemem szukania podzbioru o określonej sumie.

Zastosowanie *meet in the middle* wygląda następująco: dzielimy zbiór na dwa, w każdym z nich generujemy wszystkie możliwe sumy podzbiorów, sortujemy i próbujemy dobrać pary liczb po jednej z każdej posortowanej listy, które mają odpowiednią sumę.

Ćwiczenie 34. Jak zrealizować wyszukiwanie binarne w czasie liniowym względem długości list?

Jak wygląda kwestia złożoności? Zamiast wyjściowego $O(2^n \cdot n)$ otrzymujemy $O(2^{\frac{n}{2}} \cdot n)$. Zatem asymptotycznie udało nam się zmniejszyć stałą w wykładniku dwa razy, co jest niezłym wynikiem.

Ćwiczenie 35. Dokonaj własnoręcznie szacowań złożoności tego algorytmu. Nie zapomnij, że konieczne jest sortowanie!

Zadania

A. **Suma podzbioru**

Zaimplementuj program ilustrujący technikę „meet in the middle": masz stwierdzić czy ze zbioru $a_1, a_2, \ldots, a_n$ da się wybrać podzbiór o sumie S.

Wejście:
n S
a_1 a_2 ... a_n

Wyjście:
TAK, lub *NIE*.

B. **Szczególna wycieczka**
Mamy dany ważony, skierowany graf G o n wierzchołkach i m krawędziach. Podróżujemy w nim między wierzchołkami o numerze 1 i n. W grafie jest też wyróżnione k wierzchołków (o numerach różnych od 1 i n), przez które musimy przejechać *dokładnie jeden raz*. Znajdź minimalny czas potrzebny na przebycie trasy!

Wejście:
nm
$p_1\ k_1\ w_1$
$p_2\ k_2\ w_2$
...
$p_n\ k_n\ w_n$
Trójka $p_i,\ k_i,\ w_i$ oznacza, że między wierzchołkami p_i oraz k_i istnieje krawędź o wadze w_i.

Wyjście:
Pojedyncza liczba - długość najkrótszej możliwej trasy.

Literatura

1. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Wprowadzenie do algorytmów*, WNT, Warszawa 2004
2. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, WNT, Warszawa 2003
3. Diks K., Malinowski A., Rytter W. Waleń T., *Algorytmy i struktury danych*, `http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych`
4. *Standard Template Library Programmer's Guide*, `http://www.sgi.com/tech/stl/`
5. *Młodzieżowa Akademia Informatyki*, `http://main.edu.pl/`

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

W projekcie **Informatyka +**, poza wykładami i warsztatami, przewidziano następujące działania:

- 24-godzinne kursy dla uczniów w ramach modułów tematycznych
- 24-godzinne kursy metodyczne dla nauczycieli, przygotowujące do pracy z uczniem zdolnym
- nagrania 60 wykładów informatycznych, prowadzonych przez wybitnych specjalistów i nauczycieli akademickich
- konkursy dla uczniów, trzy w ciągu roku
- udział uczniów w pracach kół naukowych
- udział uczniów w konferencjach naukowych
- obozy wypoczynkowo-naukowe.

Szczegółowe informacje znajdują się na stronie projektu

www.informatykaplus.edu.pl

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego