Zawartość:

- algorytmy i ich rodzaje
- złożoność obliczeniowa
- sortowanie bąbelkowe
- sortowanie przez wybieranie
- sortowanie przez wstawianie
- sortowanie szybkie
- teoria grafów
- obliczanie drogi w grafie
- obliczanie drzewa rozpinającego

Algorytm

Algorytm to instrukcja prowadząca do rozwiązania problemu w skończonym czasie. Taka definicja pasuje nie tylko do algorytmu w programowaniu, ale i ogólnie.

Algorytm posiada:

- określone parametry wejściowe
- określone dane wyjściowe
- skończony czas trwania
- złożoność (o tym niżej)

Algorytmy można przedstawić w formie:

- listy kroków
- schematu blokowego
- pseudokodu (przypomina kod, ale posiada pewne uproszczenia i skróty)
- kodu

Podział algorytmów:

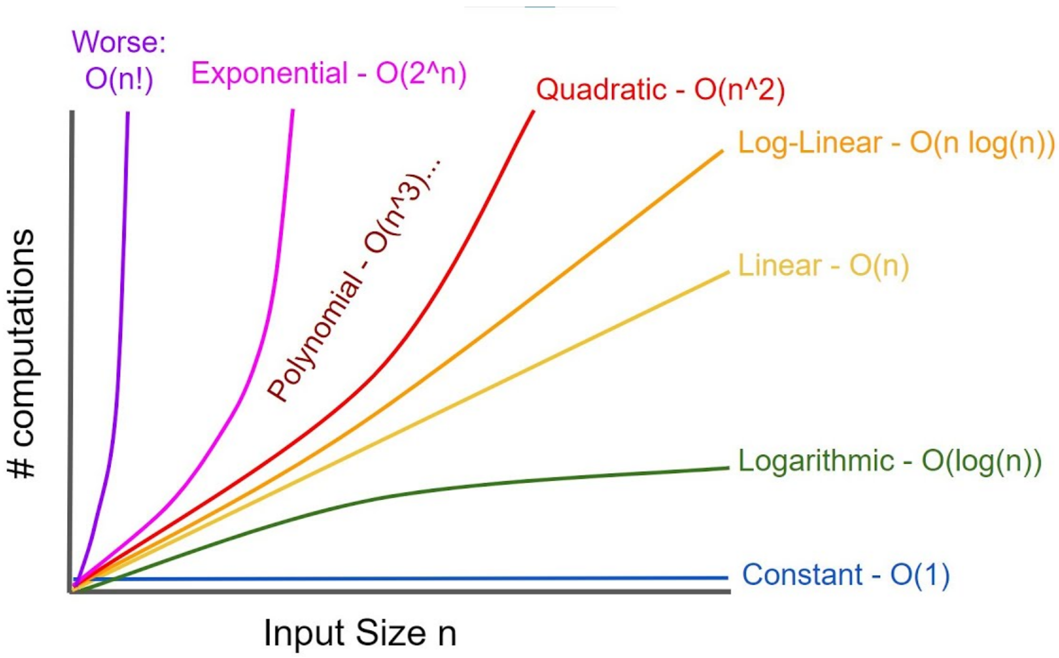
- algorytmy dziel i zwyciężaj – algorytmy, które dzielą problem na kawałki, przykładem jest sortowanie szybkie, które dzieli zbiór na dwie części, te części na kolejne dwie części i tak dalej
- algorytmy zachłanne – wybierają najbardziej korzystną ścieżkę w danym momencie, przykładem jest algorytm Kruskala, o którym niżej
- algorytmy z powrotami/algorytmy dynamiczne – algorytmy, które na każdym etapie zapamiętują wszystkie ścieżki, tak by można było do nich wracać (coś co na początku wydaje się optymalne może już takie nie być 2 kroki dalej)

Złożoność algorytmów

Złożoność czasowa – ilość operacji do wykonania względem ilości danych wejściowych n. Przykład: O(n^2) – ilość operacji jest równa ilości elementów wejściowych do kwadratu (taka sytuacja ma miejsce gdy mamy np. pętle w pętli).

Złożoność pamięciowa – ilość bajtów zużywana względem ilości elementów wejściowych

Złożoności najlepiej sobie wyobrażać jak wykres funkcji. Na osi X mamy ilość elementów wejściowych a na Y złożoność. Im bardziej płaska (wolniej rosnąca) funkcja tym lepiej. Dlatego złożoność logarytmiczna jest lepsza niż złożoność wykładnicza.



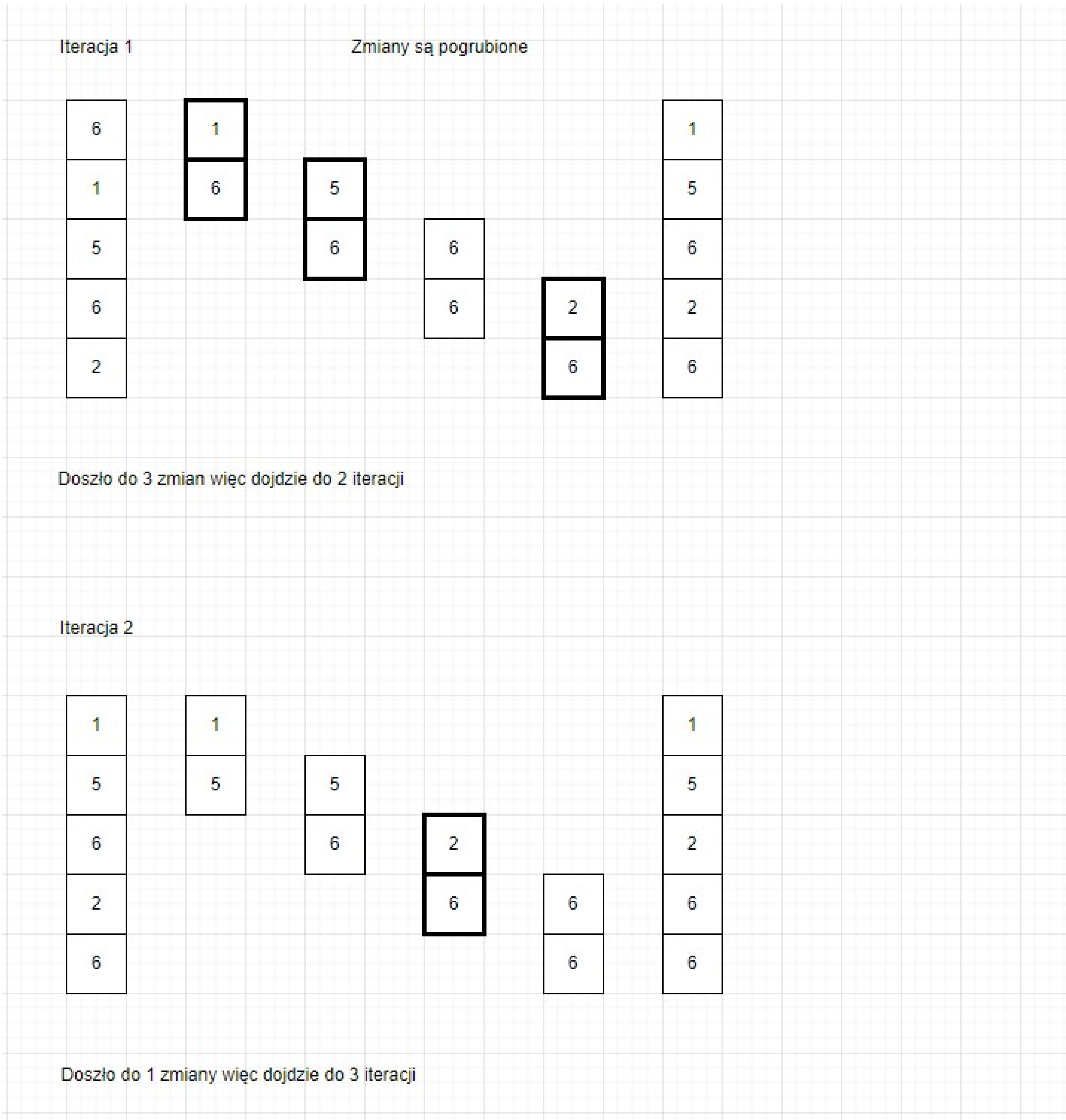
N to oczywiście ilość elementów wejściowych.

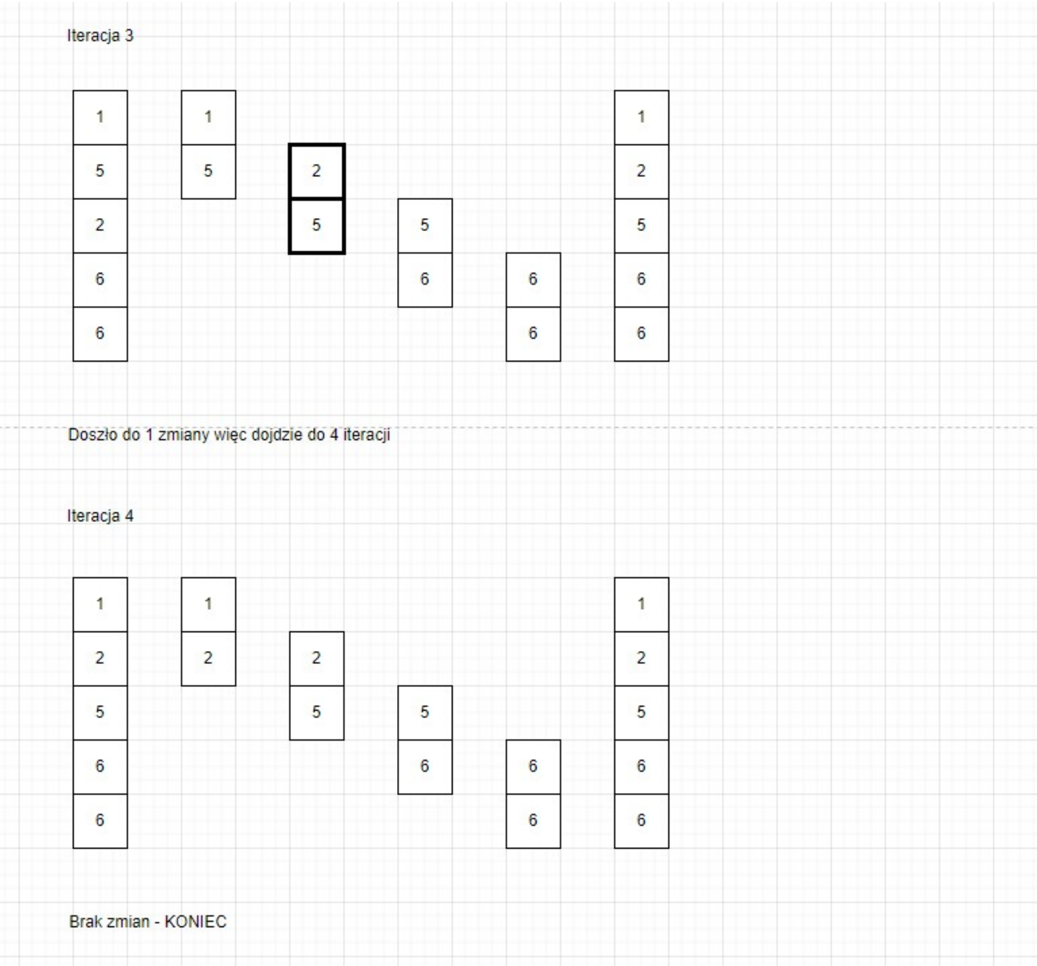
Zazwyczaj jak ktoś podaje złożoność algorytmu to podaje tę czasową.

Sortowanie bąbelkowe – złożoność O(n^2)

Algorytm polega na zamianie sąsiadujących ze sobą elementów do momentu, aż przejście po całej tablicy nie wykona żadnej zamiany.

Przechodzimy po tablicy i zamieniamy sąsiadów jeżeli następnik jest większy od poprzednika. Zliczamy zamiany. Jeżeli liczba zamian jest większa niż 0 to powtarzamy czynność. Jeżeli jest równa 0 to kończymy.





Implementacja:

```
function bubble(data){
    let counter = 0; //licznik zmian
    do{
        counter = 0;
        for(let i = 0;i<data.length-1;i++){
            if(data[i]>data[i+1]){
                let temp = data[i];
                data[i] = data[i+1];
                data[i+1] = temp;
                counter++;
            }
        }
    }
    while(counter!=0);
}

let data = [6,1,5,6,2];
bubble(data);
console.log(data);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER .NET INTERACTIVE

```
C:\Users\siedl\Desktop\PR\test>node index.js
[ 1, 2, 5, 6, 6 ]
```

Sortowanie przez wybieranie – złożoność O(n^2)

Algorytm:

1. Szukamy minimum na pozycjach od i+1 do końca i zamieniamy z pozycją i jeżeli na pozycji i znajduje się większy element
2. Zwiększamy i o 1

Dane: [6,1,5,6,2]

Na czerwono część zbioru, w której szukamy minimum:

i = 0	6 1 5 6 2	min=1	zamiana 6 z 1
i = 1	1 6 5 6 2	min=2	zamiana 6 z 2
i = 2	1 2 5 6 6	min=6	nie zmieniamy bo min >= 5
i = 3	1 2 5 6 6	min=6	nie zmieniamy bo min >= 6

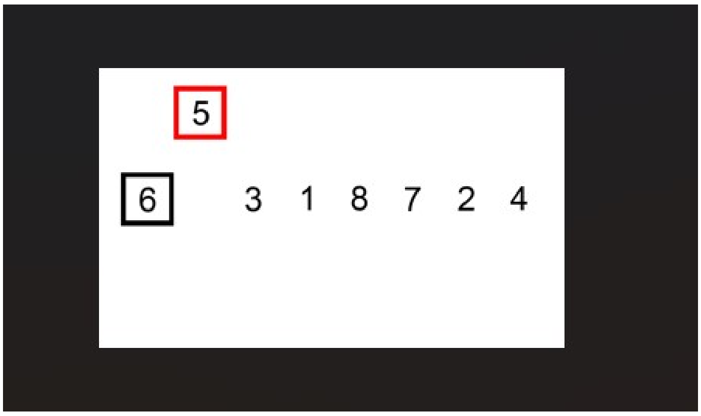
Implementacja:

```
function select(data){
    for(let i = 0;i<data.length;i++){
        //find min
        let min = 9999;
        let min_i = i+1; //minimum index
        for(let j = i;j<data.length;j++){
            if(data[j] < min){
                min = data[j];
                min_i = j;
            }
        }

        let t = data[i];
        data[i] = data[min_i];
        data[min_i] = t;
    }
}

let data = [6,1,5,6,2];
select(data);
console.log(data);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER .NET INTERACTIVE

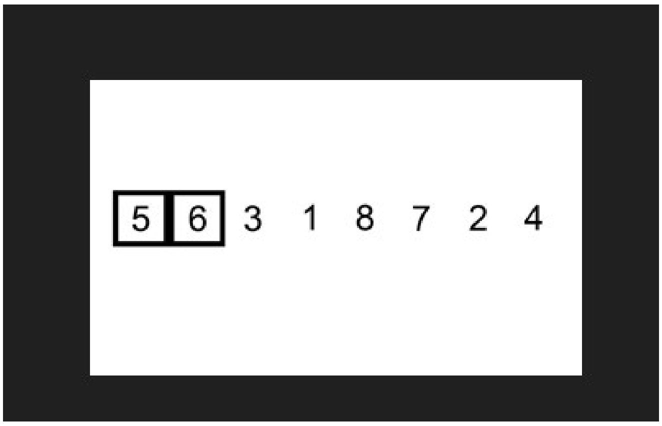
```
C:\Users\siedl\Desktop\PR\test>node index.js
[ 1, 2, 5, 6, 6 ]
```

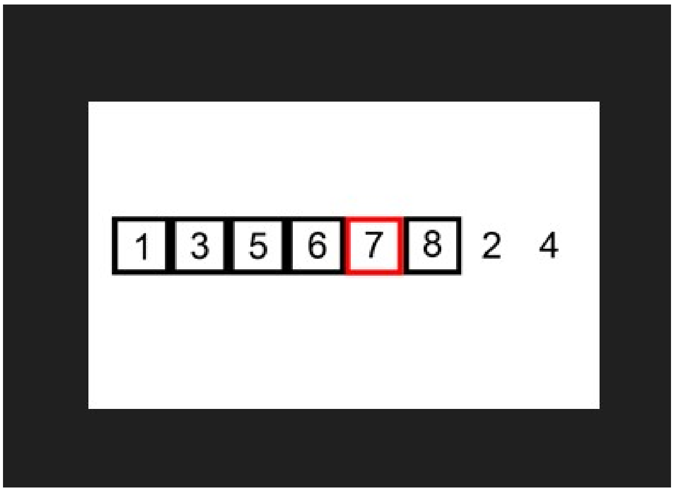
Sortownie przez wstawianie – złożoność O(n^2)



Dzielimy zbiór na dwa podzbiory. Zbiór elementów posortowanych (tylko pierwszy element) oraz zbiór elementów nie posortowanych (cała reszta).

Wybieramy pierwszy element ze zbioru nie posortowanego i porównujemy z elementami zbioru posortowanego idąc od prawej strony. Wstawiamy w odpowiednie miejsce zbioru posortowanego przesuwając elementy dalej.





Implementacja:

```javascript
function insert(data){
    for(let i = 1;i<data.length;i++){
        let j = i-1; //ostatni index zbioru posortowanego
        let temp = data[i]; //pierwszy elemenent zbioru posortowanego

        //porównywanie od prawej i przesuwanie
        while(data[j]>temp && j>=0){
            data[j+1] = data[j]; //przesuwanie elementów w prawo,zeby zrobić miejsce na wstawi
            j--;
        }
        data[j+1] = temp; //wstawienie
    }
}

let data = [6,1,5,6,2];
insert(data);
console.log(data);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER .NET INTERACTIVE node

```
C:\Users\siedl\Desktop\PR\test>node index.js
[ 1, 2, 5, 6, 6 ]
```

Sortowanie szybkie – złożoność (n log n):

Algorytm typu dziel i zwyciężaj:

Zbiór do posortowania:

2 5 1 3 4 0 6 2 5

Teraz musimy wybrać środkowy element zwany pivotem. Według niego podzielimy tablicę na dwie części.

lewa część tablicy				**pivot**	prawa część tablicy			
2	5	1	3	**4**	0	6	2	5

Ustawiamy wskaźnik i na początku tablicy po lewej stronie. Drugi wskaźnik j ustawiamy na końcu tablicy po prawej stronie.

lewa część tablicy				**pivot**	prawa część tablicy			
2	5	1	3	**4**	0	6	2	5
i								**j**

Teraz przesuwamy się oboma wskaźnikami w stronę pivota do momentu minięcia się przez nie. Wskaźnik i szuka elementów większych od pivota a wskaźnik j szuka elementów mniejszych od pivota. Jeżeli mamy parę zamieniamy te liczby miejcami.

lewa część tablicy				**pivot**	prawa część tablicy			
2	5	1	3	**4**	0	6	2	5
	i						**j**	

lewa część tablicy				**pivot**	prawa część tablicy			
2	2	1	3	**4**	0	6	5	5
	i						**j**	

Robimy tak do momentu minięcia się.

lewa część tablicy		**pivot**	prawa część tablicy	
2	2 1 3 0	**4**	6 5	5
lewy	**j**	**i**		**prawy**

Teraz następuje słynne dziel i zwyciężaj. Te same kroki wykonujemy dla dwóch podtablic [0 do j] oraz [i do końca]. W każdej z nich wybieramy pivota i postępujemy analogicznie. Na końcu dojdzie do kolejnego podziału itd. itd.

Implementacja:

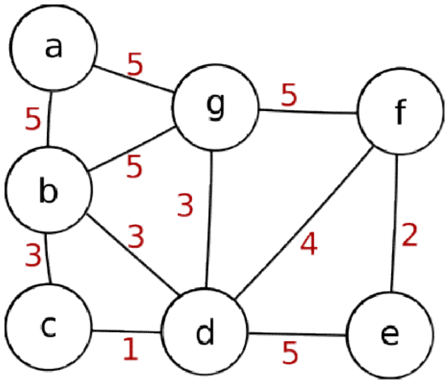
```
function split(data, begin, end){
    let temp;
    let pivot = data[end]; //tymczasowo pivotem jest ostatni element
    let j = begin;
    for(let i = begin;i<=end;i++){
        if(data[i] < pivot){
            //zamiana
            temp = data[i];
            data[i] = data[j];
            data[j] = temp;
            j++;
        }
    }
    temp = data[j];
    data[j] = data[end];
    data[end] = temp;
    return j;
}

function quick(data, begin, end){
    if(begin < end){    //zabezpiecza przez przepełnieniem
        let pivot = split(data, begin, end);
        quick(data, begin, pivot-1); //sortowanie lewej
        quick(data, pivot+1, end); //sortowanie prawej rekurencja
    }
}

let data = [6,1,5,6,2];
quick(data, 0, data.length-1);
console.log(data);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER .NET INTERACTIVE

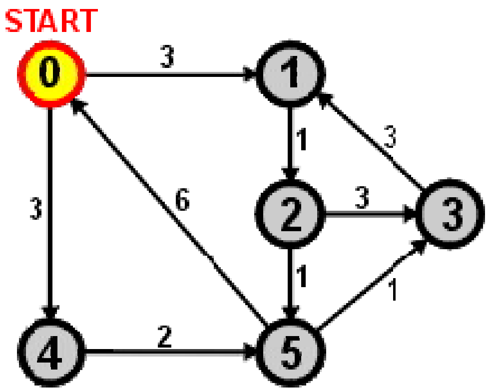
```
C:\Users\siedl\Desktop\PR\test>node index.js
[ 1, 2, 5, 6, 6 ]
```

Graf

Graf to zbiór wierzchołków połączonych krawędziami. Można sobie to wyobrazić, jak miasta na mapie połączone drogami. Każda krawędź (droga) ma określoną długość (docelowo nazywa się to wagą). **Wagi mogą być ujemne, więc analogia do drogi nie zawsze się sprawdza.**

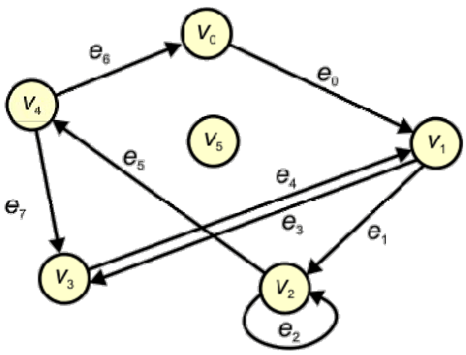


Graf nieskierowany



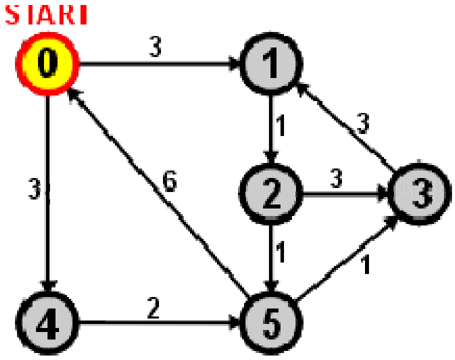
Graf skierowany

Graf skierowany posiada krawędzie, które prowadzą w konkretnym kierunku. W grafie nieskierowanym każda krawędź może prowadzić w dwie strony.



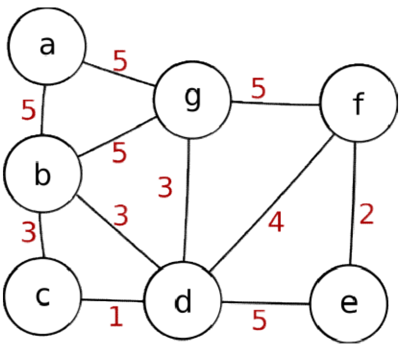
Możliwe są krawędzi prowadzące do samego siebie.

Jak zapisać poniższy graf w programie? Jako macierz czyli tablicę dwu wymiarową. Indeks wiersza to wierzchołek, z którego wychodzimy a indeks kolumny to wierzchołek docelowy:



	0	1	2	3	4	5
0	-1	3	-1	-1	3	-1
1	-1	-1	1	-1	-1	-1
2	-1	-1	-1	3	-1	1
3	-1	3	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	2
5	6	-1	-1	1	-1	-1

Liczba -1 oznacza brak krawędzi. Jak widać na obrazku, żaden wierzchołek nie posiada krawędzi do samego siebie, więc po przekątnej wszędzie znajduje się -1. **W programowaniu to jakiej liczby użyjemy do oznaczenia sobie braku krawędzi zależy od algorytmu. W grafach mogą występować krawędzie ujemne.**



	0 - a	1 - b	2 - c	3 - d	4 -e	5 - f	6 - g
0 – a	-1	5	-1	-1	-1	-1	5
1 – b	5	-1	3	3	-1	-1	5
2 – c	-1	3	-1	1	-1	-1	-1
3 – d	-1	3	1	-1	5	4	3
4 – e	-1	-1	-1	5	-1	2	-1
5 – f	-1	-1	-1	4	2	-1	5
6 - g	5	5	-1	3	-1	5	-1

Stopień wierzchołka – ilość krawędzi wchodzących i wychodzących z wierzchołka.

Rząd grafu – ilość wierzchołków w grafie

Cykl – jest to po prostu koło wykonane na krawędziach

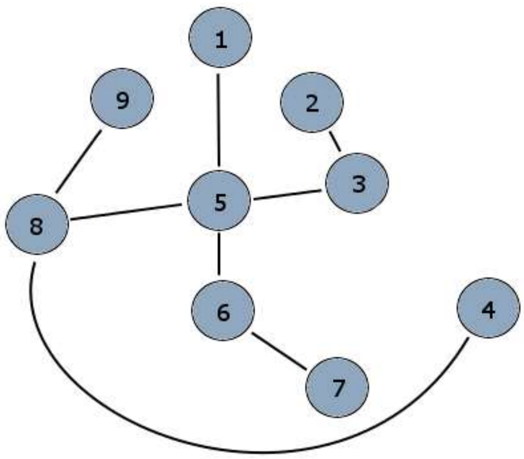
Cykl Hamiltona – wyjście z dowolnego wierzchołka, przejście po każdym wierzchołku dokładnie **RAZ** oraz powrót do wierzchołka startowego

Problem Komiwojazera – jest to cykl Hamiltona, ale łącza długość trasy musi być najmniejsza możliwa (trzeba znaleźć najmniejszy cykl)

Cykl Eulera – wyjście z dowolnego wierzchołka, przejście po każdej krawędzi dokładnie **RAZ** oraz powrót do wierzchołka startowego

Droga – po prostu droga między dwoma wierzchołkami

Drzewo – to graf, w którym między dowolnymi dwoma wierzchołkami istnieje jedna droga



Minimalne drzewo rozpinające – drzewo jak wyżej, ale suma krawędzi musi być jak najmniejsza

Klika – taki graf, w którym każdy wierzchołek ma połączenie z każdym innym

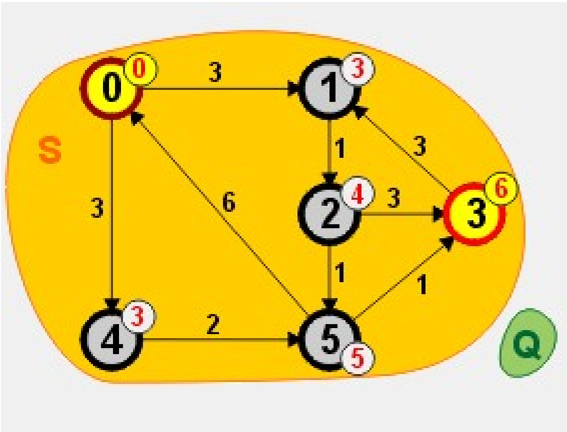
Problem szukania odległości i drogi między wierzchołkami: Algorytm Dijkstry, Algorytm Bellmana-Forda

Mamy dowolny graf. Chcemy znaleźć najkrótszą drogę między dowolnymi wierzchołkami grafu. Możemy użyć do tego Algorytmu Dijkstry (tylko jeżeli nie ma ujemnych krawędzi) lub Algorytmu Bellmana Forda (pozwala na ujemne krawędzie, ale nie mogą one tworzyć ujemnego cyklu, ponieważ wtedy każdą drogę da się skrócić robiąc parę okrążeń po ujemnym cyklu).

Oba algorytmy generują dwie tablice:

- tablice dystansu

- tablice poprzedników



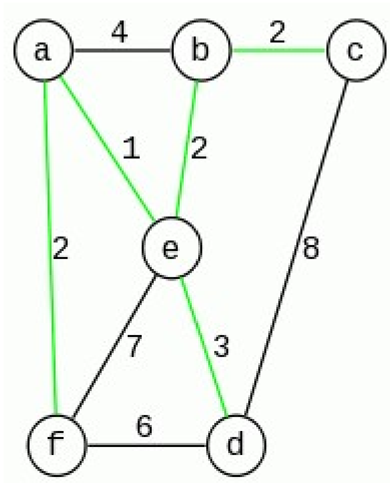
u	0	1	2	3	4	5
$d[u]$	0	3	4	6	3	5
$p[u]$	-1	0	1	5	0	4

Tablica dystansu zawiera dystans od wybranego wierzchołka do każdego innego. Powyżej startowaliśmy z wierzchołka 0. Najkrótsza droga do wierzchołka 3 to 6. Najkrótsza droga z 0 do 5 to 5.

Tablica poprzedników zawiera poprzednika na drodze do danego wierzchołka. To trochę trudno zrozumieć. Ponownie startujemy z 0. Aby dojść do 3 trzeba przejść przez 5 (tablica p), żeby dojść do 5 trzeba przejść przez 4. Żeby dojść do 4 nie trzeba robić nic bo to sąsiad. Z tablicy p odczytaliśmy, że do 3 można dojść z 0 poprzez 4 i 5.

Problem szukania minimalnego drzewa: Algorytm Kruskala, Algorytm Prima

Mamy graf:



Posiada on 9 krawędzi. Szukanie minimalnego drzewa rozpinającego polega na znalezieniu drzewa, którego suma krawędzi jest najmniejsza.

Najlepiej sobie to wyobrazić jako miasta gdzieś na świecie. Chcemy je połączyć jak najmniejszą ilością drogi (czy tam asfaltu).

Powyżej wynik poszukiwania zaznaczono na zielono.

Oba algorytmy działają tak, że sortują krawędzie od najmniejszej do największej. A później dobierają idąc od najmniejszej do momentu połączenia wszystkich wierzchołków w drzewo. Jeżeli jakaś krawędź tworzy cykl to jest odrzucana. W drzewie nie ma naturalnie cykli.

Problem Komiwojazera

Problem ten wziął się z problemu zwiedzenia przez handlarza wszystkich miast w stanach i powrocie do domu jak najmniejszym kosztem. Każde miasto musi być odwiedzone i tylko raz. Suma drogi musi być jak najmniejsza.

Jest to bardzo trudny problem. Algorytmy do niego są mega skomplikowane. Wytłumaczenie ich jest jaki pisanie książki.

Lista kroków:

K01: S_H.push (v)	*Odwiedzony wierzchołek dopisujemy do ścieżki*
K02: **Jeśli** S_H nie zawiera n wierzchołków, **to idź do** kroku K10	*Jeśli brak ścieżki Hamiltona, przechodzimy do wyszukiwania*
K03: **Jeśli** nie istnieje krawędź z v do v_0, **to idź do** kroku K17	*Jeśli ścieżka Hamiltona nie jest cyklem, odrzucamy ją*
K04: $d_H \leftarrow d_H$ + waga krawędzi z v do v_0	*Uwzględniamy w sumie wagę ostatniej krawędzi cyklu*
K05: **Jeśli** $d_H \ge d$, **to idź do** kroku K08	*Jeśli znaleziony cykl jest gorszy od bieżącego, odrzucamy go*
K06: $d \leftarrow d_H$	*Zapamiętujemy sumę wag cyklu*
K07: Skopiuj stos S_H do stosu S	*oraz sam cykl Hamiltona*
K08: $d_H \leftarrow d_H$ - waga krawędzi z v do v_0	*Usuwamy wagę ostatniej krawędzi z sumy*
K09: **Idź do** kroku K17	
K10: *visited*[v] ← true	*Wierzchołek zaznaczamy jako odwiedzony, aby nie był ponownie wybierany przez DFS*
K11: **Dla** każdego sąsiada u wierzchołka v: **wykonuj** kroki K12...K15	*Przechodzimy przez listę sąsiedztwa*
K12: **Jeśli** *visited*[u] = true, **to następny obieg pętli** K11	*Omijamy wierzchołki odwiedzone*
K13: $d_H \leftarrow d_H$ + waga krawędzi z v do u	*Obliczamy nową sumę wag krawędzi ścieżki*
K14: TSP (n, *graf*, u, v_0, d, d_H, S, S_H, *visited*)	*Wywołujemy rekurencyjnie poszukiwanie cyklu*
K15: $d_H \leftarrow d_H$ - waga krawędzi z v do u	*Usuwamy wagę krawędzi z sumy*
K16: *visited*[v] ← false	*Zwalniamy bieżący wierzchołek*
K17: S_H.pop( )	*Usuwamy bieżący wierzchołek ze ścieżki*
K18: **Zakończ**	

Lista kroków algorytmu głównego

K01: Utwórz i wyzeruj *visited*, S i S_H	
K02: $d \leftarrow \infty$; $d_H \leftarrow 0$	*Początkową sumę ustawiamy jako największą z możliwych*
K03: TSP (v_0)	*Wyszukiwanie cyklu Hamiltona rozpoczynamy od wierzchołka startowego*
K04: **Jeśli** S.empty( ) = false, **to pisz** S oraz d **inaczej pisz** "NO HAMILTONIAN CYCLE"	*Sprawdzamy, czy algorytm znalazł cykl Hamiltona. Jeśli tak, to wypisujemy go.*
K05: **Zakończ**	