# Metody Programowania

Hubert Jaremko

28 czerwca 2019

# Spis treści

# 1 Sortowania proste

## 1.1 Bubble Sort

```
void bubbleSort(int[] arr) {
    for (int k = arr.length - 1; k >= 1; k--) {
        for (int j = 0; j < k; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(j, j + 1);
            }
        }
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(n^2)$
- Średnia: $\Theta(n^2)$ (ok. $\frac{n^2}{2}$ porównań)

ZALETY

- Stabilny.
- W miejscu.

WADY

- Najmniej efektywne.

MOŻLIWE USPRAWNIENIA

- Ustawiać koniec wewnętrznej pętli na miejsce ostatnio wykonanej zamiany.
- Zapamiętanie, czy w pętli wewnętrznej potrzebna była zamiana.
- Cocktail Shaker Sort - kolejne fazy wykonuje się na przemian, rozpoczynając porównania elementów, raz od początku - drugi raz od końca tablicy. Pozwala to przyśpieszenie sortowania w przypadku tablic, np. `9, 1, 2, 3, 4, 5, 0`.

## 1.2 Cocktail Shaker Sort

```java
void cocktailSort(int[] arr) {
    int bottom = 0;
    int top = arr.length - 1;
    boolean swapped = true;

    while (swapped == true) {
        swapped = false;
        //int lastSwap = bottom;

        for (int i = bottom; i < top; i++) {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
                swapped = true;
                // lastSwap = i;
            }
        }

        top--;
        // top = lastSwap;
        // lastSwap = top;

        for (int i = top; i > bottom; i--) {
            if (arr[i] < arr[i - 1]) {
                swap(arr, i, i - 1);
                swapped = true;
                // lastSwap = i;
            }
        }

        bottom++;
        // bottom = lastSwap;
    }
}
```

## 1.3 Selection Sort

```java
void selectionSort(int[] arr) {
    for (int k = 0; k < arr.length - 1; k++) {
        int min = k;

        for (int j = k + 1; j < arr.length; j++) {
            if (arr[j] < arr[min]) {
                min = j;
            }
        }

        swap(k, min);
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(n^2)$
- Średnia: $\Theta(n^2)$ (ok. $\frac{n^2}{2}$ porównań)

ZALETY

- Wykonuje tylko $n-1$ przestawień - zalecana w przypadku niedługich tablic z długimi rekordami.

WADY

- Niestablilny.

MOŻLIWE USPRAWNIENIA

- Kosztem zwiększenia współczynnika proporcjonalności złożoności, można ten algorytm uczynić stabilnym (zamiast zamiany przesunąć element na odpowiednie miejsce).

## 1.4 Insertion Sort

```java
void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length - 1; i++) {
        int selected = arr[i];
        int j = i - 1;

        while (j >= 0 && selected < arr[j]) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = selected;
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(n^2)$
- Średnia: $\Theta(n^2)$ (ok. $\frac{n^2}{2}$ porównań)

ZALETY

- Stablilny.
- Średnio dwukrotnie szybsza niż inne proste metody.
- Optymalna dla ciągów prawie posortowanych.

MOŻLIWE USPRAWNIENIA

- Wyszukiwanie miejsca do wstawienia metodą binarną.
- Na początku w `arr[0]` ustaw najmniejszy element tablicy, będzie spełniał rolę wartownika, wtedy w wewnętrznej pętli:
  `while (j >= 0 && v < arr[j]) { ... }`
  wystarczy warunek `v < a[j]`, oraz pętla wewnętrzna `for` może zacząć się od `i = 2`.

# 2 Sortowania zaawansowane

## 2.1 Merge Sort

REKURENCYJNIE

```
void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}

void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[arr.length];

    for (int i = left; i <= right; i++)
        temp[i] = arr[i];

    int i = left;
    int j = mid + 1;
    int k = left;

    while (i <= mid && j <= right) {
        if ( temp[i] <= temp[j] )
            arr[k++] = temp[i++];
        else
            arr[k++] = temp[j++];
    }

    while (i <= mid) // && i < n) w iteracyjnym
        arr[k++] = temp[i++];
    while (j <= right) // nie trzeba w iteracyjnym
        arr[k++] = temp[j++];

}
```

ITERACYJNIE

```
void mergeSort(int[] arr, int left, int right) {
    for (int size = 1; size ⩽ n - 1; size = 2 * size) {
        for (int left = 0; left < n - 1; left += 2 * size) {
            int mid = min(left + size - 1, arr.length - 1);
            right = min(left + 2 * size - 1, arr.length - 1);

            merge(left, mid, right);
        }
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(n \log_2 n)$
- Pamięciowa: $\Theta(n)$

ZALETY

- Stabilna.

WADY

- Pamięć robocza rozmiaru $\Theta(n)$.
- Czasochłonne przepisywanie elementów.

## 2.2 Quick Sort

REKURENCYJNIE

```
void quickSort(int[] arr, int left, int right) {
    if (left < right) {
        int pivot = partition(arr, left, right);

        quickSort(left, pivot - 1);
        quickSort(pivot + 1, right);
    }
}
```

ITERACYJNIE

```
void quickSort(int[] arr, int left, int right) {
    while (left < right || !stack.isEmpty()) {
        if (left < right) {
            int pivot = partition(arr, left, right);

            //na stos prawe podzadanie
            stack.push(right);

            right = pivot - 1;
        }
        else {
            left = right + 2;
            right = stack.pop();
        }
    }
}
```

HOARE

```
int partition(int[] arr, int left, int right) {
    int i = left - 1;
    int j = right;
    int x = arr[right]; //ostatni elemnt jest dzielący

    while (true) {
        while (arr[++i] < x);
        while (j > left && arr[--j] > x);

        if (i >= j)
            break;
        else
            swap(i, j);
    }

    swap(i, right);
    return i;
}
```

LOMUTO

```
int partition(int[] arr, int left, int right) {
    int i = left - 1;
    int x = arr[right]; //ostatni element jest dzielący

    for (int j = left; j < right; j++) {
        if (arr[j] <= x) {
            swap(++i, j);
        }
    }

    swap(i + 1, right);
    return i + 1;
}
```

ZŁOŻONOŚĆ

- **Optymistyczna:** $\Theta(n \log_2 n)$
- **Średnia:** $\Theta(n \log_2 n)$
- **Pesymistyczna:** $\Theta(n^2)$
- **Pamięciowa:** Konieczność stosu: $\Theta(n)$, usprawniony $\Theta(\log_2 n)$

ZALETY

- Jest to w ogólnym przypadku tablicy najszybszy algorytm wykorzystujący operację porównywania elementów.

WADY

- **Niestabilność.**
- Koszt $\Theta(n^2)$ w przypadku pesymistycznym.

MOŻLIWE USPRAWNIENIA

- W przypadku małych podzadań (np. `n ⩽ 20`) sortuj prostą metodą (zalecana metoda przez wstawianie).
- Usprawnienia wyboru pivota:
  - **Randomizacja** - wszystkie możliwe wielkości podzadań są jednakowo prawdopodobne.
  - **Mediana trzech elementów** - wybierz środkowy element (medianę) spośród trzech elementów `A[L], A[(L+R)/2], A[R]` i zamień go z `A[R]` (3 porównania więcej). Ta metoda optymalnie radzi sobie z ciągiem uporządkowanym, ale nadal można skonstruować dowolnie długie ciągi, które wymagają czasu $\Omega(n^2)$. Oczekiwana liczba porównań w tym wariancie wynosi około $1.2n \log_2 n$, ale w praktyce czas wykonania zwykle jest gorszy.
- Można porównywać rozmiary podzadań otrzymywanych w wyniku `partition()` i iteracyjnie przechodzić do mniejszego, a większe zapamiętywać na stosie. Gwarantuje to wielkość stosu $O(\log_2 n)$.
- Można w ogóle pozbyć się stosu, organizując go wewnętrznie w sortowanej tablicy. Zmniejsza się w ten sposób złożoność pamięciową do $O(1)$, ale zwiększa się współczynnik proporcjonalności złożoności czasowej.

## 2.3 Shell Sort

```
void shellSort(int[] arr, int n) {
    int h = 1; //przyrost

    while (h <= n / 3) {
        h = h * 3 + 1;
    }

    while (h > 0) { //zmniejszamy h, aż do momentu h == 1
        for (int k = h; k < n; k++) {
            int tmp = arr[k];
            int j = k;

            while (j >= h && arr[j - h] >= tmp) {
                arr[j] = arr[j - h];
                j -= h;
            }

            arr[j] = tmp;
        }

        h = (h - 1) / 3;
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $O(n(\log_2 n)^2)$
- Pamięciowa: $\Theta(1)$

OPIS METODY

- W tej metodzie stosuje się wielokrotnie sortowanie przez wstawianie dla elementów odległych od siebie nazywaną *przyrostem*, który maleje od pewnej wartości by w końcu przyjąć wartość 1.

## 2.4 Count Sort

### ZAŁOŻENIA

- Elementy tablicy `a[0], a[1], ..., a[n - 1]` przyjmują wartości nieujemne i mniejsze niż **m**, to znaczy dla $i = 0, \ldots, n-1$ $\mathtt{a}[\mathtt{i}] \in \{0, 1, \ldots, m-1\}$
- Tablice pomocnicze: `b[n], count[m]`

### PSEUDOKOD

```
for (int j = 0; j < m; j++) count[j] = 0;

for (int i = 0; i < n; i++)
    count[a[i]]++;

for (int j = 1; j < m; j++)
    count[j] += count[j - 1];

for (int i = n - 1; i >= 0; i--)
    b[--count[a[i]]] = a[i];

for (int i = 0; i < n; i++) a[i] = b[i];
```

### OPIS METODY

1. Zerowanie tablicy `count[]`.
2. Zliczanie - ile jest elementów w tablicy `a[]` o danej wartości.
3. Obliczanie górnych granic obszarów wynikowych dla poszczególnych wartości.
4. Przepisz od końca `a[]` do `b[]` na właściwe miejsca (*zapewnia* ***stabilność***).
5. Opcjonalnie, mozna przepisać `b[]` z powrotem do `a[]`.

### ZŁOŻONOŚĆ

- Czasowa: $\Theta(n+m)$

### WADY

- Pamięć pomocnicza wielkości $\Theta(n+m)$.
- Wartości elementów muszą być liczbami całkowitymi ograniczonej wielkości.

## 2.5 Radix Sort

### ZAŁOŻENIA

- Jeśli klucze `a[i]` są długie, to znaczy ich zakres (parametr $m$) jest zbyt duży do zastosowania **count sort**, albo klucze są złożone z kilku składowych.

### PSEDOKOD

```
void radixSort() {
    for (int k = 0; k < b; k++)
        countSort(a, k);
}
```

Funkcja `countSort(a, k)` używa jako klucza $k$-ty bajt elementu `a[i]`.

### OPIS METODY

1. Podziel klucze na kilka części, na przykład:
   - kolejne cyfry dziesiętne,
   - kolejne fragmenty z zapisu bitowego klucza
2. Zastosuj **count sort** tyle razy, ile jest części, każdą z nich traktując jako klucz w jednym przebiegu.

Ważna jest kolejność wykonywania **count sort** dla części kluczy od najmniej znaczących poczynając, na najbardziej znaczących kończąc.

Załóżmy, że klucze `a[i]` są $b$-bajtowe

- `a[i][k]` - oznacza $k$-ty bajkt klucza `a[i]`, $k = b - 1, \ldots, 0$.
- `a[i][0]` - to bajt najmniej znaczący klucza `a[i]`.
- `m = 256`, bo tyle jest możliwych wartości klucza 1-bajtowego.

### ZŁOŻONOŚĆ

- Czasowa: $\Theta(b \cdot (n + m))$
- Pamięciowa: $\Theta(n + m)$

## 2.6 Bucket Sort

### ZAŁOŻENIA

- Ciąg kluczy do posortowania to `a[0], a[1], ... , a[n - 1]` typu rzeczywistego.
- Wartość każdego klucza należy do uporządkowanego zbioru np. liczb rzeczywistych: $\{q_1 < q_2 < \ldots < q_m\}$, to znaczy $\forall\ i \in \{0, \ldots, n-1\}\ \ \exists\ j \in \{1, \ldots, m\} : \mathtt{a}[\mathtt{i}] = q_j$

### OPIS METODY

1. Utwórz $m$ kubełków (np. list), początkowo pustych.
2. Dla `i = 0, ... , n - 1` , jeśli `a[i] =` $q_j$ to wstaw `a[i]` do kubełka o numerze $j$ *(wyszukiwanie binarne)*.
3. Uczyń tablicę `a[]` pustą.
4. Kolejno dla `j = 1, ... , m` przepisz zawartość kubełka $j$ do tablicy `a[]` (tak więc w `a[]` pojawią się najpierw klucze o wartości $q_1$, potem klucze o wartości $q_2$, itd.).

### PSEUDOKOD

```
double Q[m];  // Q = {q1, q2, ... , qm}
int count[m]; // tablica liczników

for (int j = 0; j < m; j++)
    count[j] = 0;

for (int i = 0; i < n; i++) {
    //m-numer kubełka do którego należy a[i]
    int m = binary_search(a[i], Q);
    count[m]++;
}

for (int j = 0; j < m; j++) {
    for (int p = 0; p < count[j]; p++)
        wypisz(Q[j]);
}
```

### ZŁOŻONOŚĆ

- Czasowa: $O(n \log_2 m)$
- Pamięciowa: $O(n + m)$

# 3 Wyszukiwania

## 3.1 Wyszukiwanie binarne

```
void binarySearch(int key) {
    int begin = 0;
    int end = n - 1;

    while (begin <= end) {
        int current = (begin + end) / 2;

        if (arr[current] == key) {
            return current;
        }
        else {
            if (arr[current] < key)
                begin = current + 1;
            else
                end = current - 1;
        }
    }

    return -1;
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(\log_2 n)$ (optymalne)
- Średnia: $\Theta(\log_2 n)$

## 3.2 Wyszukiwanie interpolacyjne

- Zakładamy liniowy rozkład wartości elementów.

$$\frac{curr - low}{upp - low} = \frac{x - \mathtt{a}[\mathtt{low}]}{\mathtt{a}[\mathtt{upp}] - \mathtt{a}[\mathtt{low}]}$$

$$curr = low + (x - \mathtt{a}[\mathtt{low}])\frac{upp - low}{\mathtt{a}[\mathtt{upp}] - \mathtt{a}[\mathtt{low}]}$$

# 4 Selekcja

## 4.1 Metoda Hoare

PSEUDOKOD

```
Item Select1(S, k) {
// znajdowanie k-tego (1 < k < n) co do wielkości elementu
// zbioru S, licząc od najmniejszego
// (k = 1 - element najmniejszy)
    a = dowolny element zbioru S;
    Przeglądnij zbiór S i wyznacz zbiory:
        S1 = {x in S : x < a};
        S2 = {x in S : x = a};
        S3 = {x in S : x > a};

    if (k ≤ |S1| ) return Select1(S1, k);
        // szukany element jest k-tym elementem w S1
    if (k ≤ |S1| + |S2|) return a;
        // szukany jest w S2, czyli równy a
    return Select1(S3, k - |S1| - |S2|);
        // szukaj w S3, numer odpowiednio mniejszy
}
```

ZŁOŻONOŚĆ

- Czasowa pesymistyczna: $O(n^2)$ *(jak quicksort)*
- Czasowa średnia: $O(n)$

## 4.2 Algorytm magicznych piątek

PSEUDOKOD

```
Item Select2(S, k) {
    (1) Jeśli n < p to posortuj i wypisz k-ty element. //baza
    (2) Podziel S na 5-elementowe podzbiory i ewentualnie
        jeden co najwyżej 4-elementowy.
    (3) Posortuj każdy podzbiór oddzielnie.
    (4) Wyznacz nowy zbiór Q = {środkowe elementy z każdego
                                podzbioru}
    (5) Wyznacz M = Select2(Q, |Q| / 2)
        //rekurancja, mediana median
    (6) Dalej jak w metodzie Hoare:

    Przeglądnij zbiór S i wyznacz zbiory:
        S1 = {x in S : x < M};
        S2 = {x in S : x = M};
        S3 = {x in S : x > M};

    if (k ⩽ |S1| ) return Select2(S1, k);
        // szukany element jest k-tym elementem w S1
    if (k ⩽ |S1| + |S2|) return M;
        // szukany jest w S2, czyli równy M
    return Select2(S3, k - |S1| - |S2|);
        // szukaj w S3, numer odpowiednio mniejszy
}
```

ZŁOŻONOŚĆ

- Czasowa pesymistyczna: $O(n)$

# 5 Liczba inwersji

```
long numInversion(int left, int right) //mergeSort
{ // wywołanie: numInversion(0, size - 1)
    long invs = 0;

    if (right > left) {
        int mid = (left + right) / 2;

        invs  = numInversion(left, mid); // ilosc z lewej i prawej czesci
        invs += numInversion(mid + 1, right);
        invs += merge(left, mid, right); // ilosc inwersji z laczenia
    }

    return invs;
}

long merge(int left, int mid, int right)
{
    copyToTemp(from left to mid); //kopiujemy tylko połowę

    int i = left;
    int j = mid + 1;
    int k = left;

    long invs = 0;

    while (i <= mid && j <= right) {
        if (temp[i] <= data[j])
            data[k++] = temp[i++];
        else {//temp[i] > data[j]
            data[k++] = data[j++];

            invs += mid + 1 - i; //skoro lewa i prawa polowa sa
            // posortowane to pozostale elementy w lewej polowie
            // sa wieksze od data[j], zatem jest mid + 1 - i inwersji
        }
    }

    while (i <= mid) //przepisz pozostale
        data[k++] = temp[i++];

    while (j <= right)
        data[k++] = data[j++];

    return invs;
}
```

# 6 Algorytm Kadane

```
int beginMax = 0;
int endMax = 0;
int sumMax = 0;
int beginBest = 0;
int currentSum = 0;

for (int i = 0; i < n; i++) {
    currentSum += arr[i];

    if (currentSum < 0) {
        currentSum = 0;
        beginBest = i + 1;
    }
    else if (currentSum > sumMax) {
        sumMax = currentSum;
        beginMax = beginBest;
        endMax = i;
    }
}
```

ZŁOŻONOŚĆ

- Pesymistyczna: $\Theta(n)$
- Pamięciowa: $\Theta(1)$

OPIS

- Metoda oblicza maksymalną podtablicę kończącą się w `i`, mając obliczoną maksymalną podtablicę kończącą się w `i - 1`.
- Zauważamy, że maksymalna podtablica dla `arr[0 .. i]` jest:
  - albo zawarta w `arr[0 .. i - 1]`
  - albo kończy się na `arr[i]`

# 7 Problem plecakowy

ALGORYTM

1. Jeśli w jakimkolwiek momencie realizacji procesu suma wag wybranych elementów będzie równa wadze docelowej, należy zakończyć działanie (sukces).

2. Początkowo wybierany jest pierwszy element. Po wybraniu wyznaczamy nową wagę docelową, jako różnicę dotychczasowej wagi docelowej i wagi pierwszego wybranego elementu. Jeśli suma wag wybranych elementów nie będzie równa wadze docelowej, należy wybrać następny element.

3. Kolejno należy wypróbować wszystkie dostępne kombinacje pozostałych elementów. Należy jednak zauważyć, że w rzeczywistości wcale nie trzeba sprawdzać wszystkich kombinacji gdyż sumowanie można zakończyć w momencie, gdy sumaryczna waga wybranych elementów przekracza wagę docelową.

4. Jeśli nie uda się odnaleźć kombinacji elementów o zadanej wadze, to należy odrzucić pierwszy element i rozpocząć cały proces od początku, wybierając element kolejny.

5. W podobny sposób należy rozpocząć cały proces sprawdzania, wybierając na początku trzeci, czwarty oraz kolejne elementy, aż do momentu przeanalizowania całego zbioru dostępnych elementów. Jeśli sprawdzenie wszystkich możliwości nie zakończy się sukcesem, będzie to oznaczać, że poszukiwane rozwiązanie nie istnieje.

Rozwiązująca ten problem metoda rekurencyjna mogłaby wybrać pierwszy element ze zbioru dostępnych elementów, a następnie, jeśli waga jest mniejsza od sumarycznej wagi docelowej, wywołać samą siebie, by sprawdzić sumę wag pozostałych dostępnych elementów.

# 8 Wieże Hanoi

## OPIS METODY

Niech na wieży źródłowej - **A** znajduje się $n$ krążków, chcemy przenieść wszystkie krążki z wieży **A** na wieżę docelową - **B**, przy czym dostępna jest wieża pomocnicza **C**.

1. Przenieś poddrzewo składające się z $n-1$ krążków z wieży **A** na wieżę **C**.
2. Przenieś ostatni (największy krążek) z **A** na wieżę docelową **B**.
3. Przenieś poddrzewo z wieży **C** na **B**.

## PSEUDOKOD

```
void Towers(n, A, B, C) {
    if (n == 0)
        return;

    Towers(n - 1, A, C, B);
    A → B;
    Towers(n - 1, C, B, A);
}
```

## ZŁOŻONOŚĆ

- **Pesymistyczna:** $\Theta(2^n)$

# 9 Eliminacja rekurencji

Po wywołaniu metody jej rekord aktywacji reprezentujący aktualny stan wywoływanej funkcji i zawierający:

- obiekty lokalne – zmienne, stałe, parametry
- adres powrotu (aktualny licznik rozkazów)

jest wstawiany na stos i następuje skok do początku kodu metody. Bezpośrednio przed zakończeniem działania metody ze stosu pobierany jest rekord aktywacji, a następnie są odtwarzane obiekty lokalne z przed wywołania i następuje skok do adresu powrotu w pobranym rekordzie.

# 10 Usuwanie duplikatów z tablicy posortowanej

```
void removeDuplicates(int[] arr, int n) {
    if (n == 0 || n == 1)
        return;

    int j = 0;

    for (int i = 0; i < n - 1; i++)
        if (arr[i] != arr[i + 1])
            arr[j++] = arr[i];

    arr[j++] = arr[n - 1];
    n = j;
}
```

# 11 Listy

## 11.1 Jednokierunkowa

### 11.1.1 Wstawianie za podanym elementem

```
void insertAfter(T elem, Node p) {
    Node newElem = new Node(elem);

    if (p == null) { //wstawianie na poczatek
        newElem.next = first;
        first = newElem;
    }
    else {
        newElem.next = p.next;
        p.next = newElem;
    }
}
```

### 11.1.2 Usuwanie elementu o podanej wartości

```java
void delete(T key) {
    Node curr = first;
    Node prev = null;

    while (curr != null && p.data != key) {
        prev = curr;
        curr = curr.next;
    }

    if (curr != null) {
        if (prev == null) {
            first = p.next;
        }
        else {
            prev.next = p.next;
        }
    }
}
```

## 11.2 Dwukierunkowa

### 11.2.1 Usuwanie elementu o podanej wartości

```java
void delete(T key) {
    Node elem = find(key);

    if (curr != null) {
        elem.prev.next = elem.next;
        elem.next.prev = elem.prev;
    }
}
```

### 11.2.2 Wstawanie elementu za elementem o podanej wartości

```
void insertAfter(T x, T key) {
    Node elem = find(key);
    Node newNode = new Node(x);

    if (elem == last) {
        newNode.next = null;
        last = newNode;
    }
    else {
        newNode.next = elem.next;
        elem.next.prev = newNode
    }

    newNode.prev = elem;
    elem.next = newNode;
}
```

### 11.2.3 Wstawanie elementu na początek

```
void insertFirst(T value) {
    Node newNode = new Node(value);

    if (isEmpty()) {
        last = newNode;
    }
    else {
        first.prev = newElem;
    }

    newNode.next = first;
    first = newElem;
}
```

## 11.3 Prosta cykliczna

### 11.3.1 Budowa

W ostatniej komórce referencja `next` jest adresem pierwszej komórki listy (lub nagłówka, jeśli wersja z nagłówkiem). Zamiast testu `p ≠ null` należy zastosować test `p ≠ s`, gdzie `s` jest referencją komórki, od której zaczęliśmy przegląd listy.

### 11.3.2 Usuwanie elementu o podanej wartości

```
void insertFirst(T value) {
    Node curr = first;

    while (curr.next ≠ first && curr.next.data ≠ value) {
        curr = curr.next;
    }

    if (curr.next ≠ first) {
        curr.next = curr.next.next;
    }
}
```

# 12 Stosy

## 12.1 Przy użyciu listy wiązanej

Lista wiązana <u>bez nagłówka</u> jest bardzo efektywną realizacją stosu. Zmienna `top` wskazuje wierzchołek stosu, który jest na początku listy, zatem - operacje wstawiania `push(x)` i usuwania `pop()` polegają na wstawianiu lub usuwaniu pierwszego elementu listy wiązanej.

## 12.2 Dostęp do największego elementu w czasie O(1)

W każdej komórce przechowujemy aktualną największą wartość.

# 13 Kolejki

## 13.1 Prosta przy użyciu listy wiązanej dwustronnej

### 13.1.1 Wstawianie

```
void enqueue(T x) {
    Node newNode = new Node(x);
    rear.next = newNode;
    rear = newNode;
}
```

### 13.1.2 Usuwanie

```
T dequeue() {
    T tmp = front.next.info;
    front = front.next;

    return tmp;
}
```

## 13.2 Priorytetowa przy użyciu uporządkowanej listy wiązanej

## 13.3 Prosta przy użyciu tablicy

```
class Queue {
    int maxSize;
    long[] elem;
    int front = 0;
    int rear = 0;

    private int addOne(int i) {
        return (i + 1) % maxSize;
    }


    void enqueue(long x) {
        if (isFull())
            error("Queue is full");
        else {
            elem[rear] = x;
            rear = addOne(rear);
        }
    }

    long dequeue() {
        if (isEmpty()) {
            error("Queue is empty");
            return -1;
        }
        else {
            long tmp = elem[front];
            front = addOne(front);
            return tmp;
        }
    }

    boolean isFull() {
        return (addOne(rear) == front);
    }
}
```

# 14 Drzewa

## 14.1 Preorder

REKURENCYJNIE

```
void preorder(Node root) {
    if (root != null) {
        print(root.info);
        preorder(root.left);
        preorder(root.right);
    }
}
```

ITERACYJNIE

```
void preorder() {
    Stack<Node> stack = new Stack<>();
    Node current = root;

    while (current != null || !stack.isEmpty()) {
        if (current != null) {
            print(current.info);
            stack.push(current.right);
            current = current.left;
        }
        else {
            current = stack.pop();
        }
    }
}
```

## 14.2 Inorder

REKURENCYJNIE

```
void inorder(Node root) {
    if (root != null) {
        inorder(root.left);
        print(root.info);
        inorder(root.right);
    }
}
```

ITERACYJNIE

```
void inorder() {
    Stack<Node> stack = new Stack<>();
    Node current = root;

    while (current != null || !stack.isEmpty()) {
        if (current != null) {
            stack.push(current);
            current = current.left;
        }
        else {
            current = stack.pop();
            print(current.info);
            current = current.right;
        }
    }
}
```

## 14.3 Postorder

REKURENCYJNIE

```
void postorder(Node root) {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        print(root.info);
    }
}
```

ITERACYJNIE

```
void postorder() {
    Stack<Node> stack = new Stack<>();
    Node current = root;

    while (current != null || !stack.isEmpty()) {
        while (current != null) {
            if (current.right != null) {
                stack.push(current.right);
            }

            stack.push(current);
            current = current.left;
        }

        current = stack.pop();

        if (current.right != null && stack.top() == current.right) {
            stack.pop();
            stack.push(current);
            current = current.right;
        }
        else {
            print(current.info);
            current = null;
        }
    }
}
```

## 14.4 Levelorder

```
void levelorder() {
    if (root == null) {
        return;
    }

    Queue<Node> queue = new Queue<>();
    queue.pushBack(root);

    while (!queue.isEmpty()) {
        Node current = queue.popFront();
        print(current.info);

        if (current.left != null) {
            queue.pushBack(current.left);
        }

        if (current.right != null) {
            queue.pushBack(current.right);
        }
    }
```

## 14.5 Drzewa poszukiwań binarnych (BST)

Jest to drzewo binarne, w którym lewe poddrzewo każdego węzła zawiera **wyłącznie elementy o kluczach nie większych niż klucz węzła a prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła**.

Dla pełnego drzewa BST o $n$ węzłach pesymistyczny koszt każdej z podstawowych operacji wynosi $O(\log_2 n)$.

Drzewo binarne jest BST wtedy i tylko wtedy gdy lista jego węzłów w porządku `inorder` jest ciągiem **niemalejącym**.

### 14.5.1 Wyszukanie wartości minimalnej

```java
Node minimum() { // Maksymalna analogicznie tylko w prawo
    Node current = root;
    Node last;

    while (current ≠ null) {
        last = current;
        current = current.left;
    }

    return last;
}
```

### 14.5.2 Wyszukanie danego klucza

```java
Node search(int key) {
    Node current = root;

    while (current ≠ null && key ≠ current.info) {
        if (key < current.info) {
            current = current.left;
        }
        else {
            current = current.right;
        }
    }

    return current;
}
```

### 14.5.3 Wstawianie danego klucza

ITERACYJNIE

```java
void insert(int key) {
    Node s, p, prev;
    s = new Node(key); //s.left = null; s.right = null;

    if (root == null) //drzewo puste
        root = s;
    else {
        p = root;
        prev = null;

        while (p != null) {
            prev = p;

            if (key < p.info)
                p = p.left;
            else
                p = p.right;
        }

        if (key < prev.info)
            prev.left = s;
        else
            prev.right = s;
    }
}
```

REKURENCYJNIE

```java
void insert(Node p, int key) { //wywołanie: insert(root, x);
    if (p == null) { //baza rekurencji, tworzenie węzła
        p = new Node(key);

        if (root == null)
            root = p;
    }
    else {
        if (key < p.info)
            p.left = insert(p.left, key);
        else
            p.right = insert(p.right, key);
    }

    return p; //referencja do wstawianego węzła
}
```

### 14.5.4 Usuwanie danego klucza

ITERACYJNIE

```
void delete(int key) {
    Node parent = _getParent(key, root);
    Node _root = root;

    while (true) {
        parent = _getParent(key, parent);
        Node curr = find(key, _root);

        if (curr == null) return;

        //oba poddrzewa puste
        if (curr.left == null && curr.right == null) {
            if (curr != root) {
                if (parent.left == curr)
                    parent.left = null;
                else
                    parent.right = null;
            }
            else root = null;

            return;
        } //oba poddrzewa niepuste
        else if (curr.left != null && curr.right != null) {
            Node succ = min(curr.right);
            curr.info = succ.info;

            key = succ.info;
            _root = curr.right;
            parent = curr;
        } //jedno niepuste poddrzewo
        else {
            Node child = (curr.left != null) ? curr.left : curr.right;

            if (curr != root) {
                if (parent.left == curr)
                    parent.left = child;
                else
                    parent.right = child;
            }
            else root = child;

            return;
        }
    }
}
```

REKURENCYJNIE

```
void delete(int key, Node root) {
    // baza
    if (root == null)
        return root;

    if (key < root.info)
        root.left = delete(key, root.left);
    else if (key > root.info)
        root.left = delete(key, root.right);
    else {// klucz taki sam jak roota wiec usuwamy go
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // węzeł z dwoma potomkami
        // weź następnik (najmniejszy w prawym)
        root.info = minValue(root.right);

        // usuń następnik
        root.right = delete(root.info, root.right);
    }

    return root;
}
```

### 14.5.5 Wyszukanie rodzica danego klucza

```java
void parent(int key) {
    Node p = root; //zaczynamy od korzenia
    Node prev = null;

    if (p == null)
        return null;

    if (p.info == key)
        return null;

    while (p != null && p.info != key) { //dopóki nie znaleziono
        prev = p;

        if (key < p.info)
            p = p.left;
        else
            p = p.right;

        if (p == null) //brak potomka -> nie odnaleziono
            return null;
    }

    return prev; //odnalzeiono
}
```

### 14.5.6 Wyszukanie poprzednika danego klucza

```java
void predecessor(int key) {
    Node p, q;
    q = search(key);

    if (q == null) //nie ma klucza zwraca null
        return null;

    if (q.left != null) { //szukamy maksimum w lewym przedziale
        p = q.left;

        while (p.right != null)
            p = p.right;

        return p;
    }

    p = parent(key);

```

```
    while (p != null && q == p.left) {
        q = p;              //idziemy w górę i szukamy węzeł p
        p = parent(p.info); //którego prawym następnikiem jest q
    }

    return p; //zwraca null jeśli nie ma poprzednika
}
```

### 14.5.7 Wyszukanie następnika danego klucza

```
void successor(int key) {
    Node p, q;
    q = search(key);

    if (q == null) //nie ma klucza zwraca null
        return null;

    if (q.right != null) { //szukamy minimum w prawym przedziale
        p = q.right;

        while (p.left != null)
            p = p.left;

        return p;
    }

    p = parent(key);

    while (p != null && q == p.right) {
        q = p;              //idziemy w górę i szukamy węzeł p
        p = parent(p.info); //którego prawym następnikiem jest q
    }

    return p; //zwraca null jeśli nie ma następnika
}
```

# 15 Kopce

## 15.1 Definicja

Jest to drzewo binarne, w którym:

- dla każdego węzła zachodzi **warunek kopca**
  - `key(v.parent) ⩾ key(v)` *(max-kopiec)*
- wszystkie poziomy, za wyjątkiem ostatniego, są całkowicie wypełnione
- ostatni poziom jest wypełniony z lewej strony

## 15.2 Implementacja

Postawową implementacją jest **tablica**.

- **Lewym** potomkiem węzła `a[i]` jest `a[2 * i + 1]`
- **Prawym** potomkiem węzła `a[i]` jest `a[2 * i + 2]`
- **Przodkiem** węzła `a[i]` jest `a[(i - 1) / 2]`

## 15.3 Sortowanie przez kopcowanie

ZŁOŻONOŚĆ: $\Theta(n \log n)$

```
void heapsort(int n) {
    // tworzenie kopca w tablicy a
    // rozpoczynamy analizę od ostatniego elementu,
    // który ma dzieci
    for (int k = (n - 2) / 2; k >= 0; k--)
        downheap(k, n);

    //usuwanie z kopca
    while (n > 0) {
        swap(a[0], a[--n]);
        downheap(0, n);
    }
}
```

## 15.4 Przesiewanie w górę

```
// podobnie jak podczas sortowania przez wstawianie
// listą jest ścieżką od węzła k do korzenia
void upheap(int k) {
    int i = (k - 1) / 2;  // indeks przodka elementu a[k]
    int tmp = a[k];

    while (k > 0 && a[i] < tmp) {
        a[k] = a[i];
        k = i;              // przenieść węzeł w dół
        i = (i - 1) / 2; // przejdź do przodka
    }
   // teraz element a[k] na swoje miejsce
   a[k] = tmp;
}
```

## 15.5 Przesiewanie w dół

```
// podobnie jak wstawianie do listy w insertsort
// lista (ścieżka do liścia) wyznaczana dynamicznie
void downheap(int k, int n) {
    int j = 0;
    int tmp = a[k];

    while (k < n / 2) {
        j = 2 * k + 1; // indeks lewego potomka a[k]

        // wybierz większy z potomków
        if (j < n - 1 && a[j] < a[j + 1])
            j++;

        if (tmp >= a[j])
            break; // warunek kopca OK

        // w przeciwnym wypadku
        // przesuń aktualny element do góry
        a[k] = a[j];
        k = j;
    }
   // teraz element a[k] na swoje miejsce
   a[k] = tmp;
}
```

## 15.6 Kolejka priorytetowa

### 15.6.1 Wstawianie do kolejki

ZŁOŻONOŚĆ: $\Theta(\log n)$

```
boolean insert(int x, int n) {
    if (n == size)
        return false;

    a[n] = x;
    upheap(n++);

    return true;
}
```

### 15.6.2 Odczyt elementu największego

ZŁOŻONOŚĆ: $\Theta(1)$

```
int getMax() {
    return a[0];
}
```

### 15.6.3 Usuwanie elementu największego

ZŁOŻONOŚĆ: $\Theta(\log n)$

```
int deleteMax(int n) {
    // zakładamy, że kopiec ma n elementów (n > 0)
    // wstaw ostatni liść w miejsce korzenia
    // i przesiej w dół
    int root = a[0];
    a[0] = a[--n];
    downheap(0, n);
    return root;
}
```

# 16 Grafy

## 16.1 Przeglądanie wszerz (BFS)

### OPIS METODY

Startujemy z **s**, chcemy dotrzeć do wszystkich wierzchołków następującej kolejności: najpierw wierzchołki odległe od startowego o $1$, następnie o $2$, następnie o $3$, itd.
Podstawa realizacji - **kolejka**, która przechowuje węzły do których już dotarliśmy, ale dla których jeszcze nie badaliśmy możliwych wyjść prowadzących dalej.

ZŁOŻONOŚĆ: $\Theta(n+m)$
PSEUDOKOD

```
void breadthFirstSearch(Graph G, Node s) {
    G.vertexList[s].visited = true; // oznacz jako odwiedzony
    G.displayVertex(s);              // wyświetl

    Queue Q = new Queue();
    Q.insert(s); // wstaw do kolejki
    int v = 0;

    while (!Q.isEmpty()) { // do opróżnienia kolejki
        int u = Q.first();

        // dopóki nie ma odwiedzonych sąsiadów
        // do v pobierz kolejnego sąsiada u
        while ((v = G.getAdjUnvisitedVertex(u)) != -1) {
            G.vertexList[v].visited = true; // oznacz v jako odwiedzony
            G.displayVertex(v);
            Q.insert(v);
        }

        Q.delete(); // usuń u
    }
}
// zwraca nie odwiedzony wierzchołek przyległy do v
int getAdjUnvisitedVertex(int v) {
    for (int j = 0; j < nVerts; j++)
        if (adjMat[v][j] == 1 && vertexList[j].visited == false)
            return j;
    return -1;
}
```

## 16.2 Przeglądanie w głąb (DFS)

### OPIS METODY

Idź do nowych wierzchołków najdalej jak się da, jeśli dalej nie można to wycofaj się do poprzedniego i próbuj inną krawędzią.
Podstawa realizacji - **stos**, który można zrealizować niejawnie, za pomocą rekurencji.

### ZŁOŻONOŚĆ: $\Theta(n+m)$

### PSEUDOKOD *(iteracyjnie)*

```
void depthFirstSearch(Graph G) { //rozpocznij od węzła 0
    G.vertexList[0].visited = true;
    G.displayVertex(0);

    Stack S = new Stack();
    S.push(0);

    while (!S.isEmpty()) {
        // pobierz nie odwiedzony węzeł przyległy do
        // szczytowego elementu stosu (u)
        int v = G.getAdjUnvisitedVertex(S.top());

        if (v == -1)
            S.pop();
        else { // jeżeli istnieje oznacz v
            G.vertexList[v].visited = true;
            G.displayVertex(v);
            S.push(v);
        }
    }
}
```

### PSEUDOKOD *(rekurencyjnie)*

```
void depthFirstSearch(Graph G, int v) {
    G.vertexList[v] = true;
    G.displayVertex(v);
    int i = 0;

    while ((i = G.getAdjUnvisitedVertex(v)) != -1) {
        dfs(G, i);
    }
}
```

## 16.3 Przeglądanie w głąb (DFS) (Cormen)

PSEUDOKOD

```
DFS(G) {
    for each u in V do { //inicjalizacja
        color[u] = white;
        P[u] = null; //poprzednik w drzewe
    }

    time = 0; //globalna

    for each u in V do {
        if (color[u] == white)
            DFS-Visit(u);
    }
}

DFS-Visit(u) {
    color[u] = grey;
    d[u] = ++time;

    //badaj (u, v)
    for each v in L[u] do {
        if (color[v] == white) {
            P[v] = u;
            DFS-Visit(v);
        }
    }

    color[u] = black;
    f[u] = ++time;
}
```

- `L[ ]` - lista sąsiedztwa
- `P[ ]` - poprzednik w drzewie
- `d[ ]` - czas odwiedzenia
- `f[ ]` - czas przetworzenia

## 16.4 Minimalne drzewo rozpinające (MST)

### OPIS METODY

**Minimalne drzewo rozpinające** to podgraf o najmniejszej liczbie krawędzi wymaganej do połączenia w zadanym grafie (spójnym/silnie spójnym). Dla danego grafu spójnego istnieje wiele różnych minimalnych drzew rozpinających.

### PSEUDOKOD *(iteracyjnie)*

```
void minimalSpanningTree(Graph G) {
    G.vertexList[0].visited = true;

    Stack S = new Stack();
    S.push(0);

    while (!S.isEmpty()) {
        int u = S.top();
        // pobierz nie odwiedzony węzeł przyległy do
        // szczytowego elementu stosu (u)
        int v = G.getAdjUnvisitedVertex(u);

        if (v == -1)
            S.pop();
        else { //jeżeli istnieje
            G.vertexList[v].visited = true;
            S.push(v);

            // wyświelt krawędź od u do v
            G.displayVertex(u);
            G.displayVertex(v); //print(',');
        }
    }
}
```

## 16.5 Test acykliczności grafu skierowanego

OPIS METODY

- Graf zorientowany zawiera cykl wtedy i tylko wtedy gdy w dowolnym drzewie **DFS** istnieje krawędź **wsteczna**.
- To znaczy, gdy podczas przeglądania grafu metodą **DFS** odwiedzimy wierzchołek pokolorowany na **szaro** to graf zawiera cykl.

PSEUDOKOD *(Cormen)*

```
DFS(G) {
    for each u in V do { //inicjalizacja
        color[u] = white;
    }

    for each u in V do {
        if (color[u] == white)
            if (DFS-Visit(u) == true)
                return true;
    }

    return false;
}

DFS-Visit(u) {
    color[u] = grey;

    //badaj (u, v)
    for each v in L[u] do {
        if (color[v] == white) {
            if (DFS-Visit(v) == true)
                return true;
        }
        else if (color[v] == grey) {
            return true;
        }
    }

    color[u] = black;
    return false;
}
```

## 16.6 Wyznaczanie spójnych składowych grafu nieskierowanego

- Należy obliczyć `ss[u] = k` - numer spójnej składowej zawierającej wierzchołek `u`, dla każdego $u \in V$ wierzchołki w tej samej składowej dostaną jeden numer `ss`.
- Zmienną globalną `k`, zerowaną na początku, zwiększamy o 1 przy każdym wywołaniu funkcji `DFS-Visit` z głównej pętli w algorytmie DFS.
- Na początku funkcji `DFS-Visit` wykonujemy instrukcję `ss[u] = k`.

Do obliczenia `ss[]` w algorytmie **DFS** można opuścić:

- trzeci kolor (wystarczą dwa)
- poprzednik w drzewie, `P[]`
- tablice `d[]`, `f[]`.

Oba powyższe problemy można rozwiązać również wykorzystując algorytm **BFS**.
**PSEUDOKOD** *(Cormen)*

```
DFS(G) {
    for each u in V do //inicjalizacja
        color[u] = white;

    k = 0; //globalna

    for each u in V do {
        if (color[u] == white) {
            k++;
            DFS-Visit(u);
        }
    }
}

DFS-Visit(u) {
    ss[u] = k;
    color[u] = grey;

    for each v in L[u] do {
        if (color[v] == white)
            DFS-Visit(v);
    }
}
```

## 16.7 Sortowanie topologiczne

OPIS METODY

- Zastosuj algorytm `DFS(G)`, wpisując wierzchołek **u** na początek listy w momencie jego kolorowania na **czarno**.

PSEUDOKOD *(Cormen)*

```
DFS(G) {
    for each u in V do //inicjalizacja
        color[u] = white;

    for each u in V do {
        if (color[u] == white) {
            if (DFS-Visit(u) == false)
                return;
        }
    }
}

DFS-Visit(u) {
    color[u] = grey;

    for each v in L[u] do {
        if (color[v] == white) {
            if (DFS-Visit(v) == false)
                return false;
        }
        else if (color[u] == grey)
        //graf zawiera cykl → nie można posortować!
            return false;
    }

    color[u] = black;
    print(u);
    return true;
}
```

## 16.8 Średnica grafu nieskierowanego

### OPIS METODY

- Obliczamy największą odległość dla każdego wierzchołka stosując algorytm BFS.

### PSEUDOKOD

```
int bfs(Graph G, Vertex s) {
    for each u in G.V {
        color[u] = white;
        dist[u] = Integer.MAX_VALUE; //infinity
    }

    color[s] = grey;
    dist[s] = 0;
    Q.push(s);

    while (!Q.isEmpty()) {
        int u = Q.pop();

        for each v in L[u] {
            if (color[v] == white) {
                color[v] = grey;
                dist[v]++;
                Q.push(v);
            }
        }
    }

    return maxValue(dist);
}

int diameter() {
    int result = 0;

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

        if (d > result)
            d = result;
    }

    return result;
}
```

# 17 Grafy ważone

## 17.1 Najkrótsze ścieżki przy ustalonym źródle

### OGÓLNA METODA ROZWIĄZYWANIA

- `D[v]` - wyliczone dotychczas najlepsze oszacowanie od góry dla `d(s, v)`, początkowo równe $+\infty$.
- `P[v]` - poprzednik wierzchołka `v`, początkowo równy `null`.

```
Init() {
    for each v in V {
        D[v] = Integer.MAX_VALUE; //+infinity
        P[v] = null;
    }
}
```

- dla wszystkich krawędzi, w odpowiedniej kolejności wykonaj odpowiednią liczbę razy procedurę relaksacji:

```
Relax(Vertex u, v) {
    if (D[v] > D[u] + w[u, v]) {
        D[v] = D[u] + w[u, v]; //poprawiono oszacowanie
        P[v] = u; //nowy poprzednik na ścieżce
    }
}
```

### ZŁOŻONOŚĆ

- Inicjalizacja: $\Theta(n)$
- `Relax`: $\Theta(1)$

### 17.1.1 Algorytm Bellman-Ford (B-F)

#### OPIS METODY

- Ponieważ nie ma cykli ujemnych, powtarzanie jakiegokolwiek cyklu na ścieżce jej nie skraca, zatem najkrótsze ścieżki są elementarne i mają nie więcej niż $n - 1$ krawędzi.
- Zatem jeśli pierwszy raz wykonamy procedurę `Relax` dla wszystkich krawędzi, to otrzymamy najkrótsze odległości po ścieżkach 1 - krawędziowych. Powtórzywszy to, otrzymamy najkrótsze odległości po ścieżkach 2 - krawędziowych, itd.

#### ZŁOŻONOŚĆ

- $\Theta(n \cdot m)$ - gdy graf reprezentowany przez listę sąsiedztwa
- $\Theta(n^3)$ - gdy graf reprezentowany przez macierz sąsiedztwa *(szukając krawędzi z wierzchołka trzeba przeglądać cały wiersz tablicy)*.

```
BellmanFord(Graph G = (V, E)) {
    Init();
    D[s] = 0;
    for (i = 1; i < n; i++)
        for each (u, v) in E
            Relax(u, v);
}
```

#### SPRAWDZENIE CZY GRAF ZAWIERA UJEMNE CYKLE

- Zauważmy, że jeśli ma taki cykl, to po zakończeniu B-F, jeszcze jedna iteracja pętli `for each` będzie 'poprawiać' oszacowanie `D[v]` dla pewnego wierzchołka `v`. Nie powiększa to złożoności algorytmu.

```
boolean Test() {
    for each (u, v) in E
        if (D[v] > D[u] + w[u, v])
            return false; // graf zawiera ujemny cykl
    return true; // graf OK
}
```

### 17.1.2 Algorytm Dijkstra (DIJ 1959)

OPIS METODY

- Zakładamy, że wagi krawędzi są **nieujemne**.

1. W zbiorze *(kolejce)* `Q` przechowuj wierzchołki `v`, dla których `D[v]` być może nie ma ostatecznej wartości, początkowo `Q = V` i ustaw `D[s] = 0`.
2. W kolejnych krokach:
   (a) Znajduj $u \in Q$ taki, że `D[u]` jest **najmniejsze**.
   (b) Usuń `u` z `Q` i popraw pozostałym $v \in Q$ ich `D[v]` względem `u`.

```
Dijkstra(G = (V, E)) {
    Init();
    D[s] = 0;
    Q = V; //kolejka wierzchołków

    while (Q ≠ empty) {
        u = DelMin(Q);
        for each v in L[u] // lista następników u
            Relax(u, v);
    }
}
```

ZŁOŻONOŚĆ:

1. **`Q` jest zwykłą listą:** $\Theta(n^2)$
   - wyszukanie i usunięcie minimum: $\Theta(n)$
   - przeglądnięcie listy następników i wykonanie `Relax`: $\Theta(n)$
   - oba powyższe - wykonywane w pętli zwenętrznej $n$ razy
2. **`Q` jest min-kopcem, a graf jest reprezentowany przez listy sąsiedztwa:** $\Theta(m \log n)$
   - $n$ razy `DelMin()`: $\Theta(n \log n)$
   - $m$ raz `Relax()`: $\Theta(m \log n)$ *(każda krawędz raz, koszt przesiewania log n)*

## 17.2 Najkrótsze ścieżki między wszystkimi wierzchołkami

### 17.2.1 Algorytm Floyda

```
Floyd(G = (V, E)) { //O(n^3)
    // inicjalizacja: ścieżki 1-krawędziowe
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            D[i, j] = w[i, j];

    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++) {
                t = D[i, k] + D[k, j];
                if (t < D[i, j]) {
                    D[i, j] = t;
                    P[i, j] = k;
                }
            }
}
```

## 17.3 Przechodnie domknięcie grafu

### 17.3.1 Algorytm Warshalla

```
Warshall(G = (V, E)) {
    // inicjalizacja
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            if ((i, j) in E)
                D[i, j] = 1;
            else
                D[i, j] = 0;

    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (D[i, j] == 0)
                    D[i, j] = D[i, k] && D[k, j];
}
```