**SORTOWANIA**

**Proste**

* **Bubble sort**

void bsort(int t[], int n) {

bool test;

for (int i = 0; i<n; i++) {

test = true;

for (int j = 0; j<n - 1 - i; j++) {

if (t[j]>t[j + 1]) {

swap(t[j], t[j + 1]);

test = false;

}

}

if (test) break;

}

}

* **Insertion sort**
* **Na tablicy**

void isortTab(int A[], int n) {

for (int j = 1; j < n; j++) {

int key = A[j];

int i = j - 1;

while (i >= 0 && A[i]>key) {

A[i + 1] = A[i];

i--;

}

A[i + 1] = key;

}

}

* **Na liście**

void insert(node \* list, node \* el) {

while (list->next != NULL && list->next->key < el->key)

list = list->next;

el->next = list->next;

list->next = el;

}

node \* isort(node \* first) {

node \* sorted = new node;

sorted->next = NULL;

while (first->next != NULL) {

node \* tmp = first->next;

first->next = tmp->next;

tmp->next = NULL;

insert(sorted, tmp);

}

delete first;

return sorted;

}

* **Selection sort**
* **Na tablicy**

void selectionSort(int A[], int n) {

for (int i = 1; i<n; i++) {

int min = i - 1;

for (int j = i; j<n; j++)

if (A[j] < A[min]) min = j;

swap(A[min], A[i - 1]);

}

}

* **Na liście**

// lista nie może być pusta

node \* removemax(node \* first) {

node \* p\_max = first;

while (first->next != NULL) {

if (p\_max->next->key < first->next->key)

p\_max = first;

first = first->next;

}

node \* max = p\_max->next;

p\_max->next = max->next;

max->next = NULL;

return max;

}

node \* ssort(node \* first) {

node \* sorted = new node;

sorted->next = NULL;

while (first->next != NULL) {

node \* tmp = removemax(first);

tmp->next = sorted->next;

sorted->next = tmp;

}

delete first;

return sorted;

}

**Szybkie**

* **Mergesort**
* **Na tablicy**

void mergeTab(int tab[], int l1, int m, int r2, int N) {

int p = l1;

int \* T2 = new int[N];

int r1 = m;

int l2 = m + 1;

int i = l1;

while (l1 <= r1 && l2 <= r2) {

if (tab[l1] < tab[l2])

T2[i] = tab[l1++];

else

T2[i] = tab[l2++];

i++;

}

while (l1 <= r1)

T2[i++] = tab[l1++];

while (l2 <= r2)

T2[i++] = tab[l2++];

for (i = p; i <= r2; i++)

tab[i] = T2[i];

}

void msortTab(int tab[], int l, int r, int n) {

if (l < r) {

int m = (l + r) / 2;

msortTab(tab, l, m, n);

msortTab(tab, m + 1, r, n);

mergeTab(tab, l, m, r, n);

}

}

* **Na liście**

// wszystkie listy mają wartownika

// l nie może być nullem

void nserie(node \* l, node \* s) {

node \* tmp = l->next;

while (tmp->next != NULL && tmp->key < tmp->next->key)

tmp = tmp->next;

s->next = l->next;

l->next = tmp->next;

tmp->next = NULL;

}

node \* merge\_series(node \* s1, node \* s2, node \* append) {

node \* end = append;

while (s1->next != NULL && s2->next != NULL) {

if (s1->next->key <= s2->next->key) {

end->next = s1->next;

end = end->next;

s1->next = s1->next->next;

end->next = NULL;

}

else {

end->next = s2->next;

end = end->next;

s2->next = s2->next->next;

end->next = NULL;

}

}

if (s1->next != NULL) {

end->next = s1->next;

s1->next = NULL;

}

else if (s2->next != NULL) {

end->next = s2->next;

s2->next = NULL;

}

while (end->next != NULL) {

end = end->next;

}

return end;

}

void mergesort(node \* head) {

node \* s1 = new node;

s1->next = NULL;

node \* s2 = new node;

s2->next = NULL;

node \* l = new node;

l->next = NULL;

node \* end = l;

int n;

do {

n = 0;

while (head->next != NULL) {

nserie(head, s1);

n++;

if (head->next != NULL) {

nserie(head, s2);

n++;

end = merge\_series(s1, s2, end);

}

else {

end->next = s1->next;

s1->next = NULL;

}

}

head->next = l->next;

l->next = NULL;

end = l;

} while (n > 2);

delete l;

delete s1;

delete s2;

}

* **Heapsort**

int left(int i) { return 2 \* i + 1; }

int right(int i) { return 2 \* i + 2; }

int parent(int i) { return (i - 1) / 2; }

void heapify(int A[], int i, int hs) {

int ind\_max = i;

if (left(i) < hs && A[left(i)] > A[ind\_max])

ind\_max = left(i);

if (right(i) < hs && A[right(i)] > A[ind\_max])

ind\_max = right(i);

if (i != ind\_max) {

swap(A[i], A[ind\_max]);

heapify(A, ind\_max, hs);

}

}

void buildheap(int A[], int n) {

for (int i = parent(n - 1); i >= 0; i--)

heapify(A, i, n);

}

void heapsort(int A[], int n) {

buildheap(A, n);

for (int i = n - 1; i>0; i--) {

swap(A[0], A[i]);

heapify(A, 0, i);

}

}

* **Quicksort**
* **Horae**

int partition(int A[], int l, int r) {

int x = A[l];

int i = l - 1;

int j = r + 1;

while (true) {

do j--; while (A[j] > x);

do i++; while (A[i] < x);

if (i < j)

swap(A[i], A[j]);

else

return j;

}

}

int partition\_rand(int A[], int l, int r) {

int q = (rand() % (r - l + 1)) + l;

swap(A[l], A[q]);

return partition(A, l, r);

}

void quicksort\_horae(int A[], int l, int r) {

if (p < r) {

int q = partition\_rand(A, l, r);

quicksort\_horae(A, l, q);

quicksort\_horae(A, q + 1, r);

}

}

* **Lomuto**

int partition(int A[], int l, int r) {

int x = A[r];

int i = l - 1;

int j;

for (j = l; j < r; j++) {

if (A[j] <= x) {

i++;

swap(A[i], A[j]);

}

}

swap(A[r], A[i + 1]);

return i + 1;

}

void quicksort\_lomuto(int A[], int l, int r) {

if (l < r) {

int q = partition2(A, l, r);

quicksort\_lomuto (A, l, q - 1);

quicksort\_lomuto(A, q + 1, r);

}

}

* **Derekursywacja**

void qsort\_stack(int tab[], int N) {

stack < int > Q;

Q.push(0);

Q.push(N - 1);

while (!Q.empty()) {

int r = Q.top(); Q.pop();

int l = Q.top(); Q.pop();

int i = partition(tab, l, r); // from horae

if (l < i) {

Q.push(l);

Q.push(i);

}

if (i + 1 < r) {

Q.push(i + 1);

Q.push(r);

}

}

}

**Liniowe**

* **Counting sort**

void countingSort(int A[], int n) {

int \* B = new int[n];

int max = A[0], min = A[0];

for (int i = 1; i < n; i++) {

if (A[i] > max)

max = A[i];

if (A[i] < min)

min = A[i];

}

int k = (max - min) + 1;

int \* C = new int[k];

for (int i = 0; i < k; i++)

C[i] = 0;

for (int i = 0; i < n; i++)

C[A[i]-min]++;

for (int i = 0; i < k - 1; i++)

C[i + 1] += C[i];

for (int i = 0; i < n; i++)

B[--C[A[i]-min]] = A[i];

for (int i = 0; i < n; i++)

A[i] = B[i];

delete[] B;

delete[] C;

}

* **Radix sort**

/\*

Mamy tablicę A n-elementową, zawierającą elementy z przedziału [0, n^2-1].

Należy ją posortować.

Rozwiązanie:

System n-kowy

Liczby z zakresu: [0, n^2-1]

można zapisać w postaci: An + B, gdzie A, B należą do przedziału [0, n-1]

Należy posortować pozycyjnie z użyciem sortowania przez zliczanie.

Czas sortowania: O(n)

\*/

// zwraca cyfrę o pozycji 1...numlen(num) licząc od tyłu

int getNumByPos(int num, int base, int pos) {

if (pos == 1) return num%base;

int i = 1;

num /= base;

int dig = 0;

while (num != 0 && i != pos) {

dig = num%base;

num /= base;

i++;

}

return dig;

}

// sortuje względem pozycji p

void posSort(int A[], int n, int p) {

int \* res = new int[n];

int \* tab = new int[n];

for (int i = 0; i < n; i++)

tab[i] = 0;

for (int i = 0; i < n; i++)

tab[getNumByPos(A[i], n, p)]++;

for (int i = 1; i < n; i++)

tab[i] = tab[i] + tab[i - 1];

for (int i = n - 1; i >= 0; i--)

res[--tab[getNumByPos(A[i], n, p)]] = A[i];

for (int i = 0; i < n; i++)

A[i] = res[i];

delete[] res;

delete[] tab;

}

// Sortuje n elementową tablicę z elementami z zakresu [0, n^p-1]

void sort(int A [], int n, int p) {

for (int i = 1; i <= p; i++)

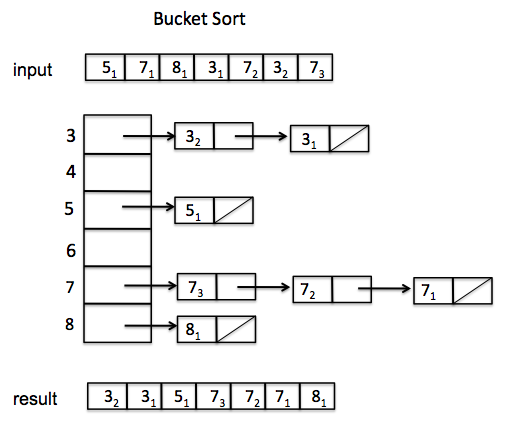
posSort(A, n, i);

}

* **Bucket sort**
* **Opis**

Zubełkowego pozwala sortować zbiory liczb całkowitych - najlepiej o dużej ilości elementów, lecz o małym zakresie wartości. Zasada działania jest następująca:

1. Określamy zakres wartości, jakie mogą przyjmować elementy sortowanego zbioru. Niech wmin oznacza najmniejszą wartość, a wmax niech oznacza wartość największą.
2. Dla każdej możliwej wartości przygotowujemy kubełek-licznik, który będzie zliczał ilość wystąpień tej wartości w sortowanym zbiorze. Liczba liczników jest równa (wmax - wmin + 1). Każdy licznik jest początkowo ustawiony na wartość zero.
3. Przeglądamy kolejno elementy zbioru od pierwszego do ostatniego. Dla każdego elementu zbioru zwiększamy o jeden zawartość licznika o numerze równym wartości elementu. Na przykład, jeśli kolejny element zbioru ma wartość 3, to zwiększamy licznik o numerze 3. W efekcie po przeglądnięciu wszystkich elementów zbioru liczniki będą zawierały ilość wystąpień każdej z możliwych wartości. Jeśli dany licznik zawiera 0, to wartość równa numerowi licznika w zbiorze nie występuje. Inaczej wartość ta występuje tyle razy, ile wynosi zawartość jej licznika.
4. Przeglądamy kolejne liczniki zapisując do zbioru wynikowego ich numery tyle razy, ile wynosi ich zawartość. Zbiór wyjściowy będzie posortowany.



**Abstrakcyjne struktury danych**

**Drzewa binarne**

* **Drzewa BST**
* **Opis**

Drzewo poszukiwań binarnych (ang. Binary Search Tree) jest drzewem binarnym, w którym każdy węzeł spełnia reguły:

1. Jeśli węzeł posiada lewe poddrzewo (drzewo, którego korzeniem jest lewy syn), to wszystkie węzły w tym poddrzewie mają wartość niewiększą od wartości danego węzła.
2. Jeśli węzeł posiada prawe poddrzewo, to wszystkie węzły w tym poddrzewie są niemniejsze od wartości danego węzła.

* **Wygląd struktury**

struct BSTNode {

BSTNode \* up;

BSTNode \* left;

BSTNode \* right;

int key;

};

* **Wypisywanie**

void inorder(BSTNode \* v) {

if (v != NULL) {

inorder(v->left);

cout << v->key << " ";

inorder(v->right);

}

}

* **Sprawdzanie czy element istnieje**

BSTNode \* find(BSTNode \* root, int key) {

if (root == NULL) return NULL;

if (root->key == key) return root;

else if (key < root->key)

return find(root->left, key);

else

return find(root->right, key);

}

* **Wstawianie**

void addNode(BSTNode \* &tree, BSTNode \* newNode) {

if (tree == NULL) {

tree = newNode;

} else {

BSTNode \* cp = tree;

while (true) {

if (newNode->key <= cp->key) {

if (cp->left == NULL)

break;

else

cp = cp->left;

} else {

if (cp->right == NULL)

break;

else

cp = cp->right;

}

}

newNode->up = cp;

if (newNode->key <= cp->key)

cp->left = newNode;

else

cp->right = newNode;

}

}

* **Znajdowanie następnika**

BSTNode \* treeSuccessor(BSTNode \* v) {

if (v->right != NULL)

return treeMin(v->right);

BSTNode \* y = v->up;

while (y != NULL && v == y->right) {

v = y;

y = y->up;

}

return y;

}

* **Znajdowanie poprzednika**

BSTNode \* treePredecessor(BSTNode \* v) {

if (v->left != NULL)

return treeMax(v->left);

BSTNode \* y = v->up;

while (y != NULL && v == y->left) {

v = y;

y = y->up;

}

return y;

}

* **Usuwanie**

BSTNode \* deleteNode(BSTNode \* &root, BSTNode \* z) {

BSTNode \* y, \* x;

if (z->left == NULL || z->right == NULL)

y = z;

else

y = treeSuccessor(z);

if (y->left != NULL)

x = y->left;

else

x = y->right;

if (x != NULL)

x->up = y->up;

if (y->up == NULL)

root = x;

else if (y == y->up->left)

y->up->left = x;

else

y->up->right = x;

if (y != z)

z->key = y->key;

return y;

}

* **Różne odmiany drzew BST**
* **Drzewa AVL**
* **Opis**

Drzewo AVL to takie drzewo BST, w którym dla każdego węzła różnica wysokości podrzew wynosi najwyżej 1.

* **Wygląd struktury**

struct AVLNode {

AVLNode \* up;

AVLNode \* left;

AVLNode \* right;

int key;

int bf; // współczynnik równowagi

};

* **Wstawianie**

1. Wstawiamy tak samo jak do drzewa BST.
2. Wracamy od wstawionego węzła w górę aktualizując współczynniki wyważenia
3. Jeśli któryś ze współczynników przyjmuje wartość 2 lub -2 to naprawiamy drzewo stosując odpowiednie rotacje

* **Drzewa Czerwono-Czarne**
* **Opis**

1. Każdy węzeł jest albo czerwony albo czarny.
2. Korzeń jest czarny
3. Każdy liść (NULL) jest czarny
4. Jeśli węzeł jest czerwony, to jego dzieci są czarne
5. Dla każdego węzła wewnętrznego każda ścieżka prosta do liścia zawiera tyle samo czarnych węzłów.

* **Wygląd struktury**

struct RBTNode {

RBTNode \* up;

RBTNode \* left;

RBTNode \* right;

int key;

char color;

};

* **Wstawianie**

1. Wstawiamy jak do zwykłego drzewa BST, kolorujemy węzeł na czerwono
2. Wędrujemy w górę drzewa naprawiając węzły
3. Korzeń ustawiamy na czarny

* **Drzewa Splay**
* **Opis**

Nazwa drzew pochodzi od operacji splay.

* **Funkcja splay**

Operacja splay(x) wyciąga węzeł o wartości x (lub najbliższy) do korzenia drzewa.

1. Znajdź węzeł z x tak jak w zwykłym drzewie BST
2. Przy pomocy serii operacji Zig, Zig-Zig, Zig-Zag wyciągnij x do korzenia

Zasada Zig:

Jeśli ojciec *y* węzła *x* jest korzeniem drzewa T, to wykonujemy rotację pojedynczą, w której uczestniczą ojciec *y* oraz węzeł *x*. Rotacja taka kończy operację splay.

Zasada Zig-Zig:

Jeśli ojciec *y* węzła *x* nie jest korzeniem, a zarówno *y* jak i *x* są jednocześnie prawymi lub jednocześnie lewymi synami, to wykonujemy dwie rotacje: najpierw rotacja z dziadkiem *z* i ojcem *y*, a następnie rotacja z ojcem *y* i węzłem *x*.

Zasada Zig-Zag:

Jeśli ojciec *y* nie jest korzeniem, a węzły *x* i *y* są naprzemiennie lewym i prawym dzieckiem, to najpierw wykonujemy rotację węzła *y* z węzłem *x*, a następnie rotację nowego ojca x (jest to teraz węzeł *z*)  z węzłem *y*.

* **Dodawanie**

void insertSplay(BSTNode \* & root, int k) {

BSTNode \* x = new BSTNode; // Tworzymy nowy węzeł

x->left = x->right = NULL; // Ustawiamy pola nowego węzła

x->key = k;

if (!root) { // Jeśli drzewo jest puste,

x->up = NULL; // to węzeł x staje się korzeniem

root = x;

} else {

splay(root, k); // W korzeniu pojawia się następnik lub poprzednik

x->up = root; // Będzie on zawsze ojcem węzła x

if (k < root->key) { // Wybieramy miejsce dla x

x->left = root->left;

root->left = x; // x staje się lewym synem korzenia

if (x->left) x->left->up = x;

} else {

x->right = root->right;

root->right = x; // x staje się prawym synem korzenia

if (x->right) x->right->up = x;

}

}

}

* **Usuwanie**

void removeSplay(BSTNode \* & root, int k) {

BSTNode \*TL, \*TR;

if (root) {

splay(root, k); // Usuwany węzeł idzie do korzenia

if (root->key == k) { // Sprawdzamy, czy rzeczywiście tam trafił

TL = root->left; // Zapamiętujemy synów węzła

TR = root->right;

delete root; // Węzeł usuwamy z pamięci

root = NULL; // Teraz drzewo jest puste

if (TL) { // Wybieramy niepuste poddrzewo

TL->up = NULL; // Teraz TL wskazuje korzeń

splay(TL, k); // Do korzenia trafia poprzednik usuniętego węzła

while (TL->right) rot\_L(TL, TL); // idziemy na skraj drzewa

TL->right = TR; // TR staje się prawym synem

if (TR) TR->up = TL;

root = TL; // Uaktualniamy korzeń

} else if (TR) { // Przypadek symetryczny dla TR

TR->up = NULL; // Teraz TR wskazuje korzeń

splay(TR, k); // Do korzenia trafia następnik usuniętego węzła

while (TR->left) rot\_R(TR, TR); // idziemy na skraj drzewa

TR->left = TL; // TL staje się lewym synem

if (TL) TL->up = TR;

root = TR; // Uaktualniamy korzeń

}

}

}

}

* **Różne zadania z drzewami BST**
* **Zwróć n-ty węzeł co do wartości**

int getNElement(BSTNode \* v, int i) {

if (v == NULL) return -1;

int lsum;

if (v->left == NULL)

lsum = 0;

else

lsum = v->sum;

if (lsum + 1 == i)

return v->key;

if (lsum >= i)

return getNElement(v->left, i);

return getNElement(v->right, i - lsum - 1);

}

* **Mając drzewo RBT napisz program zwracający sumę wartości węzłow z przedziału [x, y]**

struct Brnode {

int key, sumL, // suma lewego poddrzewa

sumR; // suma prawego poddrzewa

char \* color;

BRnode \* left, \*right;

BRNode() { this->sumL = this->sumR = 0; }

};

// O(2logn) = O(logn)

int sum(BRnode \* T, int x, int y) {

int sum = T->key + T->sumL + T->sumR; // całkowita suma

BRnode \* cp = T;

while (cp != NULL) { // idziemy w lewo

if (x < cp->key) {

cp = cp->left;

} else if (x > cp->key) {

sum -= cp->key + cp->sumL;

cp = cp->right;

} else {

sum -= cp->sumL;

break;

}

}

BRnode \* cp = T;

while (cp != NULL) { // idziemy w prawo

if (y < cp->key) {

sum -= cp->key + cp->sumR;

cp = cp->left;

} else if (y > cp->key) {

cp = cp->right;

} else {

sum -= cp->sumR;

break;

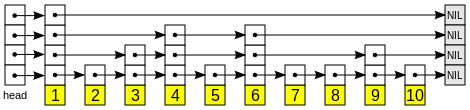
}

}

}

**Skip Lista**

* **Opis**

****

Probabilistyczna [struktura danych](https://pl.wikipedia.org/wiki/Struktura_danych) przeznaczona do przechowywania danych uporządkowanych (np. posortowanych rosnąco liczb), będąca rozwinięciem [listy jednokierunkowej](https://pl.wikipedia.org/wiki/Lista), a stanowiąca alternatywę dla drzew zbalansowanych (wyważonych), takich jak [drzewa AVL](https://pl.wikipedia.org/wiki/Drzewo_AVL), czy [czerwono-czarne](https://pl.wikipedia.org/wiki/Drzewo_czerwono-czarne).

Oczekiwana [złożoność](https://pl.wikipedia.org/wiki/Z%C5%82o%C5%BCono%C5%9B%C4%87_obliczeniowa) operacji wyszukiwania, wstawiania nowego elementu do listy oraz usunięcia elementu wynosi **O(logn).**

* **Wygląd struktury**

struct SLNode {

int val;

SLNode \*\* next;

};

struct SkipList {

SLNode \* head;

int height;

};

* **Losowanie wysokości**

int getHeight(int max\_h, double p = 1/2) {

int h = 1;

while (h < max\_h && rand48() < p)

h++;

return h;

}

* **Wyszukiwanie**

SLNode \* find(SkipList \* skip, int key) {

SLNode \* it = skip->head;

for (int i = skip->height - 1; i >= 0; i--)

while (key > it->next[i]->val)

it = it->next[i];

return (it->next[0]->val == key) ? it->next[0]->val : NULL;

}

* **Dodawanie**

void insert(SkipList \* skip, int key) {

SLNode \* it = skip->head;

int h = getHeight(it->height);

SLNode \* newNode = new SLNode;

newNode->val = key;

newNode->next = new SLNode \*[h];

for (int i = skip->height - 1; i >= 0; i--) {

while (it->next[i] != NULL && key > it->next[i]->val)

it = it->next[i];

if (i<h) {

newNode->next[i] = it->next[i];

it->next[i] = newNode;

}

}

}

* **Usuwanie**

void insert(SkipList \* skip, int key) {

SLNode \* it = skip->head;

SLNode \* c;

for (int i = skip->height - 1; i >= 0; i--) {

while (it->next[i] != NULL && key > it->next[i]->val)

it = it->next[i];

if (it->next[i]->val == key) {

c = it->next;

it->next[i] = c[i]->next[i];

}

}

if (it->next[0]->val != key) return;

delete c[0];

}

**Tablice z haszowaniem**

* **Idea**

Czy dany łańcuch możemy znaleźć wśród innych łańcuchów w czasie porównywalnym z czasem stałym **O(1)**? Rozwiązaniem jest **haszowanie** (ang. hashing). Wyobraźmy sobie, że posiadamy pewną funkcję, która dla danego łańcucha daje w wyniku liczbę całkowitą z zakresu od 0 do n-1. Tworzymy tablicę *n* elementową łańcuchów i łańcuchy umieszczamy w tej tablicy na pozycjach określonych przez naszą funkcję, którą będziemy nazywać **funkcją haszującą** (ang. hash function). Aby teraz znaleźć łańcuch w tablicy, wyznaczamy dla niego wartość funkcji haszującej i sięgamy do komórki o tym indeksie.

* **Funkcje haszujące**

// przykład implementacji funkcji haszującej

typedef unsigned int hashType;

struct Data {

char \* firstName;

char \* lastName;

int age;

hashType hash;

};

hashType getHash(Data \* data) {

int waga = 65599, sum = 0;

for (int i = 0; data->firstName[i] != 0; i++)

sum = sum \* waga + data->firstName[i];

for (int i = 0; data->lastName[i] != 0; i++)

sum = sum \* waga + data->lastName[i];

return sum\*waga + data->age;

}

* **Rodzaje rozwiązywania konfliktów**
* **Metoda listowa**

struct mydata {

char \* name, \*surname;

mydata \* next;

mydata() {

this->next = NULL;

}

};

struct hashTable {

mydata \*\* T;

int size;

};

void insert(hashTable \* hTab, mydata \* dat) {

hashType h = getHash(dat);

hashType start = h % hTab->size;

dat->next = hTab->T[start];

hTab->T[start] = dat;

}

bool search(hashTable \* hTab, mydata \* dat) {

hashType h = getHash(dat);

hashType start = h % hTab->size;

mydata \* tmp = hTab->T[start];

while (tmp != NULL) {

if (cmp(tmp, dat))

return true;

tmp = tmp->next;

}

return false;

}

* **Adresowanie otwarte**

// funkcja dodawania dla adresowania otwartego

int hashInsert(T, k) {

i = 0;

while (i != m) { // m - rozmiar tablicy z danymi

j = h(k, i);

if (T[j] == NULL) {

T[j] = k;

return j;

}

i++;

}

return -1; // nastąpiło przepełnienie

}

// funkcja wyszukiwania dla adresowania otwartego

int hashSearch(T, k) {

i = 0;

while (i != m) {

j = h(k, i);

if (T[j] == NULL)

break;

if (T[j] == k)

return j;

i++;

}

return -1; // brak elementu

}

* **Liniowe**

h(k, i) = h’(k) + i

typedef unsigned int hashType;

struct Data {

char \* firstName;

char \* lastName;

int age;

hashType hash;

};

Data \* EMPTY = new Data{" ", " ", -1, 0};

// przykład funkcji dodawania z rozwiązywaniem linowym

bool insert(Data \* arr[], Data \* data) {

hashType h = getHash(data);

hashType start = h % M;

if (arr[start] == NULL || arr[start] == EMPTY) {

data->hash = h;

arr[start] = data;

return true;

}

int i = start + 1 % M;

int j = 0;

while (j != M) {

if (arr[i] == NULL || arr[start] == EMPTY) {

data->hash = h;

arr[i] = data;

return true;

}

i = (i + 1) % M; j++;

}

return false; // przepełnienie

}

// przykład funkcji wyszukiwania z rozwiązywaniem linowym

int search(Data \* arr[], Data \* el) {

hashType h = getHash(el);

for (int i = 0; i < M; i++) {

int ind = (h + i) % M;

if (arr[ind] == NULL) break;

if (arr[ind]->hash != h) continue;

if (cmp(arr[ind], el)) return ind;

}

return -1;

}

// przykład funkcji usuwania

void remove(Data \* arr[], Data \* el) {

int ind = search(arr, el);

if (ind == -1) return;

Data \* tmp = arr[ind];

delete tmp;

arr[ind] = EMPTY;

}

* **Kwadratowe**

h(k, i) = h’(k) + i^2

* **Adresowanie uniwersalne**

h(k, i) = h’(k) + i\*h’’(k)

**Zbiory rozłączne**

* **Reprezentacja tablicowa**

// O(1)

void makeSetTab(int i, int R[]) {

R[i] = i;

}

// O(1)

int findTab(int i, int R[]) {

return R[i];

}

// O(n)

void unionTab(int x, int y, int R[]) {

int rx = findTab(x, R);

int ry = findTab(y, R);

if (rx != ry) {

for (int i = 0; i < N; i++) {

if (R[i] == ry)

R[i] = rx;

}

}

}

// O(n)

void initTab(int R[]) {

for (int i = 0; i < N; i++) {

makeSetTab(i, R);

}

}

// O(1)

bool isSameSetTab(int x, int y, int R[]) {

return findTab(x, R) == findTab(y, R);

}

* **Reprezentacja listowa**

struct setNode;

struct set {

setNode \* head, \* tail;

int length; // posłuży nam do łączenia krótszej listy z większą

};

struct setNode {

set \* r;

setNode \* next;

int val;

};

// O(1)

set \* findList(setNode \* c) {

return c->r;

}

// O(1)

setNode \* makeSetList(int v) {

set \* s = new set;

setNode \* t = new setNode;

t->r = s;

t->next = NULL;

t->val = v;

s->head = s->tail = t;

s->length = 1;

return t;

}

// O(m) - gdzie m to liczba el. w mniejszej liście

void unionList(setNode \* x, setNode \* y) {

set \* rx = findList(x);

set \* ry = findList(y);

if (rx != ry) {

if (rx->length > ry->length) {

rx->tail->next = ry->head;

rx->tail = ry->tail;

rx->length += ry->length;

while (ry->head != NULL) {

ry->head->r = rx;

ry->head = ry->head->next;

}

delete ry;

} else {

ry->tail->next = rx->head;

ry->tail = rx->tail;

ry->length += rx->length;

while (rx->head != NULL) {

rx->head->r = ry;

rx->head = rx->head->next;

}

delete rx;

}

}

}

// O(n)

void initList(setNode \* tab[]) {

for (int i = 0; i < N; i++) {

tab[i] = makeSetList(rand() % 100);

}

}

* **Reprezentacja drzewiasta (las)**

struct node {

node \* up;

int val, rank;

};

// O(1)

node \* makeSetTree(int v) {

node \* s = new node;

s->up = s;

s->rank = 0;

s->val = v;

return s;

}

//

node \* findTree(node \* s) {

if (s->up != s) {

node \* y = findTree(s->up);

s->up = y;

return y;

} else {

return s;

}

}

//

void unionTree(node \* x, node \* y) {

node \* rx = findTree(x);

node \* ry = findTree(y);

if (rx->rank > ry->rank) {

ry->up = rx;

} else if (rx->rank < ry->rank) {

rx->up = ry;

} if (ry != rx) {

ry->up = rx;

rx->rank++;

}

}

//

void initTree(node \* tab[]) {

for (int i = 0; i < N; i++) {

tab[i] = makeSetTree(rand() % 100);

}

}

* **Zadania z find & union**
* **Program obliczający ilość różnych pól, na które można przejść z wybranej pozycji z tablicy booli A[m][n]. Można przejść na pozycję różniącą się o 1 i będącą wartością true.**

struct point2 {

bool val;

int rx, ry;

point2 \* rep;

};

void makeset(point2 \* &item, bool val) {

item = new point2;

item->val = val;

item->rep = item;

}

point2 \* find(point2 \* item) {

return item->rep;

}

void unionPoint(point2 \* item1, point2 \* item2, point2 \* B[m][n]) {

point2 \* r1 = find(item1);

point2 \* r2 = find(item2);

if (r1 != r2) {

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (B[i][j]->rep == r2)

B[i][j]->rep = r1;

}

}

}

}

int check(point2 \* tab[m][n], int py, int px) {

int count = 0;

for (int y = 0; y < m; y++) {

for (int x = 0; x < n; x++) {

if ((y + 1 < m) && tab[y + 1][x]->val && tab[y][x]->val)

unionPoint(tab[y + 1][x], tab[y][x], tab);

if ((y - 1 >= 0) && tab[y - 1][x]->val && tab[y][x]->val)

unionPoint(tab[y - 1][x], tab[y][x], tab);

if ((x + 1 < n) && tab[y][x + 1]->val && tab[y][x]->val)

unionPoint(tab[y][x + 1], tab[y][x], tab);

if ((x - 1 >= 0) && tab[y][x - 1]->val && tab[y][x]->val)

unionPoint(tab[y][x - 1], tab[y][x], tab);

}

}

for (int y = 0; y<m; y++) {

for (int x = 0; x<n; x++) {

if (find(tab[y][x]) == find(tab[py][px]))

count++;

}

}

return count-1;

}

**Grafy i algorytmy**

**Algorytmy BFS i DFS**

* **BFS (przeszukiwanie wszerz)**
* **Opis**

Przechodzenie grafu rozpoczyna się od zadanego wierzchołka *s* i polega na odwiedzeniu wszystkich osiągalnych z niego wierzchołków. Wynikiem działania algorytmu jest drzewo przeszukiwania wszerz o korzeniu w *s*, zawierające wszystkie wierzchołki osiągalne z *s*.

Złożoność czasowa: O(V + E)

Złożoność pamięciowa: O(V + E)

* **Pseudokod**

// pseudokod

void BFS(graph g, vertex s) {

Queue Q;

for (v in V) {

v.visited = false;

v.parent = NULL;

}

s.visited = true;

Q.push(s);

while (!Q.empty()) {

vertex u = Q.pop();

for (v neighbor u) {

if (!v.visited) {

v.visited = true;

v.d = u.d + 1;

v.parent = u;

Q.push(v);

}

}

}

}

* **Implementacja**

// reprezentacja macierzowa

template <size\_t size>

void BFS(int (&g)[size][size], int vertex, void (\* action)(int)) {

queue< int > Q;

bool \* visited = new bool[size];

int \* parent = new int[size];

for (int i = 0; i < size; i++) {

visited[i] = false;

parent[i] = -1;

}

visited[vertex] = true;

Q.push(vertex);

action(vertex);

while (!Q.empty()) {

int u = Q.front();

Q.pop();

for (int i = 0; i < size; i++) {

if (g[u][i] == 1 && !visited[i]) {

visited[i] = true;

parent[i] = u;

action(i);

Q.push(i);

}

}

}

delete[] visited;

delete[] parent;

}

// reprezentacja listowa

struct node {

int v;

node \* next;

};

void BFSOnList(node \*\* g, int vertex, int size, void(\*action)(int)) {

queue< int > Q;

bool \* visited = new bool[size];

int \* parent = new int[size];

for (int i = 0; i < size; i++) {

visited[i] = false;

parent[i] = -1;

}

visited[vertex] = true;

Q.push(vertex);

action(vertex);

while (!Q.empty()) {

int u = Q.front();

Q.pop();

node \* list = g[u];

while (list != NULL) {

if (!visited[list->v]) {

visited[list->v] = true;

parent[list->v] = u;

action(list->v);

Q.push(list->v);

}

list = list->next;

}

}

}

// przykład użycia – inne rozwiązanie zadania z przechodzeniem po tablicy A[m][n]

struct point {

int rx, ry;

};

const int pos[][2] = {

{1, 0},

{-1, 0},

{0, 1},

{0, -1}

};

int BFS(bool A[m][n], int y, int x) {

stack< point > s;

A[y][x] = false;

s.push({ x, y });

int c = 0;

while (!s.empty()) {

point u = s.top(); s.pop();

for (int i = 0; i < 4; i++) {

int ny = u.ry + pos[i][0];

int nx = u.rx + pos[i][1];

if (ny >= 0 && ny < m &&

nx >= 0 && nx < n &&

A[ny][nx]) {

A[ny][nx] = false;

c++;

s.push({ nx, ny });

}

}

}

return c;

}

* **DFS (przeszukiwanie w głąb)**
* **Opis**

Przeszukiwanie w głąb polega na badaniu wszystkich krawędzi wychodzących z podanego wierzchołka. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka algorytm powraca do wierzchołka, z którego dany wierzchołek został odwiedzony.

Złożoność czasowa: O(V + E)

Złożoność pamięciowa: O(h) h-długość najdłuższej prostej ścieżki

* **Implementacja**

void DFSVisit(int graph[][V\_SIZE], int v, int &timevar, void(\*action)(int, int)) {

visited[v] = true;

for (int i = 0; i<V\_SIZE; i++) {

if (graph[v][i] == 1 && !visited[i]) {

parent[i] = v;

DFSVisit(graph, i, timevar, action);

}

}

action(v, timevar);

timevar = timevar + 1;

timetab[v] = timevar;

}

void DFS(int graph[][V\_SIZE], void(\*action)(int, int)) {

for (int i = 0; i<V\_SIZE; i++) {

visited[i] = false;

timetab[i] = -1;

}

int timevar = 0;

for (int i = 0; i<V\_SIZE; i++) {

if (!visited[i])

DFSVisit(graph, i, timevar, action);

}

}

**Zastosowania algorytmu DFS**

* **Sortowanie topologiczne**
* **Opis**

Sortowanie topologiczne (ang. topological sort) grafu skierowanego polega na utworzeniu listy wierzchołków grafu w taki sposób, aby każdy wierzchołek posiadający sąsiadów znalazł się na tej liście przed nimi. Zadanie to jest możliwe do wykonania tylko wtedy, gdy graf jest **acyklicznym grafem skierowanym** (ang. DAG - directed acyclic graph). Kolejność wierzchołków nie połączonych ścieżką jest dowolna.

* **Implementacja**

struct Vertex {

int \* edges;

int size;

bool visited;

};

stack< int > S;

void DFSVisit(Vertex \* g, int n, int i) {

g[i].visited = true;

for (int j = 0; j < g[i].size; j++) {

int u = g[i].edges[j];

if (!g[u].visited)

DFSVisit(g, n, u);

}

S.push(i);

}

void topolog(Vertex \* g, int n) {

while (!S.empty()) S.pop();

for (int i = 0; i < n; i++)

g[i].visited = false;

for (int i = 0; i < n; i++)

if(!g[i].visited)

DFSVisit(g, n, i);

while (!S.empty()) {

cout << S.top() << endl;

S.pop();

}

}

* **Alternatywne rozwiązanie bez użycia DFS**

struct vertex {

int v;

vertex \* next;

};

int vInd[V\_SIZE]; // tablica stopni wchodzących grafu

// ten algorytm może być stosowany również do badania cykliczności grafu

void topoSort(int g[][V\_SIZE]) {

for (int i = 0; i < V\_SIZE; i++)

vInd[i] = 0;

for (int i = 0; i < V\_SIZE; i++)

for (int j = 0; j < V\_SIZE; j++)

if (g[i][j] == 1)

vInd[j]++;

bool test;

vertex \* list = makeList(V\_SIZE); // z wartownikiem

vertex \* p;

do {

test = false;

p = list;

while (p->next != NULL) {

if (vInd[p->next->v] > 0) {

p = p->next;

continue;

}

test = true; // znaleziono

for (int i = 0; i < V\_SIZE; i++)

if (g[p->next->v][i] == 1)

vInd[i]--;

cout << p->next->v << endl;

vertex \* r = p->next;

p->next = r->next;

delete r;

}

} while (test);

}

* **Znajdowanie spójnych składowych**
* **Spójna składowa grafu nieskierowanego**

[Spójny](https://pl.wikipedia.org/wiki/Graf_sp%C3%B3jny) [podgraf](https://pl.wikipedia.org/wiki/Podgraf) grafu G nie zawarty w większym podgrafie spójnym grafu G. Graf spójny ma jedną spójną składową.

Aby policzyć ilość spójnych składowych należy zliczyć „wyjścia” z rekurencyjnej funkcji DFSVisit:

...

int count = 0;

for (int i = 0; i < V\_SIZE; i++) {

if (!visited[i]) {

DFSvisit(graph, i);

count++;

}

}

...

* **Silnie spójna składowa grafu skierowanego**

Maksymalny [podgraf](https://pl.wikipedia.org/wiki/Podgraf) **H**, a jednocześnie jego [spójna składowa](https://pl.wikipedia.org/wiki/Sp%C3%B3jna_sk%C5%82adowa_grafu), taka, że pomiędzy każdymi dwoma jej [wierzchołkami](https://pl.wikipedia.org/wiki/Wierzcho%C5%82ek_grafu) istnieje [ścieżka](https://pl.wikipedia.org/wiki/%C5%9Acie%C5%BCka_%28teoria_graf%C3%B3w%29). Dla [grafu nieskierowanego](https://pl.wikipedia.org/wiki/Graf_%28matematyka%29) każda [spójna składowa](https://pl.wikipedia.org/wiki/Sp%C3%B3jna_sk%C5%82adowa_grafu) będzie silnie spójna.

Znajdowanie silnie spójnych składowych:

* 1. Wykonaj DFS obliczając czasy przetworzenia
  2. Odwróć kierunek wszystkich krawędzi
  3. Wykonaj DFS idąc po wierzchołkach w kolejności malejących czasów przetworzenia
* **Cykl Eulera**
* **Definicja**

Cykl Eulera to taki [cykl](https://pl.wikipedia.org/wiki/Cykl_%28teoria_graf%C3%B3w%29) w [grafie](https://pl.wikipedia.org/wiki/Graf_%28matematyka%29), który przechodzi przez każdą jego [krawędź](https://pl.wikipedia.org/wiki/Kraw%C4%99d%C5%BA_grafu) dokładnie raz. Graf spójny nieskierowany posiada cykl Eulera gdy każdy stopień jego wierzchołka jest parzysty. Natomiast graf spójny skierowany posiada cykl Eulera gdy każdy jego wierzchołek ma taką samą liczbę krawędzi wchodzących i wychodzących.

* **Implementacja**

void DFSEuler(int g[V\_SIZE][V\_SIZE], int v, int n) {

for (int i = 0; i < n; i++)

while (g[v][i] > 0) {

g[v][i]--;

g[i][v]--;

DFSEuler(g, i, n);

}

S.push(v);

}

bool eulerCycle(int g[V\_SIZE][V\_SIZE], int n) {

while (!S.empty()) S.pop();

for (int i = 0; i < n; i++) {

int c = 0;

for (int j = 0; j < n; j++)

if (g[i][j]) c++;

if (c % 2 != 0) return false;

}

DFSEuler(g, 0, n);

while (!S.empty()) {

cout << S.top() << endl;

S.pop();

}

return true;

}

* **Ścieżka hamiltona**
* **Definicja**

[Ścieżka](https://pl.wikipedia.org/wiki/%C5%9Acie%C5%BCka) w [grafie](https://pl.wikipedia.org/wiki/Graf_%28matematyka%29) przebiegająca przez wszystkie jego [wierzchołki](https://pl.wikipedia.org/wiki/Wierzcho%C5%82ek) dokładnie raz.

* **Znajdowanie mostów w grafie**
* **Definicja**

**Mostem** nazywamy krawędź grafu, której usunięcie zwiększa liczbę spójnych składowych.

**Minimalne drzewo rozpinające**

* **Algorytm Kruskala**
* **Opis**

[Algorytm](https://pl.wikipedia.org/wiki/Algorytm) [grafowy](https://pl.wikipedia.org/wiki/Graf_%28matematyka%29) wyznaczający [minimalne drzewo rozpinające](https://pl.wikipedia.org/wiki/Minimalne_drzewo_rozpinaj%C4%85ce) dla grafu nieskierowanego ważonego, o ile jest on spójny. Innymi słowy, znajduje drzewo zawierające wszystkie wierzchołki grafu, którego waga jest najmniejsza możliwa. Jest to przykład algorytmu zachłannego.

Złożoność czasowa O(E \* logV)

* **Implementacja**

struct edge {

int v, u, w;

};

// find and union

inline void makeSet(int \* tab, int i) {

tab[i] = i;

}

inline int findSet(int \* tab, int i) {

return tab[i];

}

void unionSets(int R[], int x, int y) {

int rx = findSet(R, x);

int ry = findSet(R, y);

if (rx != ry) {

for (int i = 0; i < V\_SIZE; i++) {

if (R[i] == ry)

R[i] = rx;

}

}

}

template <size\_t size>

priority\_queue<edge, vector< edge >, compareWeights>

kruskal(int(&g)[size][size]) {

priority\_queue<edge, vector< edge >, compareWeights> Q, T; // krawędzie wdlg wag

int Z[size]; // tablica reprezentantów

for (int v = 0; v < size; v++) {

makeSet(Z, v);

for (int u = 0; u < size; u++) {

if (g[v][u] > 0) {

edge e = { v, u, g[v][u] };

Q.push(e);

}

}

}

for (int i = 0; i < size-1; i++) {

edge e;

do {

e = Q.top(); Q.pop();

} while (findSet(Z, e.v) == findSet(Z, e.u));

T.push(e);

unionSets(Z, e.v, e.u);

}

return T;

}

* **Algorytm Prima**
* **Opis**

Złożoność czasowa O(E \* logV)

* **Implementacja**

template <size\_t size>

priority\_queue<edge, vector< edge >, compareWeights>

prima(int(&g)[size][size], int v) {

priority\_queue<edge, vector< edge >, compareWeights> Q, T; // krawędzie według wag

bool \* visited = new bool[size];

for (int i = 0; i < size; i++)

visited[i] = false;

visited[v] = true;

edge e;

for (int i = 0; i < size-1; i++) {

for (int u = 0; u < size; u++) {

if (graph[v][u] <= 0) continue;

if (!visited[u]) {

e = { v, u, graph[v][u] };

Q.push(e);

}

}

do {

e = Q.top(); Q.pop();

} while (visited[e.u]);

T.push(e);

visited[e.u] = true;

v = e.u;

}

delete[] visited;

return T;

}

**Znajdowanie najkrótszych ścieżek**

* **Algorytm Dijkstry**
* **Opis**

Algorytm służy do znajdowania najkrótszej ścieżki z pojedynczego źródła w grafie o nieujemnych wagach krawędzi. Mając dany graf z wyróżnionym wierzchołkiem (*źródłem*) algorytm znajduje odległości od źródła do wszystkich pozostałych wierzchołków, przy okazji wyliczając również koszt przejścia każdej ze ścieżek.

Złożoność czasowa:

O(E \* logV)

O(E + V \* logV) – przy użyciu kopca Fibonacciego

* **Implementacja**

/\* reprezentacja macierzowa \*/

int d[V\_SIZE]; // koszty dojścia

int p[V\_SIZE]; // tablica poprzedników

bool visited[V\_SIZE];

int getMinD() {

int min\_v, i;

for (i = 0; i < V\_SIZE && visited[i]; i++);

min\_v = i;

for (; i < V\_SIZE; i++) {

if (!visited[i] && d[i] < d[min\_v])

min\_v = i;

}

return min\_v;

}

template <size\_t size>

void pathProcess(int(&g)[size][size], int v) {

for (int i = 0; i < size; i++) {

visited[i] = false;

d[i] = INT16\_MAX;

p[i] = -1;

}

d[v] = 0;

int num\_vis = 0;

while (num\_vis < size) {

int u = getMinD();

for (int i = 0; i < size; i++) {

if (g[u][i] <= 0) continue;

if (d[i] > d[u] + g[u][i]) {

d[i] = d[u] + g[u][i];

p[i] = u;

}

}

visited[u] = true;

num\_vis++;

}

}

/\* wyświetla najkrótszą ścieżkę \*/

void showShortestPath(int \* parents, int length, int dest) {

if (dest != -1 && length - 1 >= 0) {

showShortestPath(parents, length - 1, parents[dest]);

cout << dest << endl;

}

}

/\* reprezentacja listowa \*/

struct edge {

int v;

int w;

edge \* next;

};

struct vertex {

int d;

int parent;

struct edge \* edge;

bool visited;

};

struct vertexId {

int id;

int d;

};

void dijkstraOnList(vertex \* graph, int n, int start) {

start--;

if (start < 0) return;

for (int i = 0; i < n; i++) {

graph[i].d = INT16\_MAX;

graph[i].parent = -1;

graph[i].visited = false;

}

graph[start].d = 0;

priority\_queue<vertexId, vector< vertexId >, compare> queue;

vertexId s = { start, graph[start].d };

queue.push(s);

while (!queue.empty()) {

int u = queue.top().id; queue.pop();

if (graph[u].visited) continue;

graph[u].visited = true;

struct edge \* edge = graph[u].edge;

while (edge != NULL) {

int v = edge->v;

if (graph[u].d + edge->w < graph[v].d) {

graph[v].d = graph[u].d + edge->w;

graph[v].parent = u;

vertexId vid = { v, graph[v].d };

queue.push(vid);

}

edge = edge->next;

}

}

}

* **Algorytm Bellmana-Forda**
* **Opis**

**Algorytm Bellmana-Forda** rozwiązuje [problem najkrótszej ścieżki](https://pl.wikipedia.org/wiki/Problem_najkr%C3%B3tszej_%C5%9Bcie%C5%BCki), tj. pozwala znaleźć ścieżkę o najmniejszej wadze pomiędzy dwoma [wierzchołkami](https://pl.wikipedia.org/wiki/Wierzcho%C5%82ek_grafu) w [grafie](https://pl.wikipedia.org/wiki/Graf_%28matematyka%29) ważonym. Idea [algorytmu](https://pl.wikipedia.org/wiki/Algorytm) opiera się na [metodzie relaksacji](https://pl.wikipedia.org/wiki/Metoda_relaksacji) (dokładniej następuje relaksacja V − 1 {\displaystyle V-1} razy każdej z [krawędzi](https://pl.wikipedia.org/wiki/Kraw%C4%99d%C5%BA_grafu)).

W odróżnieniu od [algorytmu Dijkstry](https://pl.wikipedia.org/wiki/Algorytm_Dijkstry), poprawność algorytmu Bellmana-Forda nie opiera się na założeniu, że wagi w grafie są nieujemne (nie może jednak występować cykl o łącznej ujemnej wadze osiągalny ze źródła). Za tę ogólność płaci się jednak wyższą [złożonością czasową](https://pl.wikipedia.org/wiki/Z%C5%82o%C5%BCono%C5%9B%C4%87_obliczeniowa).

Złożoność czasowa: O(V\*E)

Złożonośćpamięciowa: O(V)

* **Implementacja**

int d[V\_SIZE]; // koszty dojścia

int p[V\_SIZE]; // tablica poprzedników

template <size\_t size>

bool BF(int(&g)[size][size], int s) {

for (int i = 0; i < size; i++) {

d[i] = INT16\_MAX;

p[i] = -1;

}

d[s] = 0;

for (int i = 0; i < size - 1; i++) {

bool test = true;

for (int u = 0; u < size; u++) {

for (int v = 0; v < size; v++) {

if (g[u][v] == 0 || g[u][v] == INT16\_MAX) continue;

if (d[v] > d[u] + g[u][v]) {

test = false;

d[v] = d[u] + g[u][v];

p[v] = u;

}

}

}

if (test) return true;

}

for (int u = 0; u < size; u++)

for (int v = 0; v < size; v++)

if (g[u][v] != 0 && g[u][v] != INT16\_MAX && d[v] > d[u] + g[u][v])

return false;

return true;

}

* **Algorytm Floyda-Warshalla**

void floydWarshall(int g[][V\_SIZE]) {

for (int i = 0; i < V\_SIZE; i++) {

for (int j = 0; j < V\_SIZE; j++) {

if (i == j)

d[i][j] = 0;

else if (g[i][j] != 0)

d[i][j] = g[i][j];

else

d[i][j] = INT16\_MAX;

}

}

for (int k = 0; k < V\_SIZE; k++) {

for (int i = 0; i < V\_SIZE; i++) {

for (int j = 0; j < V\_SIZE; j++) {

if (d[i][j] > d[i][k] + d[k][j])

d[i][j] = d[i][k] + d[k][j];

}

}

}

}

**Sieci przepływowe**

* **Sieci przepływowe i przepływy**

**Siecią przepływową** G=(V, E) nazywamy graf spójny skierowany, w którym każda krawędź ma nieujemną **przepustowość** (ang. capacity) c(u, v)>=0. Jeśli między u i v nie ma krawędzi przyjmujemy, że c(u, v)=0. W sieci rozróżniamy dwa wierzchołki: **źródło** s i **ujście** t.

**Przepływem** w sieci G nazywamy każdą funkcję f: VxV->R spełniającą poniższe warunki:

**- Warunek przepustowości:** Dla wszystkich u, v należących do V zachodzi:

f(u, v )<=c(u, v)

**-** **Warunek skośnej symetryczności:** Dla wszystkich u, v należących do V zachodzi:

f(u, v)=-f(v, u)

**- Warunek zachowania przepływu:** Dla każdego u należącego do V\{s, t} zachodzi:

Wielkość f(u, v), która może byc dodatnia lub ujemna nazywamy **przepływem netto** z wierzchołka u do v. Wartość przepływu definiuje się jako:

* **Sieci residualne**
* **Opis**

Intuicyjnie, dla danej sieci przepływowej i pewnego przepływu sieć residualna składa się z krawędzi, które dopuszczają większy przepływ netto. **Przepustowością residualną** dla (u, v) definiujemy następująco: cf(u, v)=c(u, v)-f(u, v). Ponieważ w odwrotnym kierunku również występuje przepływ (ujemny), w sieci rezydualnej mogą pojawiać się dodatkowe kanały skierowane przeciwnie do kanałów pierwotnych. Dla kanałów przeciwnych przepustowość rezydualna wyraża się wzorem: cf(v, u) = f(u, v).

Dana jest sieć G=(V, E) oraz przepływ f. Siecią residualną dla sieci G indukowaną przez przepływ f nazywamy sieć Gf=(V, Ef), w której Ef={(u, v) ∈ VxV: cf(u, v)>0 }. Oznacza to, że każda krawędź w sieci residualnej, lub inaczej **krawędź residualna**, umożliwia dodatni przepływ netto.

**Sieżką powiększającą (rozszerzającą)** p nazywamy każdą scieżkę ze źródła s do ujścia t w sieci residualnej Gf. Największy możliwy przpływ netto po krawędziach ścieżki powiększającej p nazywamy **przepustowością residualną ścieżki powiększającej** i definiujemy ją następująco: cf(p)=min{cf(u, v): (u, v) jest na p}.

Każdy podział zbioru V na S i T=V\S taki, że s nalezy do S i t należy do T, nazywamy **przekrojem** (S, T) w sieci G.

|  |  |  |
| --- | --- | --- |
| http://eduinf.waw.pl/inf/alg/001_search/images/0145_05.gif | → | http://eduinf.waw.pl/inf/alg/001_search/images/0145_06.gif |

* **Metoda Floyda-Fluckersona**
* **Pseudokod**

Ford\_Flukerson\_Method(Graph g, int s, int t) {

for (edge in E) {

f[u, v] = 0;

f[v, u] = 0;

}

while (augmenting path p exists) {

c = smallest capacity on p path

for (edge in p) {

f[u, v] += c;

f[v, u] -= c;

}

}

}

* **Maksymalny przepływ – algorytm Forda-Fulkersona/Edmondsa-Karpa**
* **Pseudokod**

Ford\_Fulkerson(Graph g, int s, int t) {

for (edge in E) {

f[u, v] = 0;

f[v, u] = 0;

}

int fmax = 0;

// scieżka roszerzająca powinna być najkrótsza pod względem liczności krawędzi

// wtedy algorytm działa optymalnie

// do tego celu należy użyć BFSa, tak zaimplementowaną metodę Forda-Fulkersona

// nazywamy algorytmem Edmondsa-Karpa, który ma złożoność O(VE^2)

while (augmenting path p exists) {

cf\_p = smallest capacity on p path; // znajdujemy tzw. Wąskie gardło na ścieżce

for (edge in p) {

f[u, v] += cf\_p;

f[v, u] = -f[u, v];

cf[u, v] = c[u, v] – f[u, v]; // zgodnie z kierunkiem, tzw. zapas,

// jeśli f(u, v) == c(u, v) to kanał zanika

cf[v, u] = f[u, v];

}

fmax += cf\_p;

}

}

**PROGRAMOWANIE DYNAMICZNE**

**Podciągi**

* **LIS (longest increasing subsequence)**

void printSolution(int A[], int P[], int i) {

if (i != -1) {

printSolution(A, P, P[i]);

cout << A[i] << " ";

}

}

int lis(int A[], int n) {

int \* F = new int [n];

int \* P = new int [n];

int max = 0;

F[0] = 1;

P[0] = -1; // tablica poprzedników

for (int i = 1; i < n; i++) {

F[i] = 1;

P[i] = -1;

for (int j = 0; j < i; j++) {

if (A[j] < A[i] && F[j] + 1 > F[i]) {

F[i] = F[j] + 1;

P[i] = j;

}

}

if (F[max] < F[i]) max = i;

}

printSolution(A, P, max);

cout << endl;

int c = F[max];

delete[] F;

delete[] P;

return c;

}

* **LCS (longest common subsequence) i LIS oparty na LCS**

/\*

f(i, j) returns length of LCS (longest common subseqence)

A[0...i]

B[0...j]

| max{ f(i-1, j), f(i, j-1) } gdy A[i] != B[j]

f(i, j) = |

| f(i-1, j-1)+1 gdy A[i] = B[j]

\*/

int LCS(int A[], int B[], int n, const int m) {

int \*\* f = new int \*[n+1];

for (int i = 0; i < n + 1; i++)

f[i] = new int[m + 1];

for (int i = 0; i < n + 1; i++)

f[i][0] = 0;

for (int i = 0; i < m + 1; i++)

f[0][i] = 0;

for (int i = 1; i < n + 1; i++) {

for (int j = 1; j < m + 1; j++) {

if (A[i] != B[j])

f[i][j] = max(f[i-1][j], f[i][j-1]);

else

f[i][j] = f[i-1][j-1] + 1;

}

}

return f[n][m];

}

int LIS(int A[], int n) {

int \* B = new int[n];

for (int i = 0; i < n; i++)

B[i] = A[i];

sort(B, B + n); // sortuje B

int result = LCS(A, B, n, n);

delete[] B;

return result;

}

**Problem plecakowy**

/\*

f: NxN->N - maksymalna wartość przedmiotów

p1 ... pn

f(0, m) = 0 (m>=0)

f(i, 0) = 0 (i>=0)

| f(i-1, m) gdzie wi > m

f(i, m) = |

| max{ f(i-1, m), f(i-1, m-w(pi)) + c(pi)}

\*/

int backpack(int W[], int C[], int n, int m) {

n++; m++;

int \*\* f = new int \* [n];

for (int i = 0; i < n; i++)

f[i] = new int[m];

for (int i = 0; i < n; i++)

f[i][0] = 0;

for (int i = 0; i < m; i++)

f[0][i] = 0;

for (int i = 1; i < n; i++) {

for (int j = 1; j < m; j++) {

if (W[i-1] > j)

f[i][j] = f[i - 1][j];

else {

int w = j - W[i-1];

if (w < 0) w = 0;

f[i][j] = max(f[i - 1][j], f[i-1][w] + C[i-1]);

}

}

}

return f[n-1][m-1];

}

**PROGRAMOWANIE Zachłanne**

**Problem wyboru zajęć**

struct activity {

double start;

double end;

};

queue< int > greedyActivitySelector(activity A[], int n) {

qsort(A, n, sizeof(\*A), sortAsc); // sortuj rosnąco względem końca zajęć

queue< int > Q;

int j = 0;

Q.push(j);

for (int i = 1; i < n; i++) {

if (A[i].start >= A[j].end) {

Q.push(i);

j = i;

}

}

return Q;

}