Algorytmy i struktury danych

wykład 7

Plan wykładu:

- Algorytmy grafowe II.
- Algorytmy sortujące.

Algorytmy grafowe II

Problem najkrótszej ścieżki – polega na znalezieniu w grafie ważonym najkrótszego połączenia między dwoma zadanymi węzłami.

Uwagi:

- jest to problem klasy NP,
- rozwiązanie tego problemu pozwala na:
  - znalezienie najkrótszej drogi między miastami,
  - efektywne trasowanie ruchu w sieciach komputerowych,
  - wyznaczanie optymalne rozkładu pewnych elementów w czasie lub przestrzeni.
- najkrótszą ścieżkę w grafie wyznacza się dla wszystkich typów grafów.

Algorytmy stosowane dla dwóch węzłów:

- algorytm Dijkstry,
- algorytm Bellmana-Forda,
- algorytm A*.

Algorytmy stosowane dla każdej pary węzłów:

- algorytm Floyda-Warshalla,
- algorytm Johnsona.

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Oznaczenia:

- *s* – węzeł źródłowy,
- w(*i*,*j*) – waga krawędzi (*i*,*j*).

Algorytm:

1. Stworzenie tablicy d odległości od źródła dla wszystkich węzłów grafu. Na początku d[*s*] = 0, d[*u*] = w(*s*,*u*), natomiast *u* jest sąsiadem *s*, a d[*v*] = nieskończoność dla wszystkich pozostałych węzłów.
2. Utworzenie kolejki priorytetowej *Q* wszystkich węzłów grafu. Priorytetem kolejki jest aktualnie wyliczona odległość od węzła źródłowego *s*.
3. Dopóki kolejka nie jest pusta:
   - usunięcie z kolejki węzła *u* o najniższym priorytecie (węzeł najbliższy źródła, który nie został jeszcze rozważony),
   - dla każdego sąsiada *v* i węzła *u*: jeśli d[*u*] + w(*u*,*v*) < d[*v*] (poprzez *u* da się dojść do *v* szybciej niż dotychczasową ścieżką), to d[*v*]: = d[*u*] + w(*u*,*v*).
4. Na końcu tablica d zawiera najkrótsze odległości do wszystkich węzłów.

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

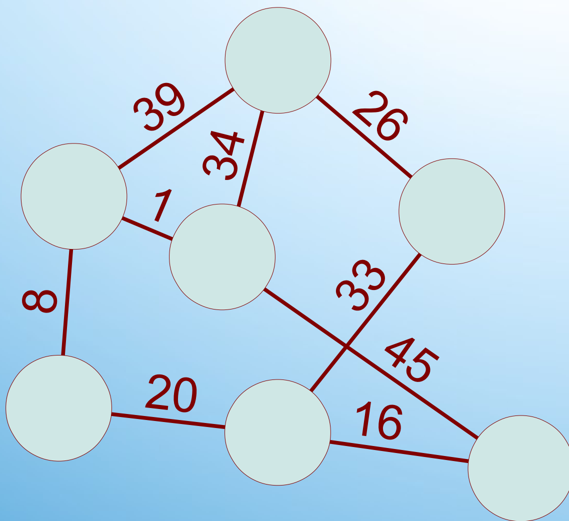
Oznaczenia:
- *s* – węzeł źródłowy,
- w(*i*,*j*) – waga krawędzi (*i*,*j*).

Algorytm:
1. Stworzenie tablicy d odległości od źródła dla wszystkich węzłów grafu. Na początku d[*s*] = 0, d[*u*] = w(*s*,*u*), natomiast *u* jest sąsiadem *s*, a d[*v*] = nieskończoność dla wszystkich pozostałych węzłów.
2. Utworzenie kolejki priorytetowej *Q* wszystkich węzłów grafu. Priorytetem kolejki jest aktualnie wyliczona odległość od węzła źródłowego *s*.
3. Dopóki kolejka nie jest pusta:
   - usunięcie z kolejki węzła *u* o najniższym priorytecie (węzeł najbliższy źródła, który nie został jeszcze rozważony),
   - dla każdego sąsiada *v* i węzła *u*: jeśli d[*u*] + w(*u*,*v*) < d[*v*] (poprzez *u* da się dojść do *v* szybciej niż dotychczasową ścieżką), to d[*v*]: = d[*u*] + w(*u*,*v*).
4. Na końcu tablica d zawiera najkrótsze odległości do wszystkich węzłów.

Uwagi:
- złożoność obliczeniowa pesymistyczna wynosi $O(|V|^2)$,
- złożoność obliczeniowa w przypadku implementacji kolejki *Q* w postaci kopca wynosi O(|E| log |V|).

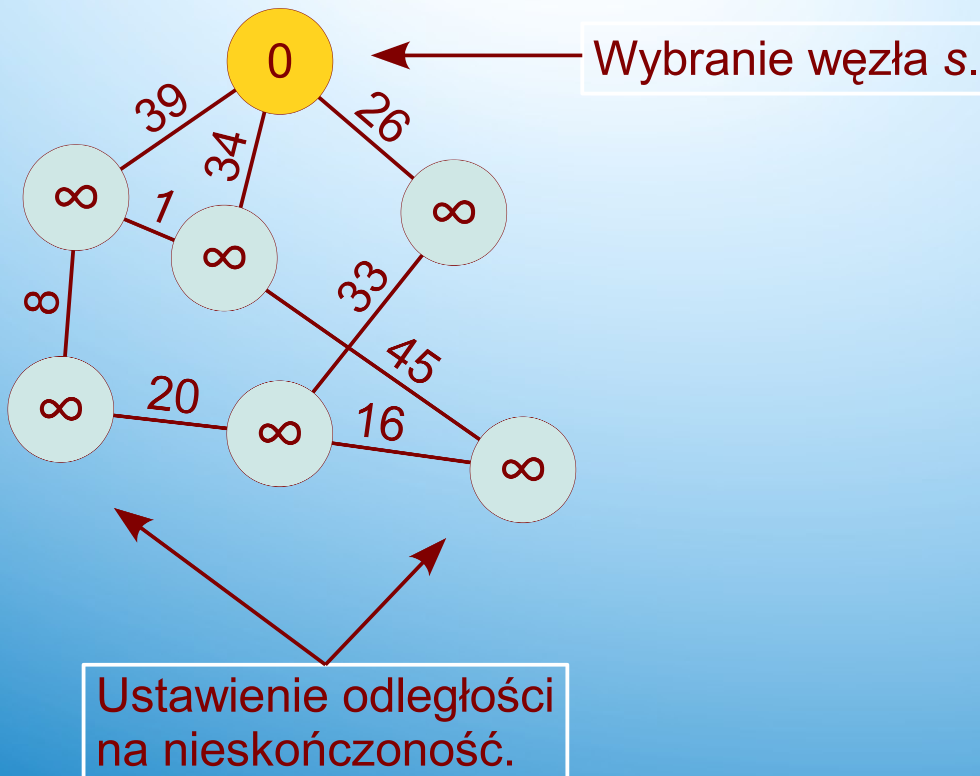
Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



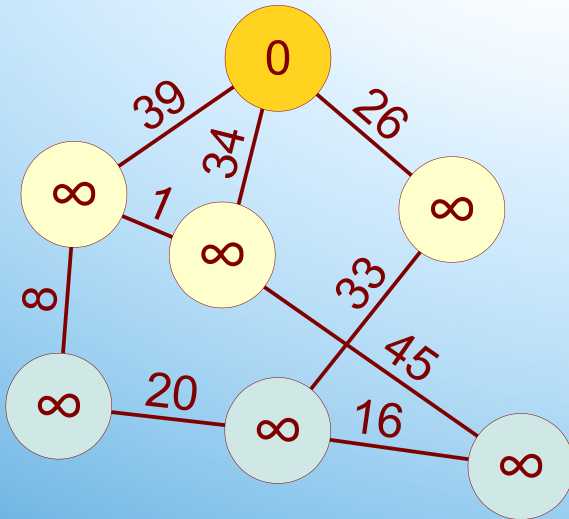
Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

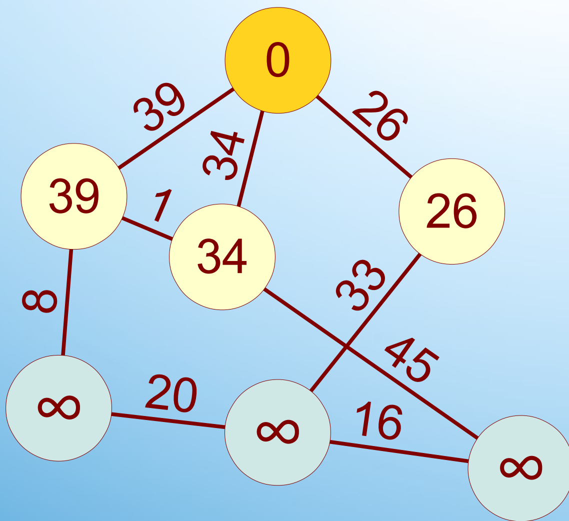
Przykład:



Etap 1:
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

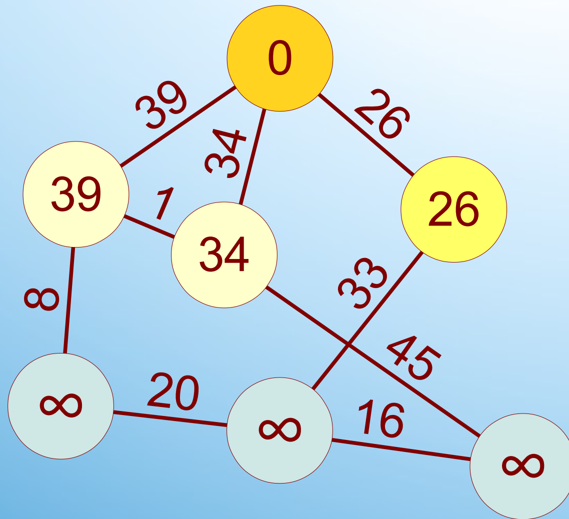
Przykład:



Etap 1:
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

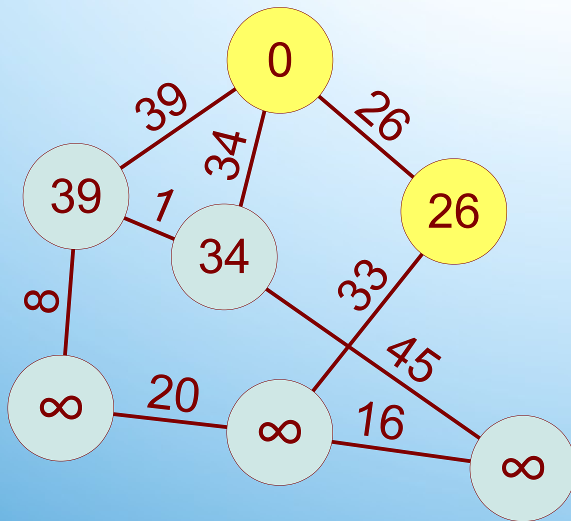
Przykład:

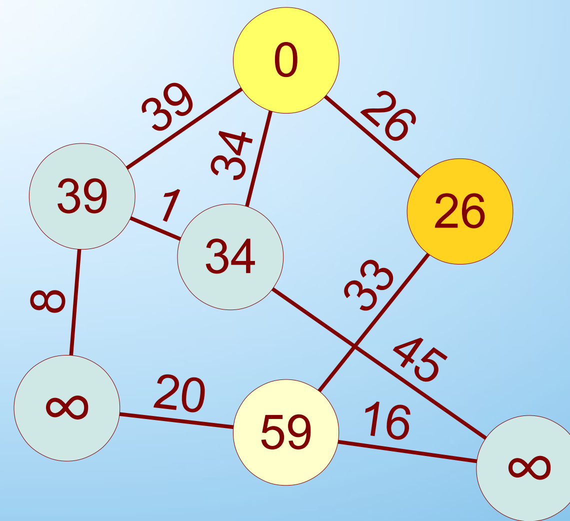


Etap 1:
ustalenie węzła

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



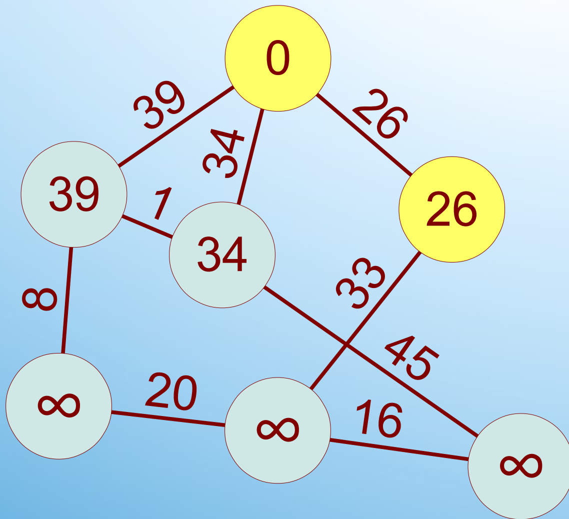


Etap 1:

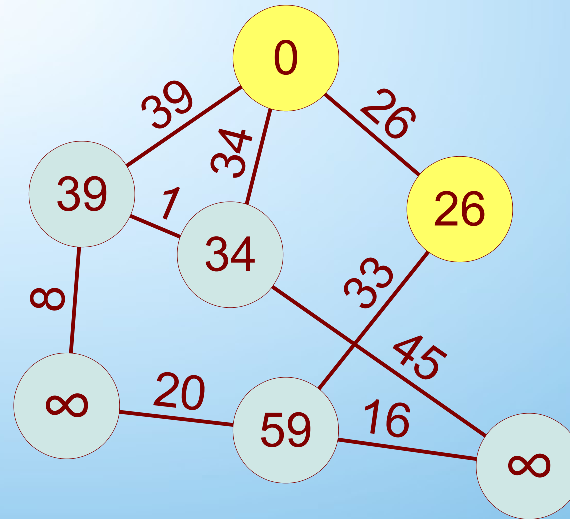
Etap 2:
wybranie węzła
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



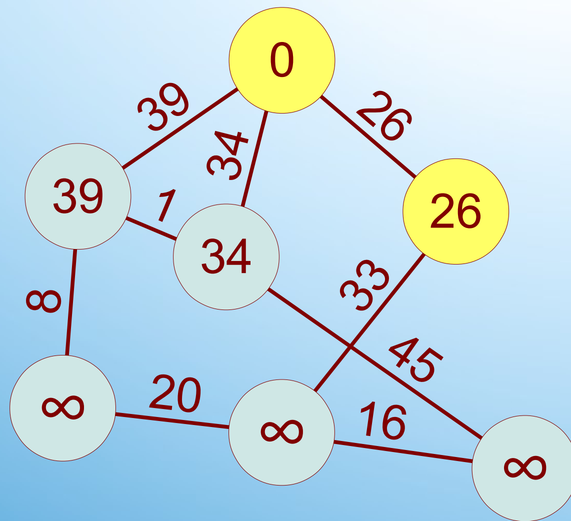
Etap 1:

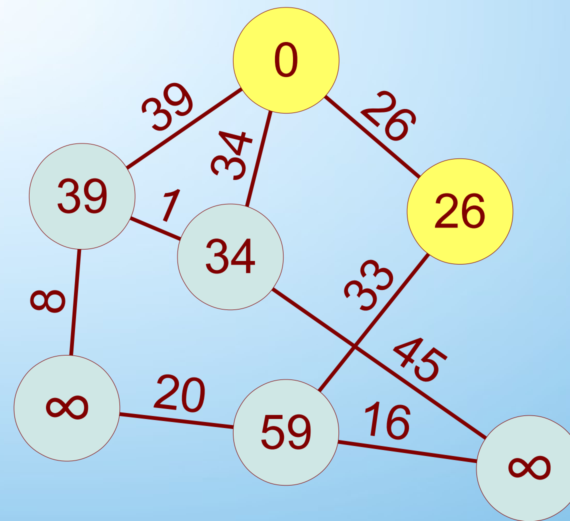


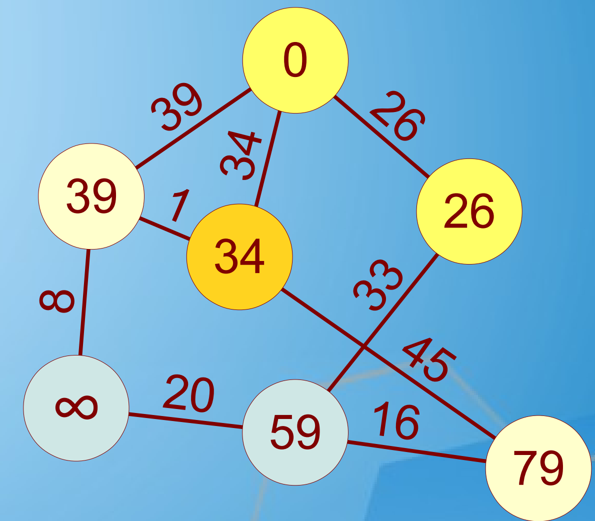
Etap 2:
ustalenie węzła

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:







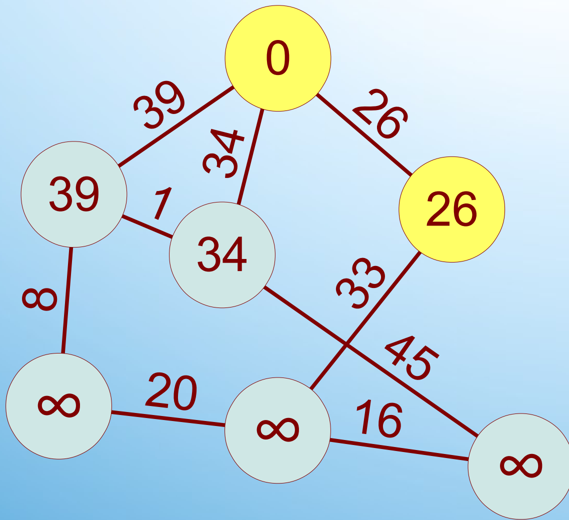
Etap 1:

Etap 2:

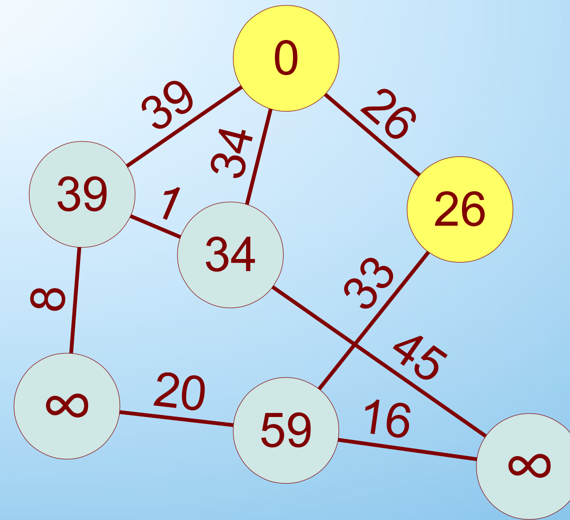
Etap 3:
wybranie węzła
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

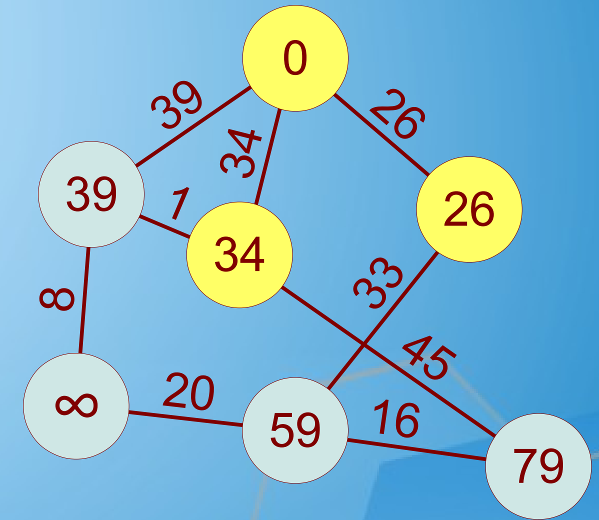
Przykład:



Etap 1:



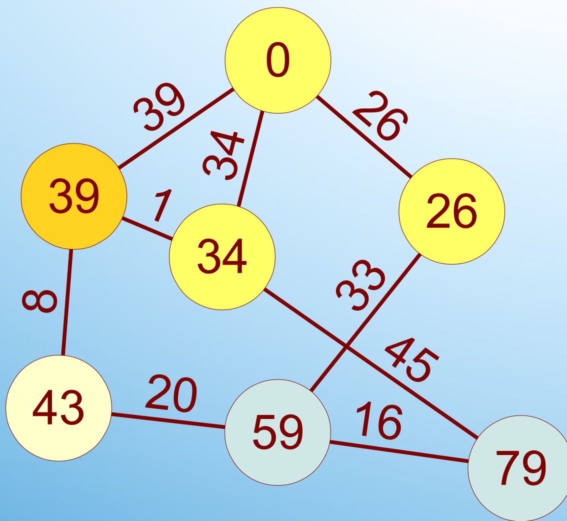
Etap 2:



Etap 3:
ustalenie węzła

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

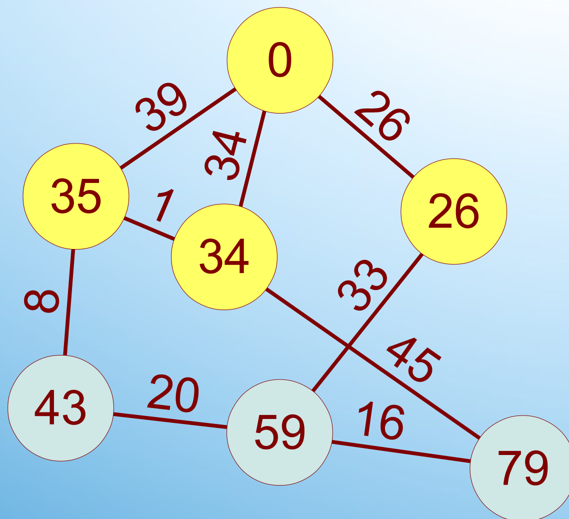
Przykład:



Etap 4:
wybranie węzła
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:

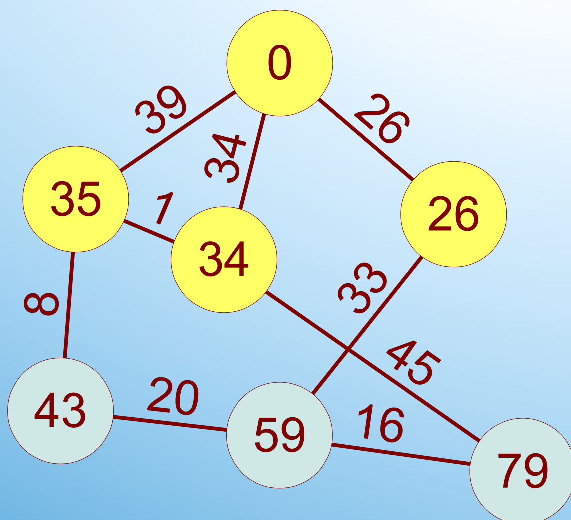


Etap 4:

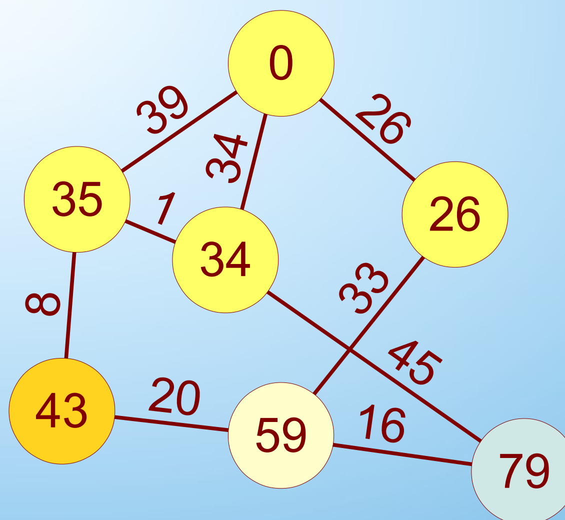
ustalenie węzła: 39 → 35

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



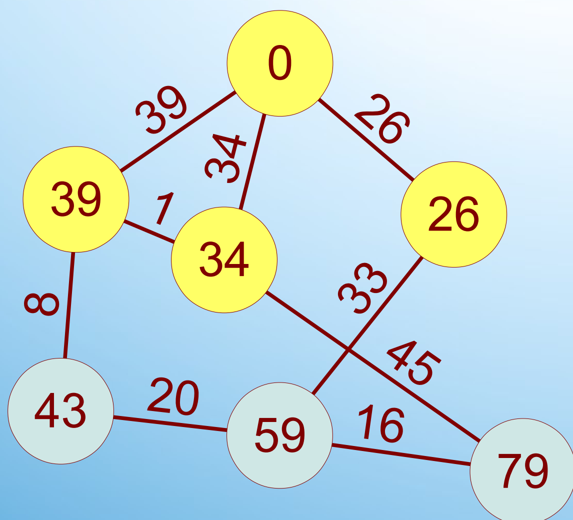
Etap 4:



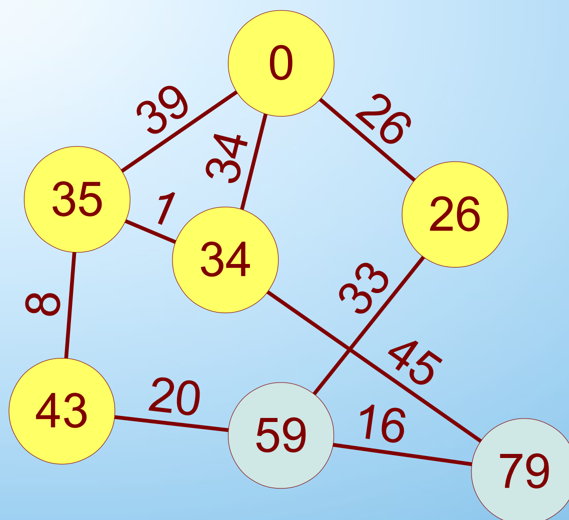
Etap 5:
wybranie węzła
sprawdzenie sąsiada: 43 + 20 > 59

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:



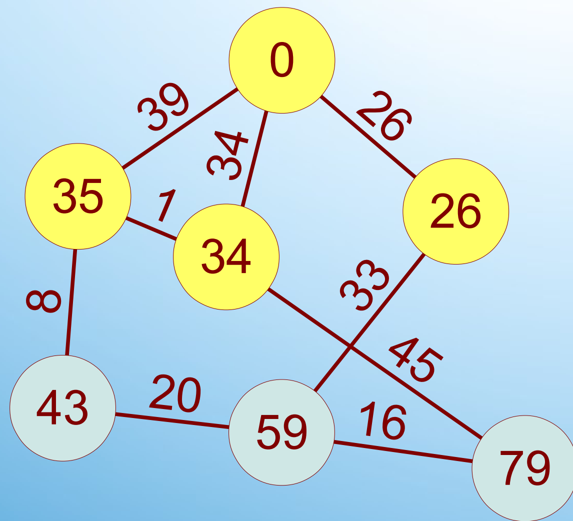
Etap 4:



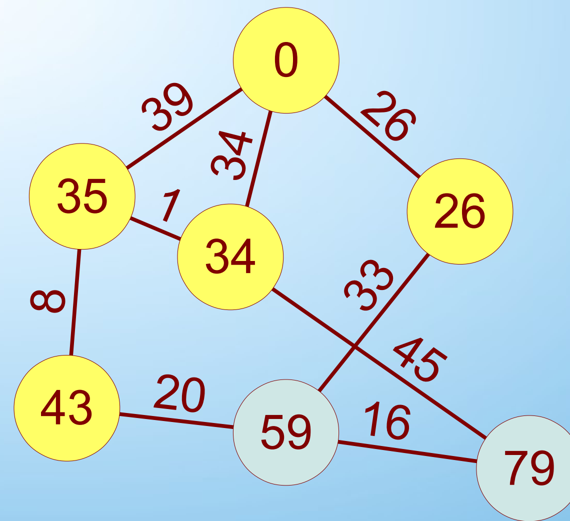
Etap 5:
ustalenie węzła

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

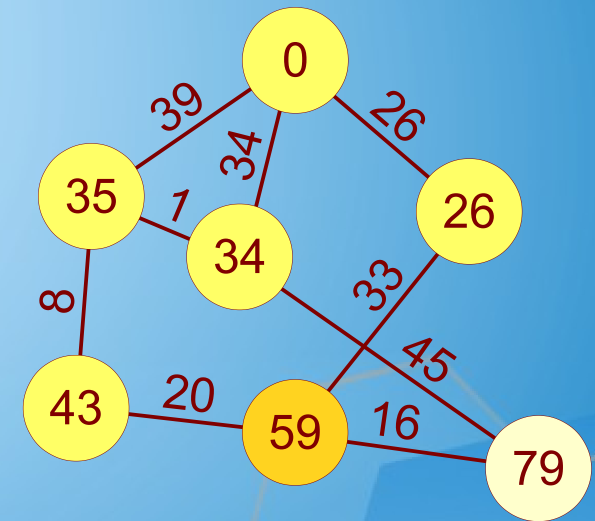
Przykład:



Etap 4:



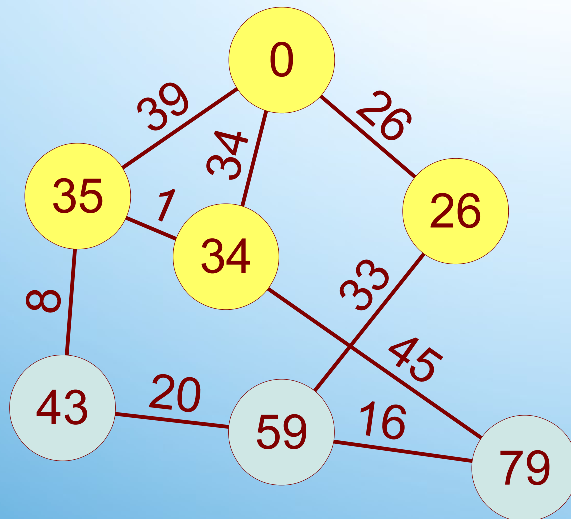
Etap 5:

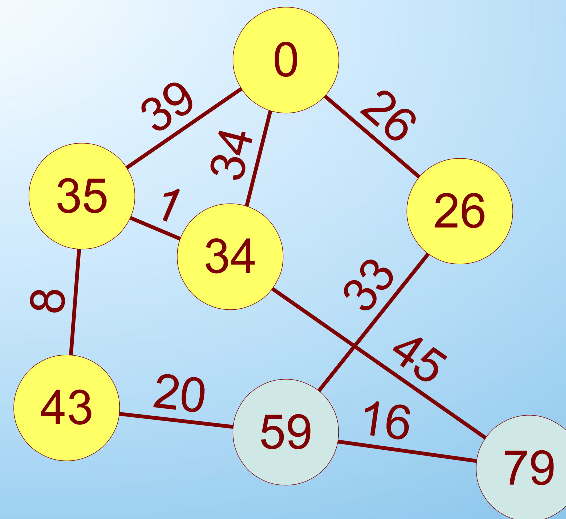


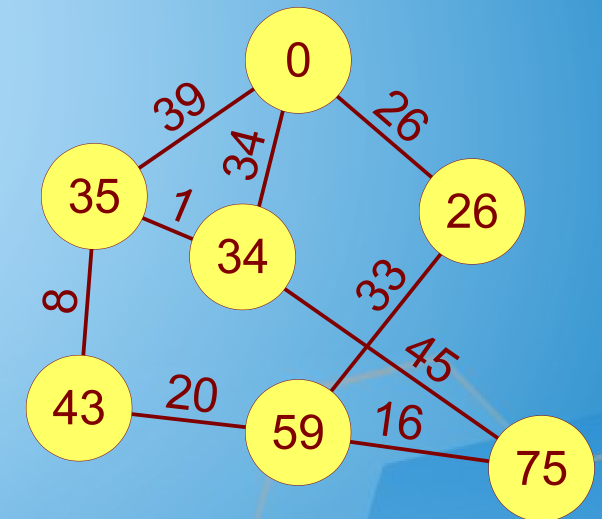
Etap 6:
wybranie węzła
sprawdzenie sąsiadów

Algorytm Dijkstry – znajduje najkrótszą ścieżkę z pojedynczego źródła w grafie o nieujemnych wagach krawędzi.

Przykład:







Etap 4:

Etap 5:

Etap 6:
ustalenie węzła: 79 → 75

Algorytm Bellmana-Forda – służy do znajdowania najkrótszej ścieżki między dwoma węzłami w grafie ważonym.

Oznaczenia:

- *s* – węzeł źródłowy,
- v – węzeł docelowy,
- w(*u*,*v*) – waga krawędzi (*u*,*v*).

Algorytm:

1. Obliczenie górnego oszacowanie d[*v*] odległości od węzła *s* do pozostałych węzłów *v*.
2. Jeśli d[*v*] > d[*u*] + w(*u*,*v*) to przeszacowanie odległości d[*v*] = d[*v*] + w(*u*.*v*).
3. Jeśli nie można przeszacować żadnej odległości to KONIEC.

Algorytm Bellmana-Forda – służy do znajdowania najkrótszej ścieżki między dwoma węzłami w grafie ważonym.

Oznaczenia:
- *s* – węzeł źródłowy,
- v – węzeł docelowy,
- w(*u*,*v*) – waga krawędzi (*u*,*v*).

Algorytm:
1. Obliczenie górnego oszacowanie d[*v*] odległości od węzła *s* do pozostałych węzłów *v*.
2. Jeśli d[*v*] > d[*u*] + w(*u*,*v*) to przeszacowanie odległości d[*v*] = d[*v*] + w(*u*.*v*).
3. Jeśli nie można przeszacować żadnej odległości to KONIEC.

Uwagi:
- wagi mogą być dodatnie i ujemne,
- złożoność obliczeniowa wynosi O(|V|*|E|).

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Cechy:
- algorytm optymalizuje ścieżkę wykorzystując funkcję heurystyczną $h(x)$,
- w każdym przypadku algorytm znajdzie drogę (jeśli taka istnieje).

Algorytm:
1. Wybranie sąsiednich, jeszcze nie sprawdzanych węzłów i obliczenie funkcji $f(x) = g(x) + h(x)$, gdzie:
   - $g(x)$ – droga pomiędzy węzłem początkowym a x (suma wag krawędzi, które należą już do ścieżki, powiększona o wagę krawędzi łączącej aktualny węzeł z x),
   - $h(x)$ – przewidywana przez heurystykę droga od x do węzła docelowego.
2. Jeśli $f(x) < f_{min}$ to dodanie węzła do ścieżki, $f_{min} = f(x)$.
3. Jeśli x nie jest węzłem docelowym to powtarzaj od etapu 1.

$\leqslant$

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Cechy:
- algorytm optymalizuje ścieżkę wykorzystując funkcję heurystyczną $h(x)$,
- w każdym przypadku algorytm znajdzie drogę (jeśli taka istnieje).

Algorytm:
1. Wybranie sąsiednich, jeszcze nie sprawdzanych węzłów i obliczenie funkcji $f(x) = g(x) + h(x)$, gdzie:
   - $g(x)$ – droga pomiędzy węzłem początkowym a x (suma wag krawędzi, które należą już do ścieżki, powiększona o wagę krawędzi łączącej aktualny węzeł z x),
   - $h(x)$ – przewidywana przez heurystykę droga od x do węzła docelowego.
2. Jeśli $f(x) < f_{min}$ to dodanie węzła do ścieżki, $f_{min} = f(x)$.
3. Jeśli x nie jest węzłem docelowym to powtarzaj od etapu 1.

Uwagi:
- funkcja heurystyczna h jest poprawna, jeśli dla dwóch dowolnych węzłów x i y zachodzi zależność: $h(x) \leqslant d(x,y) + h(y)$, gdzie $d(x,y)$ jest faktyczną odległością między węzłami x i y,
- złożoność czasowa algorytmu jest zależna od zastosowanej heurystyki i wynosi $O(\log h^*(x))$, gdzie h^* jest optymalną heurystyką, czyli zawsze podaje dokładny, rzeczywisty koszt ścieżki z x do węzła końcowego.

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:

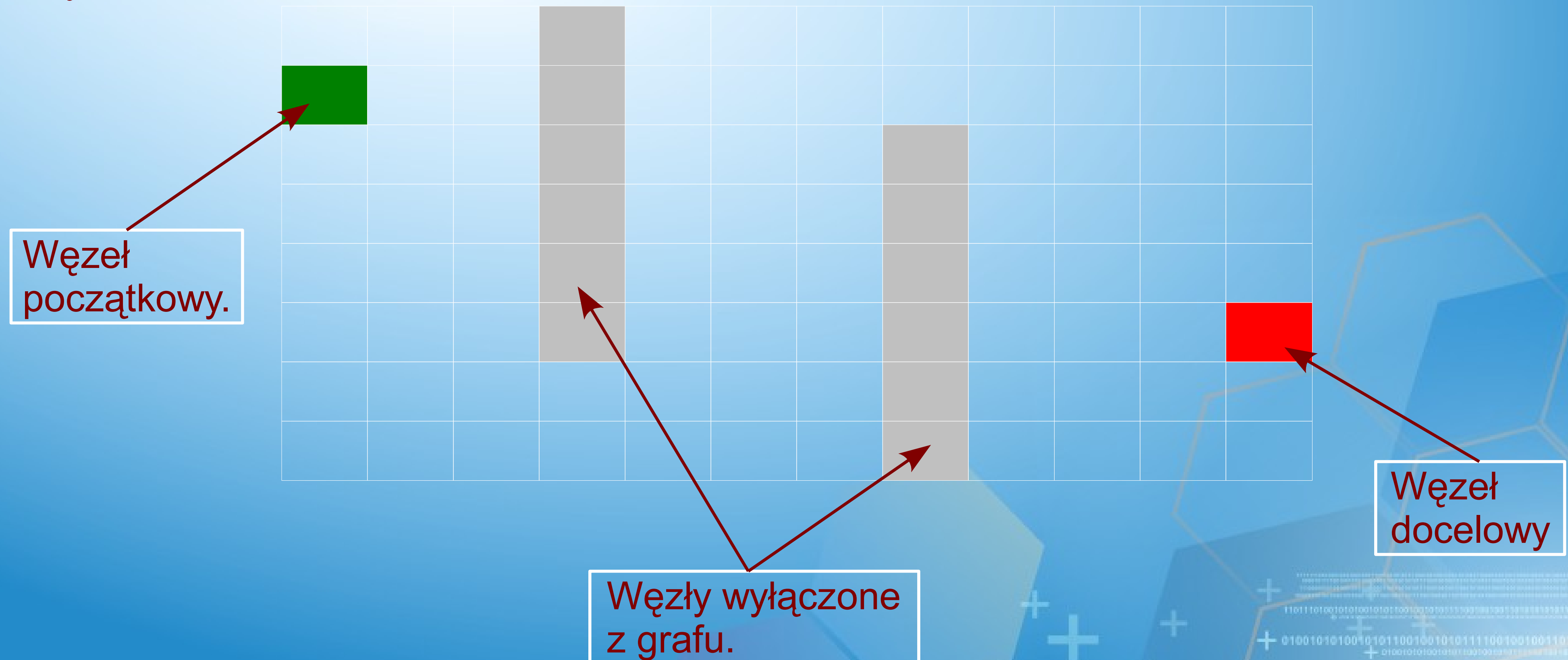
- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:

- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:

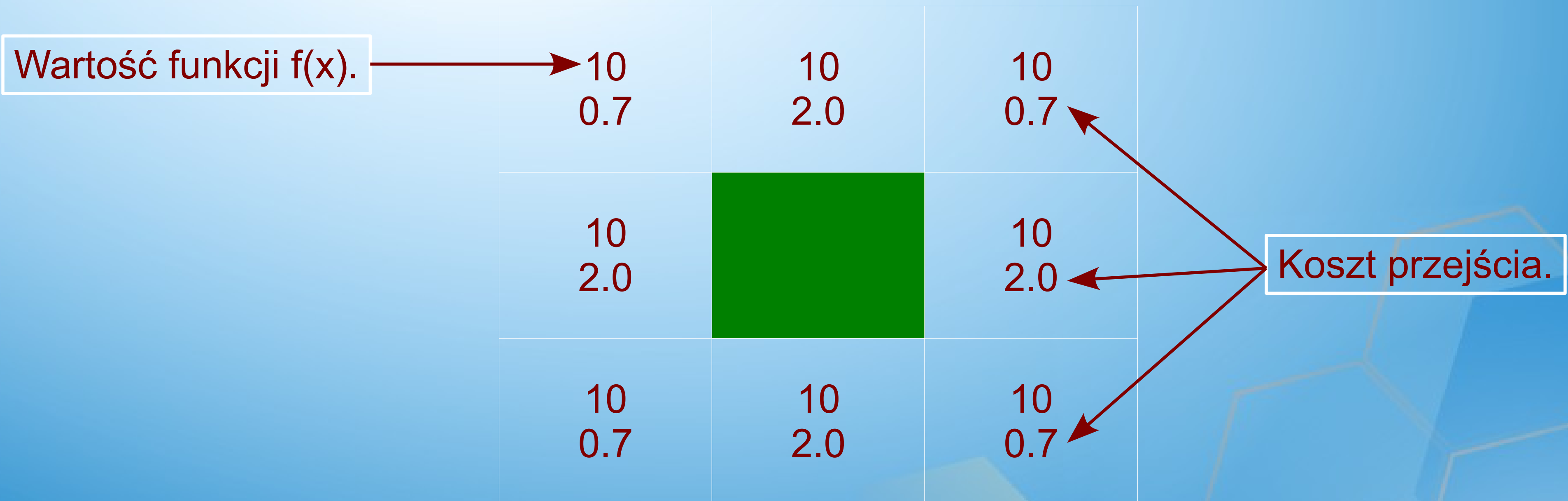


Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:

- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:

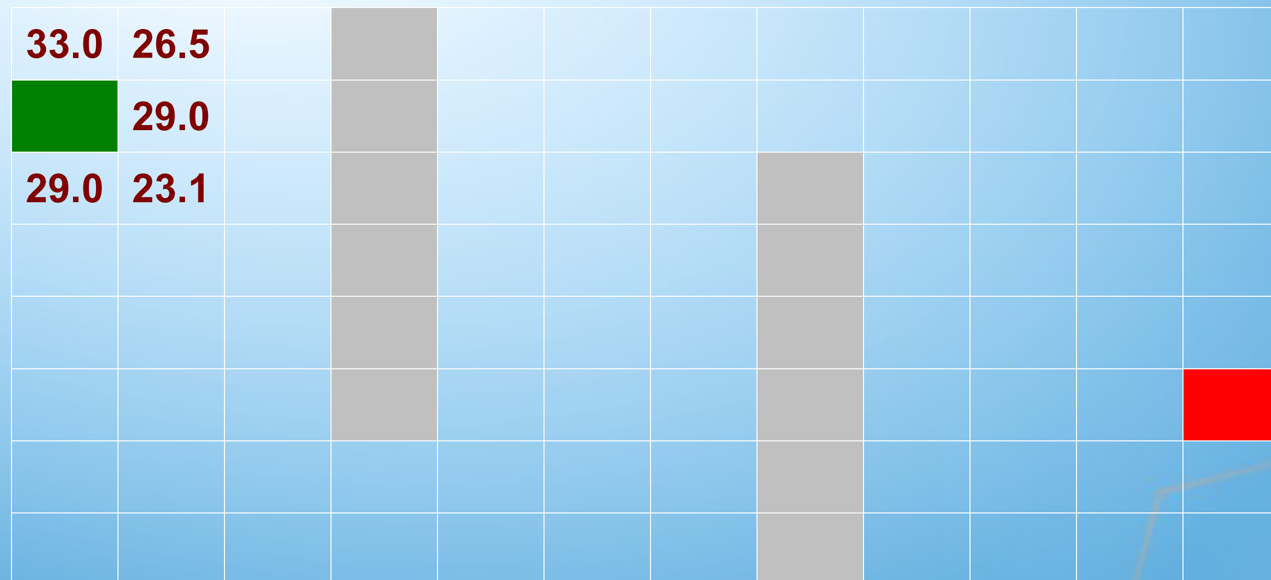


f(x) = liczba węzłów do *x* + długość ścieżki + (1 + koszt przejścia) · $h_{manhattan}$

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:
- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:



Etap 1

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:

- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:

33.0	26.5										
	29.0	47.2									
29.0	23.1	49.1									
47.2	49.1	43.8									

Etap 2: wybranie węzła, obliczenie wartości funkcji *f*(*x*).

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:

- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:

33.0	26.5										
	29.0	47.2									
29.0	23.1	49.1	∞								
47.2	49.1	43.8	∞								
	21.7	23.0	∞								

Etap 3: wybranie węzła, obliczenie wartości funkcji *f*(*x*).

Algorytm A* – służy do znajdowania najkrótszej ścieżki w grafie między dwoma węzłami.

Funkcje heurystyczne:
- typ Manhattan – odległość dwóch węzłów to suma ich odległości w pionie i w poziomie,
- odległość euklidesowa – odległość dwóch węzłów w przestrzeni.

Przykład:

33.0	26.5					237	238	234			
	29.0	47.2		215	217	212	209	234	253		
29.0	23.1	49.1	∞		187	187	∞	230	252	269	
47.2	49.1	43.8	∞	166	166	162	∞	252	249	268	
93.5	65.5	66.8	∞	146	141	181	∞	269	268	266	
88.2	89.5	84.8	∞	123	142	159					
	108	109	105	127	141	162					
		129	131	126	166	162					

Etap 4-13

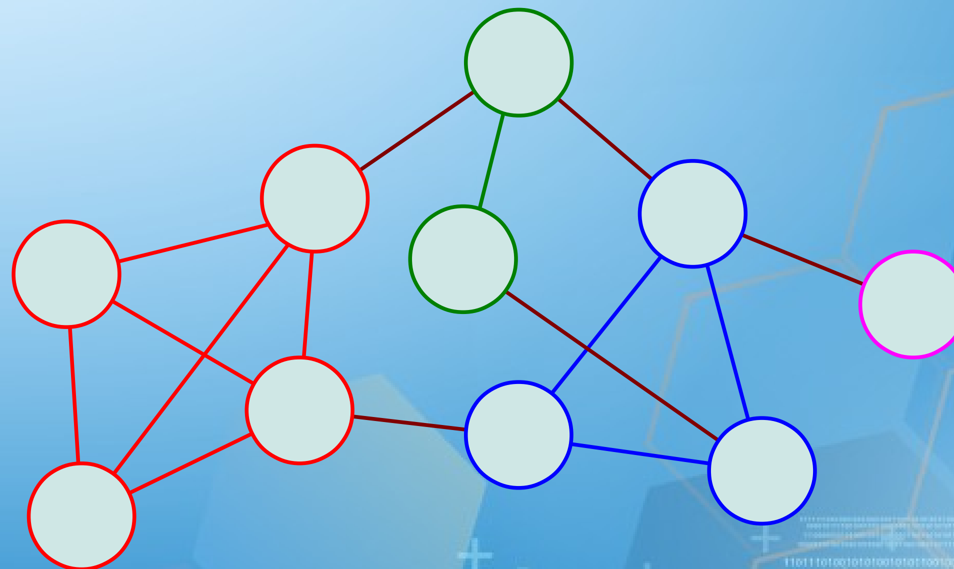
Podział grafu na kliki – polega na znalezieniu kliki *k*-tego stopnia w grafie.

Uwagi:

- jest to problem NP-zupełny,
- kliki wykorzystuje się w problemach optymalizacyjnych,
- graf w którym liczba chromatyczna jest równa rozmiarowi największej kliki jest grafem doskonałym,
- do rozwiązania problemu kliki stosuje się algorytmy heurystyczne.

Algorytmy:

- Tseng-Siewiorek,
- Bhasker,
- Kim-Shin.



Algorytm Tseng-Siewiorek – heurystyczny algorytm wykonujący podział grafu na kliki.

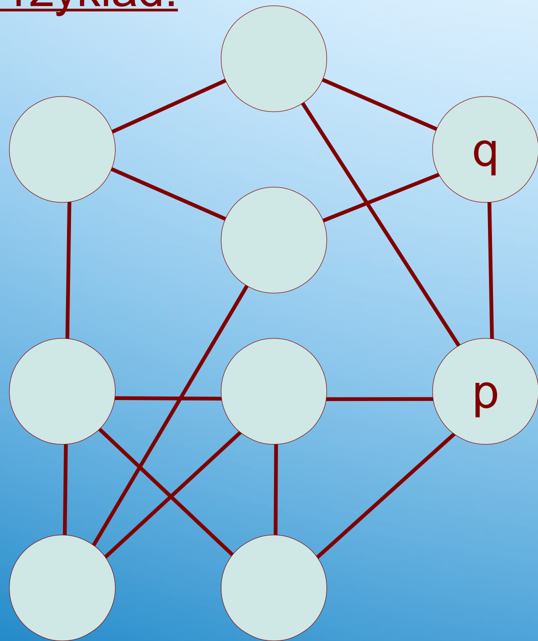
Algorytm:

1. Wybranie krawędzi (p,q) posiadającej największą liczbę sąsiadów – wybranie takiego p i takiego q, aby stopień węzłów był maksymalny.
2. Połączenie p z q i przypisanie ich do p. Usunięcie z p i q krawędzi nie łączących ich z sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC.
3. Wybranie węzła q, posiadającego najwięcej wspólnych sąsiadów z p. Jeśli taki istnieje to wykonanie etapu 1, jeśli nie to wykonanie etapu 2.

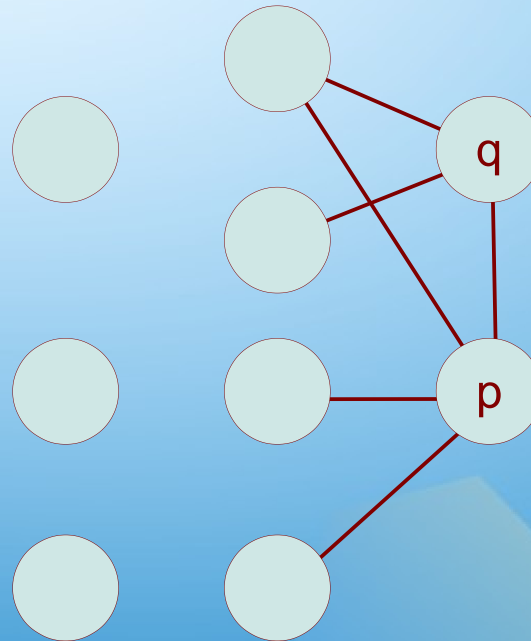
Algorytm Tseng-Siewiorek – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:
1. Wybranie krawędzi (*p*,*q*) posiadającej największą liczbę sąsiadów – wybranie takiego *p* i takiego *q*, aby stopień węzłów był maksymalny.
2. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie z *p* i *q* krawędzi nie łączących ich z sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC.
3. Wybranie węzła *q*, posiadającego najwięcej wspólnych sąsiadów z *p*. Jeśli taki istnieje to wykonanie etapu 1, jeśli nie to wykonanie etapu 2.

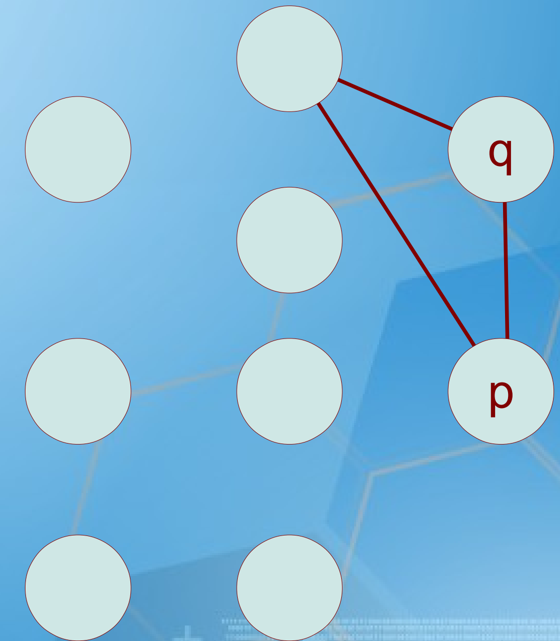
Przykład:



Iteracja 1, etap 1



Iteracja 1, etap 2



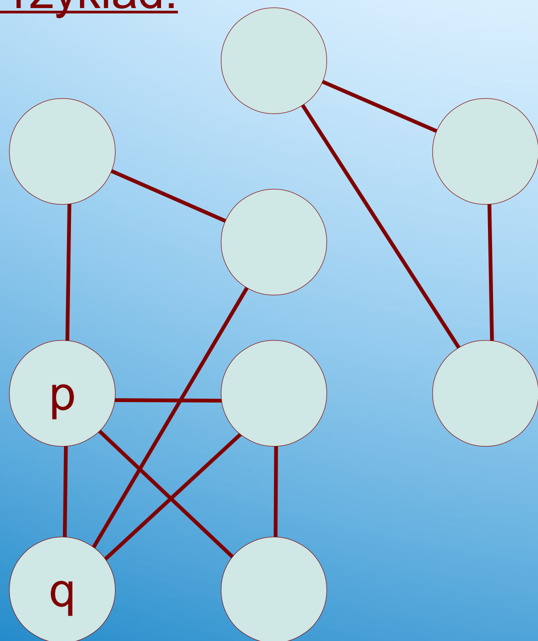
Iteracja 1, etap 3

Algorytm Tseng-Siewiorek – heurystyczny algorytm wykonujący podział grafu na kliki.

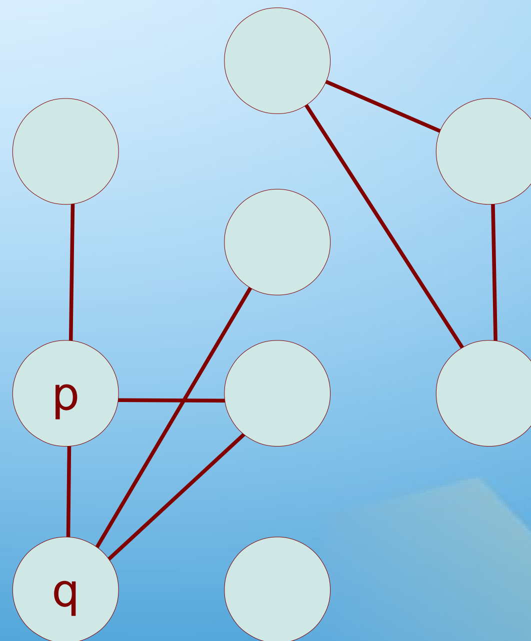
Algorytm:

1. Wybranie krawędzi (*p*,*q*) posiadającej największą liczbę sąsiadów – wybranie takiego *p* i takiego *q*, aby stopień węzłów był maksymalny.
2. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie z *p* i *q* krawędzi nie łączących ich z sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC.
3. Wybranie węzła *q*, posiadającego najwięcej wspólnych sąsiadów z *p*. Jeśli taki istnieje to wykonanie etapu 1, jeśli nie to wykonanie etapu 2.

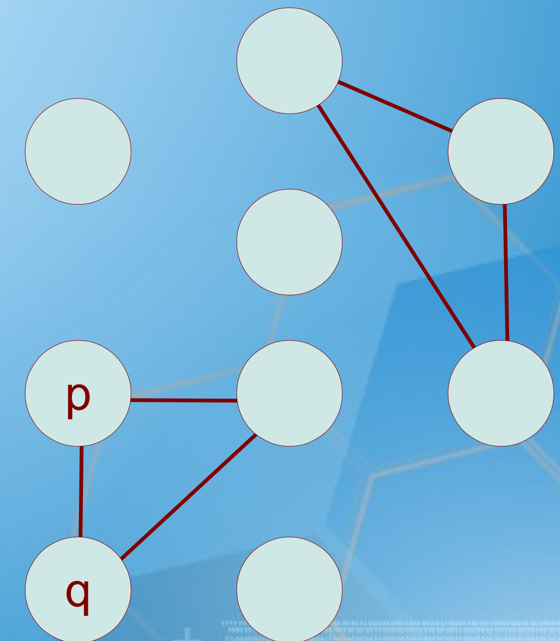
Przykład:



Iteracja 2, etap 1



Iteracja 2, etap 2

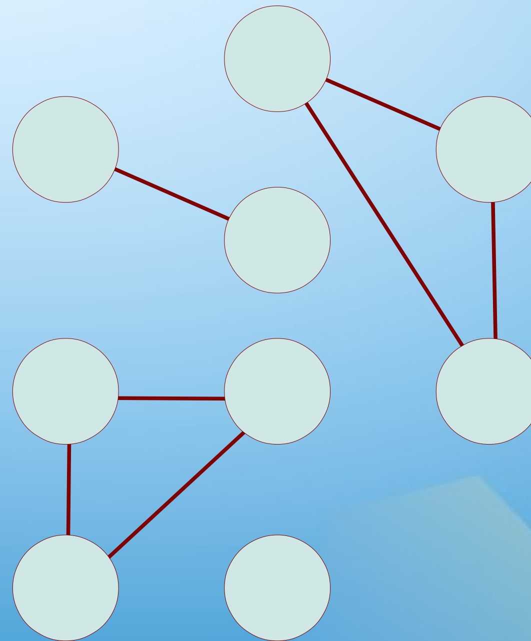


Iteracja 2, etap 3

Algorytm Tseng-Siewiorek – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:
1. Wybranie krawędzi (*p*,*q*) posiadającej największą liczbę sąsiadów – wybranie takiego *p* i takiego *q*, aby stopień węzłów był maksymalny.
2. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie z *p* i *q* krawędzi nie łączących ich z sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC.
3. Wybranie węzła *q*, posiadającego najwięcej wspólnych sąsiadów z *p*. Jeśli taki istnieje to wykonanie etapu 1, jeśli nie to wykonanie etapu 2.

Przykład:



Koniec algorytmu

Algorytm Bhasker'a – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:

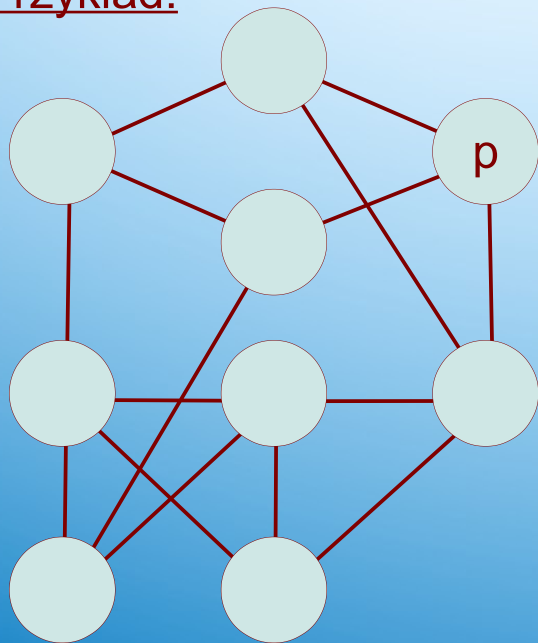
1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najmniej z nim sąsiadów i nazwanie go *q*. (węzeł *q* musi posiadać co najmniej jednego sąsiada z *p*).
3. Połączenie *p* i *q*, przypisanie ich do *p*. Usunięcie takich krawędzi z *p* i *q*, które nie są połączone z ich sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonanie etapu 1.

Algorytm Bhasker'a – heurystyczny algorytm wykonujący podział grafu na kliki.

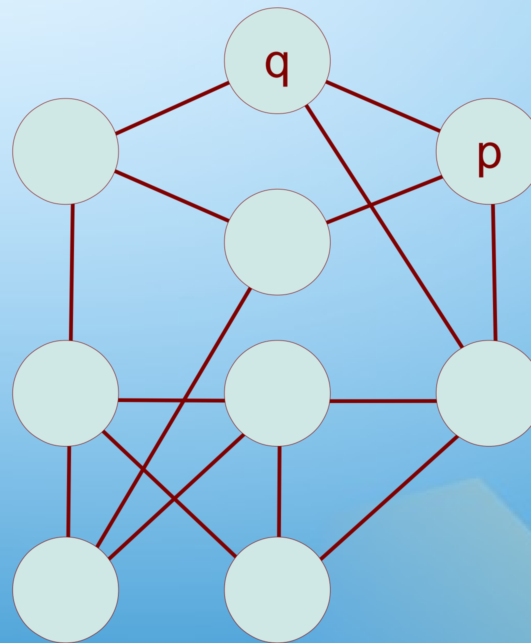
Algorytm:

1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najmniej z nim sąsiadów i nazwanie go *q*. (węzeł *q* musi posiadać co najmniej jednego sąsiada z *p*).
3. Połączenie *p* i *q*, przypisanie ich do *p*. Usunięcie takich krawędzi z *p* i *q*, które nie są połączone z ich sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonanie etapu 1.

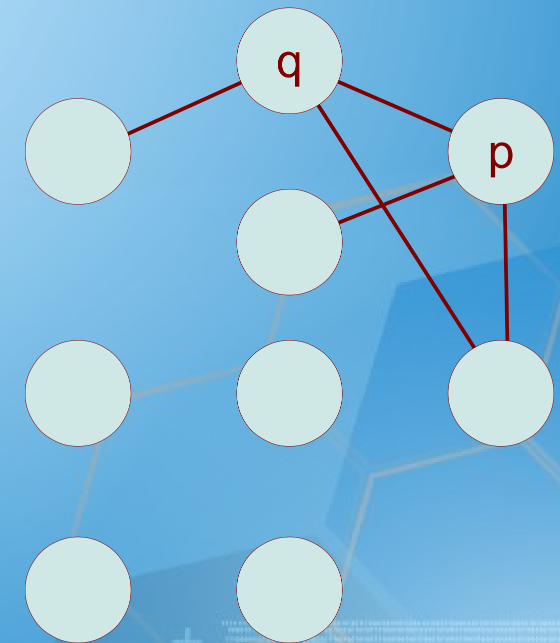
Przykład:



Iteracja 1, etap 1



Iteracja 1, etap 2



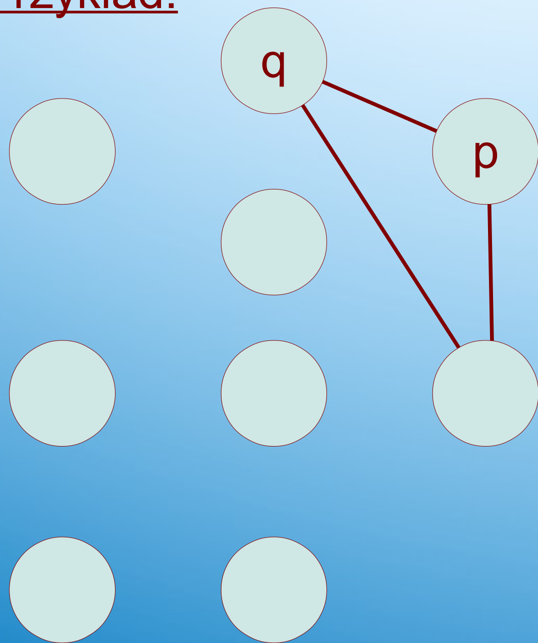
Iteracja 1, etap 3

Algorytm Bhasker'a – heurystyczny algorytm wykonujący podział grafu na kliki.

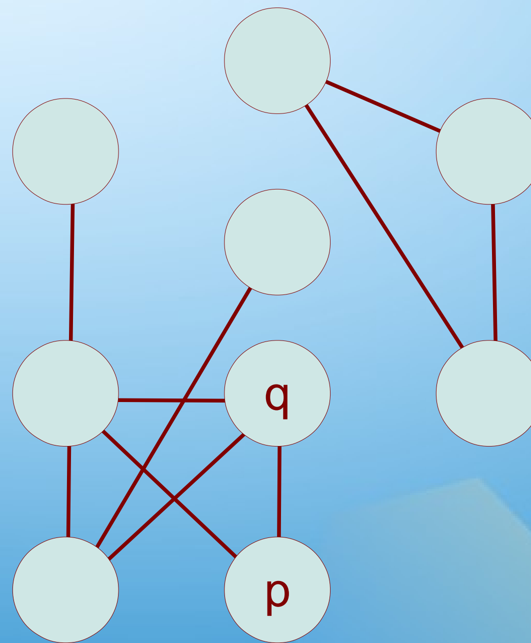
Algorytm:

1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najmniej z nim sąsiadów i nazwanie go *q*. (węzeł *q* musi posiadać co najmniej jednego sąsiada z *p*).
3. Połączenie *p* i *q*, przypisanie ich do *p*. Usunięcie takich krawędzi z *p* i *q*, które nie są połączone z ich sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonanie etapu 1.

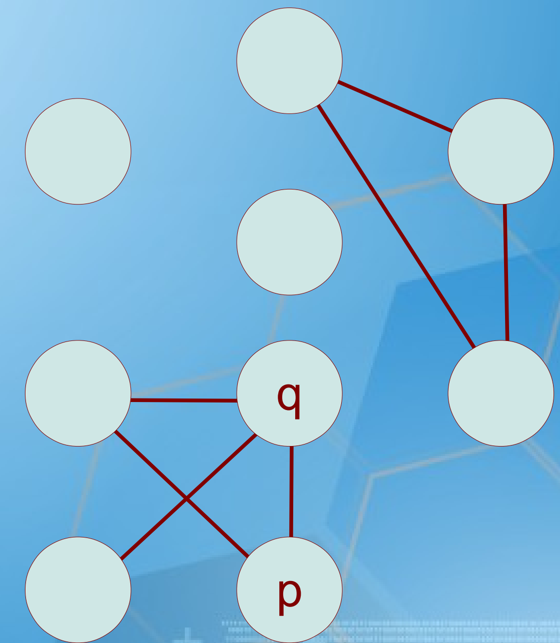
Przykład:



Iteracja 1, etap 3



Iteracja 2, etap 1



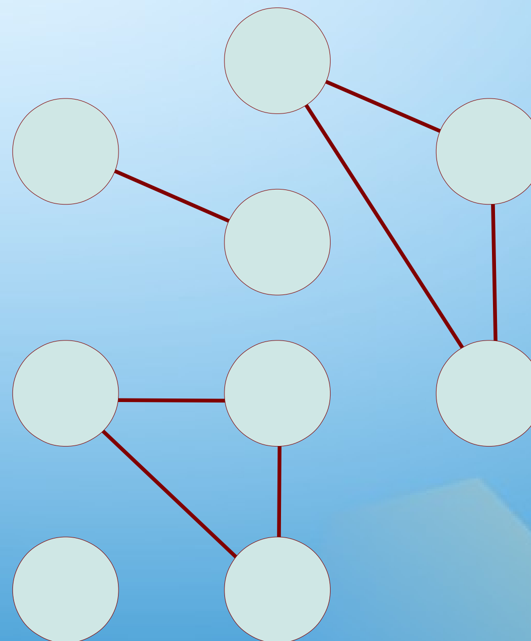
Iteracja 2, etap 2

Algorytm Bhasker'a – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:

1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najmniej z nim sąsiadów i nazwanie go *q*. (węzeł *q* musi posiadać co najmniej jednego sąsiada z *p*).
3. Połączenie *p* i *q*, przypisanie ich do *p*. Usunięcie takich krawędzi z *p* i *q*, które nie są połączone z ich sąsiadami. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonanie etapu 1.

Przykład:



Iteracja 2, etap 3

Algorytm Kim-Shin – heurystyczny algorytm wykonujący podział grafu na kliki.

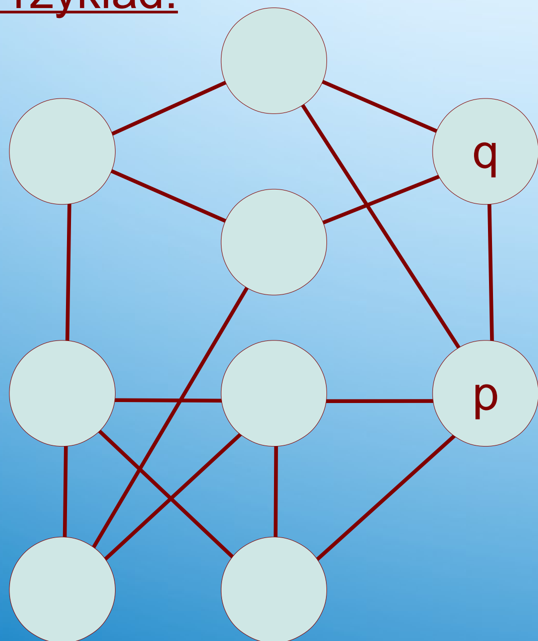
Algorytm:

1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najwięcej z nim sąsiadów, ale o najmniejszym stopniu i nazwanie go *q*. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie krawędzi z *p* i *q*, które nie łączą ich z sąsiadami. Jeśli *p* posiada jakichś sąsiadów to wykonanie etapu 2.
3. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonane etapu 1.

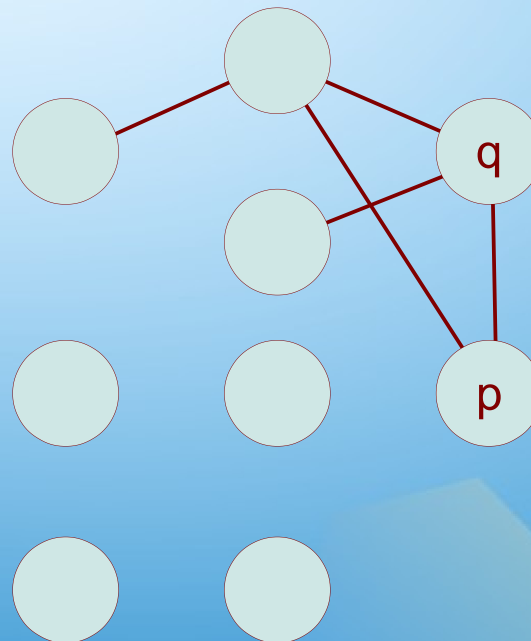
Algorytm Kim-Shin – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:
1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najwięcej z nim sąsiadów, ale o najmniejszym stopniu i nazwanie go *q*. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie krawędzi z *p* i *q*, które nie łączą ich z sąsiadami. Jeśli *p* posiada jakichś sąsiadów to wykonanie etapu 2.
3. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonane etapu 1.

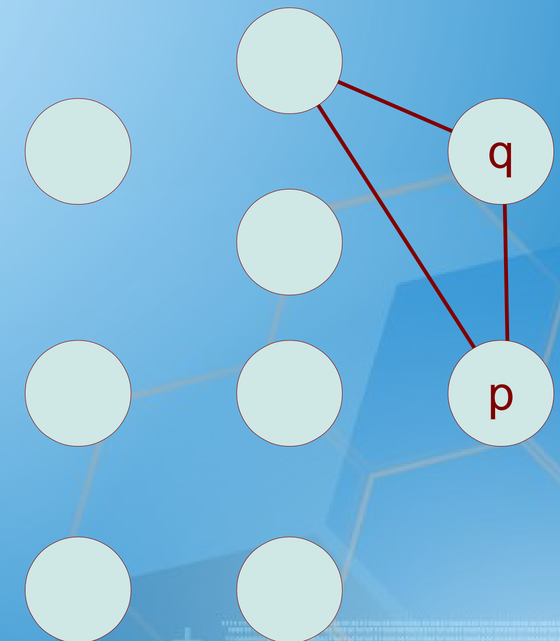
Przykład:



Iteracja 1, etap 3



Iteracja 1, etap 2.1



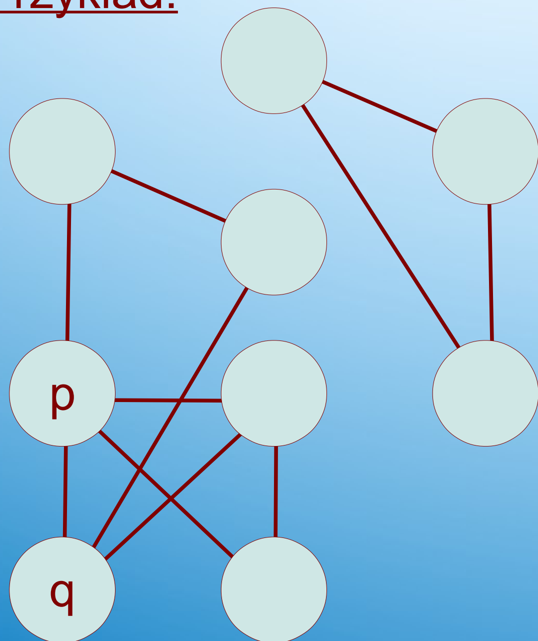
Iteracja 1, etap 2.2

Algorytm Kim-Shin – heurystyczny algorytm wykonujący podział grafu na kliki.

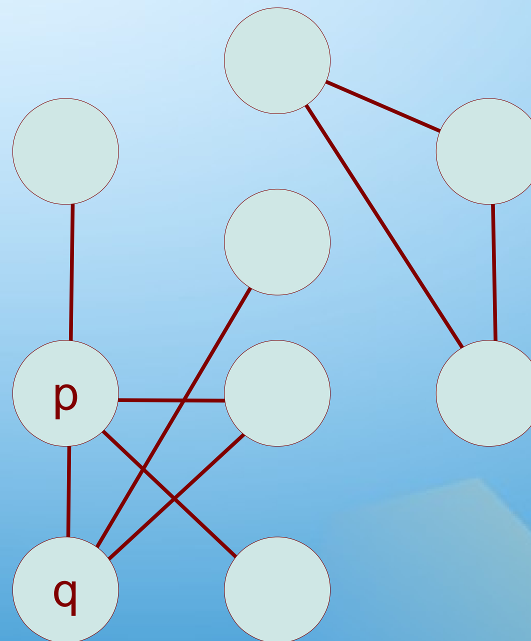
Algorytm:

1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najwięcej z nim sąsiadów, ale o najmniejszym stopniu i nazwanie go *q*. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie krawędzi z *p* i *q*, które nie łączą ich z sąsiadami. Jeśli *p* posiada jakichś sąsiadów to wykonanie etapu 2.
3. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonane etapu 1.

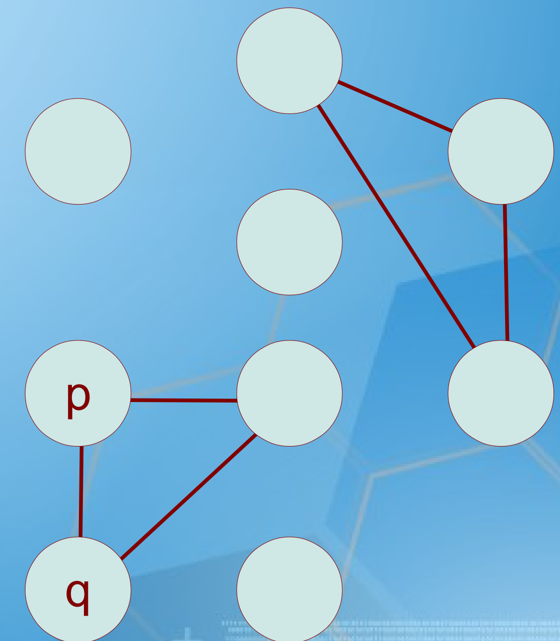
Przykład:



Iteracja 2, etap 1



Iteracja 2, etap 2.1

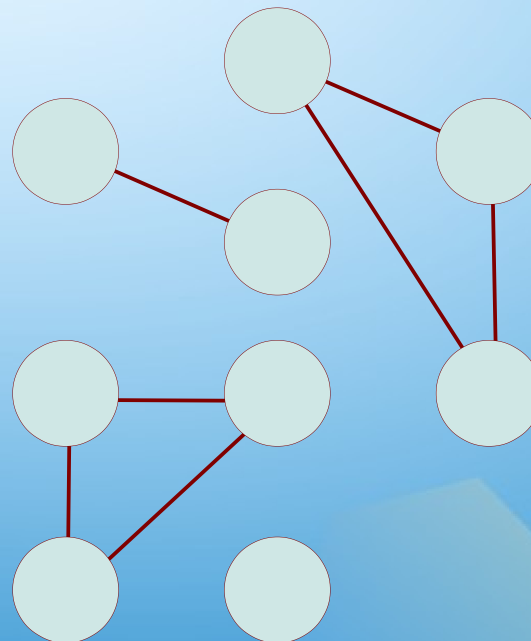


Iteracja 2, etap 2.2

Algorytm Kim-Shin – heurystyczny algorytm wykonujący podział grafu na kliki.

Algorytm:
1. Wybranie węzła z najmniejszą liczbą sąsiadów i nazwanie go *p*.
2. Wybranie sąsiada węzła *p* posiadającego najwięcej z nim sąsiadów, ale o najmniejszym stopniu i nazwanie go *q*. Połączenie *p* z *q* i przypisanie ich do *p*. Usunięcie krawędzi z *p* i *q*, które nie łączą ich z sąsiadami. Jeśli *p* posiada jakichś sąsiadów to wykonanie etapu 2.
3. Jeśli lista krawędzi jest pusta to KONIEC, jeśli nie jest pusta to wykonane etapu 1.

Przykład:



Koniec algorytmu

Algorytmy sortujące

Sortowanie – proces polegający na uporządkowaniu danych względem pewnych cech charakterystycznych (kluczy).

Algorytmy sortujące stosowane są w celu optymalizacji zbioru danych, pozwalającej wydajniej stosować inne algorytmy, np. wyszukiwanie danych.

Sortowanie – proces polegający na uporządkowaniu danych względem pewnych cech charakterystycznych (kluczy).

Algorytmy sortujące stosowane są w celu optymalizacji zbioru danych, pozwalającej wydajniej stosować inne algorytmy, np. wyszukiwanie danych.

Typy algorytmów:

- stabilne,
- niestabilne.

Sortowanie – proces polegający na uporządkowaniu danych względem pewnych cech charakterystycznych (kluczy).

Algorytmy sortujące stosowane są w celu optymalizacji zbioru danych, pozwalającej wydajniej stosować inne algorytmy, np. wyszukiwanie danych.

Typy algorytmów:

- stabilne, ←
- niestabilne.

Utrzymują kolejność elementów o tych samych cechach charakterystycznych – dla dwóch elementów *R* i *S*, jeśli przed sortowaniem *R* występowało po *S* to i po sortowaniu R będzie występowało po *S*:

{(4,1)(3,7)(3,1)(5,6)} —sortowanie stabilne→ {(3,7)(3,1)(4,1)(5,6)}

Sortowanie – proces polegający na uporządkowaniu danych względem pewnych cech charakterystycznych (kluczy).

Algorytmy sortujące stosowane są w celu optymalizacji zbioru danych, pozwalającej wydajniej stosować inne algorytmy, np. wyszukiwanie danych.

Typy algorytmów:

- stabilne,
- niestabilne.

Stabilne: Utrzymują kolejność elementów o tych samych cechach charakterystycznych – dla dwóch elementów *R* i *S*, jeśli przed sortowaniem *R* występowało po *S* to i po sortowaniu R będzie występowało po *S*:

{(4,1)(3,7)(3,1)(5,6)} —sortowanie stabilne→ {(3,7)(3,1)(4,1)(5,6)}

Niestabilne: Zmieniają kolejność elementów o tych samych cechach charakterystycznych:

{(4,1)(3,7)(3,1)(5,6)} —sortowanie niestabilne→ {(3,1)(3,7)(4,1)(5,6)}

Sortowanie – proces polegający na uporządkowaniu danych względem pewnych cech charakterystycznych (kluczy).

Algorytmy sortujące stosowane są w celu optymalizacji zbioru danych, pozwalającej wydajniej stosować inne algorytmy, np. wyszukiwanie danych.

Typy algorytmów:
- stabilne,
- niestabilne.

Stabilne: Utrzymują kolejność elementów o tych samych cechach charakterystycznych – dla dwóch elementów *R* i *S*, jeśli przed sortowaniem *R* występowało po *S* to i po sortowaniu R będzie występowało po *S*:

{(4,1)(3,7)(3,1)(5,6)} —sortowanie stabilne→ {(3,7)(3,1)(4,1)(5,6)}

Niestabilne: Zmieniają kolejność elementów o tych samych cechach charakterystycznych:

{(4,1)(3,7)(3,1)(5,6)} —sortowanie niestabilne→ {(3,1)(3,7)(4,1)(5,6)}

Stabilność nie jest istotna jeśli klucze nie pozwalają na rozróżnianie elementów.

Sortowanie bąbelkowe – ang. bubblesort – stabilny algorytm sortujący o złożoności czasowej O(n^2) i pamięciowej O(1).

Algorytm:

1. Powtarzanie od pierwszego elementu ciągu do ostatniego:
   - jeśli element *i* jest większy od elementu *i+1* to zamiana elementów miejscami w ciągu.
2. Jeśli zamiana elementów miejscami miała miejsce, to przejście do etapu 1, jeśli nie to KONIEC.

Sortowanie bąbelkowe – ang. bubblesort – stabilny algorytm sortujący o złożoności czasowej O(n^2) i pamięciowej O(1).

Algorytm:
1. Powtarzanie od pierwszego elementu ciągu do ostatniego:
   - jeśli element *i* jest większy od elementu *i+1* to zamiana elementów miejscami w ciągu.
2. Jeśli zamiana elementów miejscami miała miejsce, to przejście do etapu 1, jeśli nie to KONIEC.

Pseudokod algorytmu:

```
function sortuj(tablica A[0..r])
begin
   repeat
      i = 0;
      repeat
         if (A[i] > A[i + 1]) then zamień elementy A[i] i A[i + 1];
         i = i + 1;
      until i = r or była zamiana elementów;
   until nie było zamiany elementów;
end;
```

Sortowanie bąbelkowe – ang. bubblesort – stabilny algorytm sortujący o złożoności czasowej $O(n^2)$ i pamięciowej $O(1)$.

Algorytm:

1. Powtarzanie od pierwszego elementu ciągu do ostatniego:
   - jeśli element *i* jest większy od elementu *i+1* to zamiana elementów miejscami w ciągu.
2. Jeśli zamiana elementów miejscami miała miejsce, to przejście do etapu 1, jeśli nie to KONIEC.

Pseudokod algorytmu:

```
function sortuj(tablica A[0..r])
begin
  repeat
    i = 0;
    repeat
      if (A[i] > A[i + 1]) then zamień elementy A[i] i A[i + 1];
      i = i + 1;
    until i = r or była zamiana elementów;
  until nie było zamiany elementów;
end;
```

Uwagi:

- efektywność algorytmu można zwiększyć kontynuując szukanie kolejnych elementów do zamiany po znalezieniu pierwszej pary.

Sortowanie przez wstawianie – stabilny algorytm sortujący o złożoności czasowej O(n^2).

Algorytm:
Polega na wstawianiu elementów we właściwe miejsca.

Sortowanie przez wstawianie – stabilny algorytm sortujący o złożoności czasowej O(n^2).

Algorytm:
Polega na wstawianiu elementów we właściwe miejsca.

Pseudokod algorytmu:

```
function sort(tablica A, n)
begin
    for i = 2 to n do
    begin
        klucz = A[i];
        j = i – 1;
        while j > 0 and A[j] > klucz do
        begin
            A[j + 1] = A[j];
            j = j – 1;
            A[j + 1] = klucz;
        end
    end
end;
```

Sortowanie przez wstawianie – stabilny algorytm sortujący o złożoności czasowej O(n^2).

Algorytm:
Polega na wstawianiu elementów we właściwe miejsca.

Pseudokod algorytmu:

```
function sort(tablica A, n)
begin
    for i = 2 to n do
    begin
        klucz = A[i];
        j = i – 1;
        while j > 0 and A[j] > klucz do
        begin
            A[j + 1] = A[j];
            j = j – 1;
            A[j + 1] = klucz;
        end
    end
end;
```

Uwagi:
- algorytm jest efektywny dla małych porcji danych lub dla danych częściowo posortowanych.

Sortowanie przez scalanie – stabilny algorytm sortujący o złożoności czasowej równej O(n log n) i pamięciowej O(n).

Sortowanie przez scalanie – stabilny algorytm sortujący o złożoności czasowej równej O(n log n) i pamięciowej O(n).

<u>Algorytm:</u>

1. Podzielenie zestawu danych na dwie równe części.
2. Zastosowanie procedury sortującej dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element.
3. Połączenie posortowanych podciągów A[1..n] i B[1..m] do ciągu C[1..m+n]:
    1. Utworzenie wskaźników na początki ciągów A i B → i = 1, j = 1.
    2. Jeżeli $i > n$ to dołączenie pozostałych elementów ciągu B do ciągu C i KONIEC.
    3. Jeżeli $j > m$ to dołączenie pozostałych elementów ciągu A do ciągu C i KONIEC.
    4. Jeżeli A[i] ⩽ B[j] to dołączenie A[i] do C i zwiększenie i o jeden, w przeciwnym przypadku dołączenie B[j] do C i zwiększenie j o jeden.
    5. Powtarzanie od etapu 3.2 aż wszystkie elementy A i B trafią do C.

Sortowanie przez scalanie – stabilny algorytm sortujący o złożoności czasowej równej O(n log n) i pamięciowej O(n).

Algorytm:

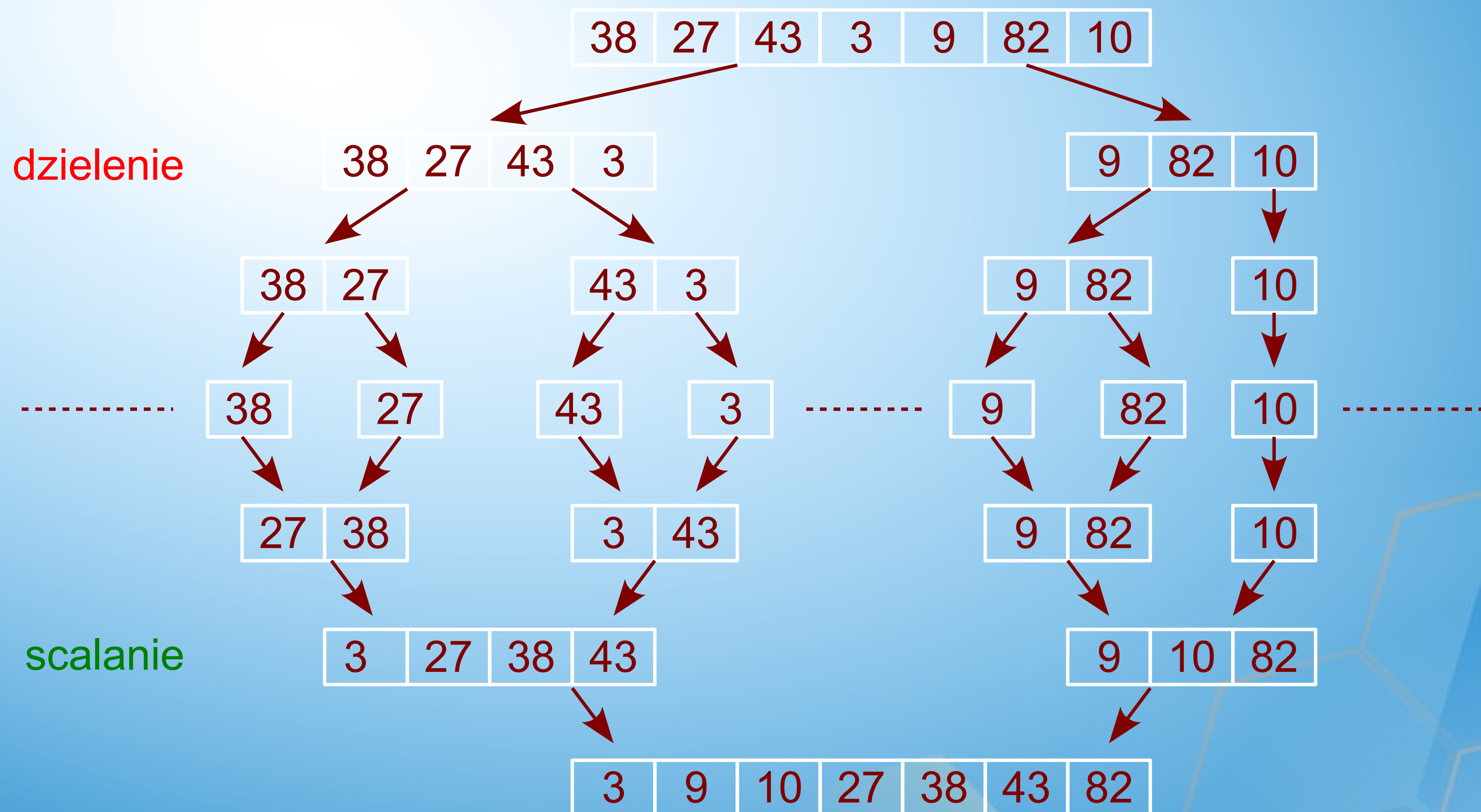
1. Podzielenie zestawu danych na dwie równe części.
2. Zastosowanie procedury sortującej dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element.
3. Połączenie posortowanych podciągów A[1..n] i B[1..m] do ciągu C[1..m+n]:
   1. Utworzenie wskaźników na początki ciągów A i B → i = 1, j = 1.
   2. Jeżeli $i > n$ to dołączenie pozostałych elementów ciągu B do ciągu C i KONIEC.
   3. Jeżeli $j > m$ to dołączenie pozostałych elementów ciągu A do ciągu C i KONIEC.
   4. Jeżeli A[i] ⩽ B[j] to dołączenie A[i] do C i zwiększenie i o jeden, w przeciwnym przypadku dołączenie B[j] do C i zwiększenie j o jeden.
   5. Powtarzanie od etapu 3.2 aż wszystkie elementy A i B trafią do C.

Uwagi:

- scalenie wymaga O(n+m) porównań elementów i wstawienia ich do tablicy wynikowej.

Sortowanie przez scalanie – stabilny algorytm sortujący o złożoności czasowej równej O(*n* log *n*) i pamięciowej O(n).

Przykład:



Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(n + k) i pamięciowej O(n + k).

Algorytm:

1. Przeglądanie elementów ciągu A od 0 do n.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(n + k) i pamięciowej O(n + k).

Algorytm:
1. Przeglądanie elementów ciągu A od 0 do n.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Uwagi:
- liczba k oznacza rozpiętość sortowanych danych,
- najczęściej tablica pomocnicza ma taką liczbę elementów jak zakres sortowanych danych,
- tablicę pomocniczą można zaimplementować w postaci listy.

Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(n + k) i pamięciowej O(n + k).

Algorytm:

1. Przeglądanie elementów ciągu A od 0 do n.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Uwagi:

- liczba k oznacza rozpiętość sortowanych danych,
- najczęściej tablica pomocnicza ma taką liczbę elementów jak zakres sortowanych danych,
- tablicę pomocniczą można zaimplementować w postaci listy.

Przykład:

Dane wejściowe : 1 3 4 1 5 9 0 0 8 2 6 3 8 3 7 5 9 2 7 9 9 2 1 8 8 0 8 5 8
Tablica pomocnicza : {0:0; 1:0; 2:0; 3:0; 4:0; 5:0; 6:0; 7:0; 8:0; 9:0} – początek algorytmu

Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(*n* + *k*) i pamięciowej O(*n* + *k*).

Algorytm:
1. Przeglądanie elementów ciągu A od 0 do *n*.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Uwagi:
- liczba *k* oznacza rozpiętość sortowanych danych,
- najczęściej tablica pomocnicza ma taką liczbę elementów jak zakres sortowanych danych,
- tablicę pomocniczą można zaimplementować w postaci listy.

Przykład:

Dane wejściowe : 1 3 4 1 5 9 0 0 8 2 6 3 8 3 7 5 9 2 7 9 9 2 1 8 8 0 8 5 8
Tablica pomocnicza : {0:0; 1:0; 2:0; 3:0; 4:0; 5:0; 6:0; 7:0; 8:0; 9:0} – początek algorytmu
{0:3; 1:3; 2:3; 3:3; 4:1; 5:3; 6:1; 7:2; 8:6; 9:4} – koniec zliczania

Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(n + k) i pamięciowej O(n + k).

Algorytm:
1. Przeglądanie elementów ciągu A od 0 do *n*.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Uwagi:
- liczba *k* oznacza rozpiętość sortowanych danych,
- najczęściej tablica pomocnicza ma taką liczbę elementów jak zakres sortowanych danych,
- tablicę pomocniczą można zaimplementować w postaci listy.

Przykład:

Dane wejściowe : 1 3 4 1 5 9 0 0 8 2 6 3 8 3 7 5 9 2 7 9 9 2 1 8 8 0 8 5 8
Tablica pomocnicza : {0:0; 1:0; 2:0; 3:0; 4:0; 5:0; 6:0; 7:0; 8:0; 9:0} – początek algorytmu
{0:3; 1:3; 2:3; 3:3; 4:1; 5:3; 6:1; 7:2; 8:6; 9:4} – koniec zliczania
Dane wyjściowe : 0 0 0 – wartość 0 jest 3 razy w sortowanej tablicy,

Sortowanie przez zliczanie – stabilny algorytm sortujący o złożoności czasowej równej O(n + k) i pamięciowej O(n + k).

Algorytm:

1. Przeglądanie elementów ciągu A od 0 do n.
2. Jeśli wybrany element znajduje się w tablicy pomocniczej to zwiększenie liczby jego wystąpień, jeśli nie znajduje się to wstawienie tego elementu do tablicy pomocniczej zgodnie z porządkiem sortowania.
3. Odtworzenie tablicy wynikowej poprzez wstawienie do niej elementów z tablicy pomocniczej we właściwej liczbie ich wystąpień.

Uwagi:

- liczba k oznacza rozpiętość sortowanych danych,
- najczęściej tablica pomocnicza ma taką liczbę elementów jak zakres sortowanych danych,
- tablicę pomocniczą można zaimplementować w postaci listy.

Przykład:

Dane wejściowe : 1 3 4 1 5 9 0 0 8 2 6 3 8 3 7 5 9 2 7 9 9 2 1 8 8 0 8 5 8
Tablica pomocnicza : {0:0; 1:0; 2:0; 3:0; 4:0; 5:0; 6:0; 7:0; 8:0; 9:0} – początek algorytmu
{0:3; 1:3; 2:3; 3:3; 4:1; 5:3; 6:1; 7:2; 8:6; 9:4} – koniec zliczania
Dane wyjściowe : 0 0 0 – wartość 0 jest 3 razy w sortowanej tablicy,
1 1 1 – wartość 1 jest 3 razy w sortowanej tablicy,
...
2 2 2 3 3 3 4 5 5 5 6 7 7 8 8 8 8 8 8 9 9 9 9

Sortowanie kubełkowe – stabilny algorytm sortujący o złożoności czasowej średniej równej O(n) i pamięciowej O(k).

Algorytm:

1. Podzielenie zadanego przedziału liczb na n podprzedziałów (kubełków) o równej długości.
2. Przypisanie liczb z sortowanej tablicy do odpowiednich kubełków.
3. Sortowanie liczb w niepustych kubełkach.
4. Wypisanie po kolei zawartość niepustych kubełków.

Uwagi:

- złożoność pesymistyczna algorytmu wynosi O(n^2).

Sortowanie kubełkowe – stabilny algorytm sortujący o złożoności czasowej średniej równej O(n) i pamięciowej O(k).

Algorytm:
1. Podzielenie zadanego przedziału liczb na n podprzedziałów (kubełków) o równej długości.
2. Przypisanie liczb z sortowanej tablicy do odpowiednich kubełków.
3. Sortowanie liczb w niepustych kubełkach.
4. Wypisanie po kolei zawartość niepustych kubełków.

Uwagi:
- złożoność pesymistyczna algorytmu wynosi O(n^2).

Sortowanie biblioteczne – stabilny algorytm sortujący o złożoność czasowej O(n log n).

Algorytm:
Algorytm bazuje na algorytmie sortowania przez wstawianie, ale z dodawaniem pustych miejsc w tablicy w celu przyśpieszenie wstawiania elementów.

Uwagi:
- złożoność pesymistyczna algorytmu wynosi O(n^2).

Sortowanie szybkie – algorytm sortujący o złożoności czasowej średniej O(*n* log *n*).

Algorytm:

1. Ze zbioru wejściowego wybranie elementu osiowego.
2. Przeniesienie na początek tablicy wszystkich elementów mniejszych od elementu osiowego.
3. Przeniesienie na koniec tablicy wszystkich elementów większych od elementu osiowego.
4. Wykonanie etapów 1-3 dla podzbiorów tablicy wyznaczonych przez element osiowy.
5. Jeśli podzbiór zawiera 1 element to KONIEC.

Sortowanie szybkie – algorytm sortujący o złożoności czasowej średniej O(*n* log *n*).

Algorytm:

1. Ze zbioru wejściowego wybranie elementu osiowego.
2. Przeniesienie na początek tablicy wszystkich elementów mniejszych od elementu osiowego.
3. Przeniesienie na koniec tablicy wszystkich elementów większych od elementu osiowego.
4. Wykonanie etapów 1-3 dla podzbiorów tablicy wyznaczonych przez element osiowy.
5. Jeśli podzbiór zawiera 1 element to KONIEC.

Pseudokod algorytmu:

```
function Quicksort(A, l, r)
begin
    if l < r then begin
        i = podzial_tablicy(A, l, r);
        Quicksort(A, l, i – 1);
        Quicksort(A, i, r);
    end
end;
```

Sortowanie szybkie – algorytm sortujący o złożoności czasowej średniej O(*n* log *n*).

Algorytm:

1. Ze zbioru wejściowego wybranie elementu osiowego.
2. Przeniesienie na początek tablicy wszystkich elementów mniejszych od elementu osiowego.
3. Przeniesienie na koniec tablicy wszystkich elementów większych od elementu osiowego.
4. Wykonanie etapów 1-3 dla podzbiorów tablicy wyznaczonych przez element osiowy.
5. Jeśli podzbiór zawiera 1 element to KONIEC.

Pseudokod algorytmu:

```
function Quicksort(A, l, r)
begin
    if l < r then begin
        i = podzial_tablicy(A, l, r);
        Quicksort(A, l, i – 1);
        Quicksort(A, i, r);
    end
end;
```

Uwagi:

- algorytm został zaprojektowany metoda „dziel i zwyciężaj”,
- algorytm został zaimplementowany w wielu językach programowania,
- złożoność pesymistyczna algorytmu wynosi O(n^2).

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

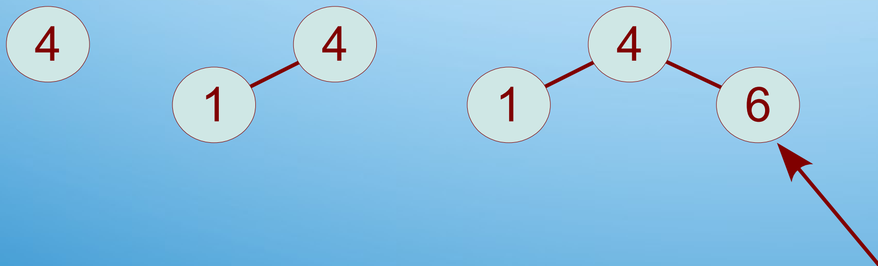
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

4	1	6	5	8	12	3	11	9	2



Warunek kopca przestaje być spełniony.

Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

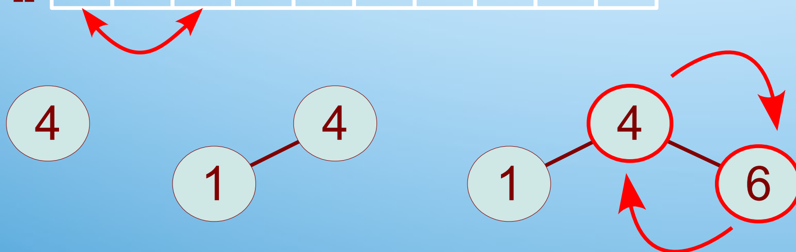
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

4	1	6	5	8	12	3	11	9	2



Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

6	1	4	5	8	12	3	11	9	2

Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

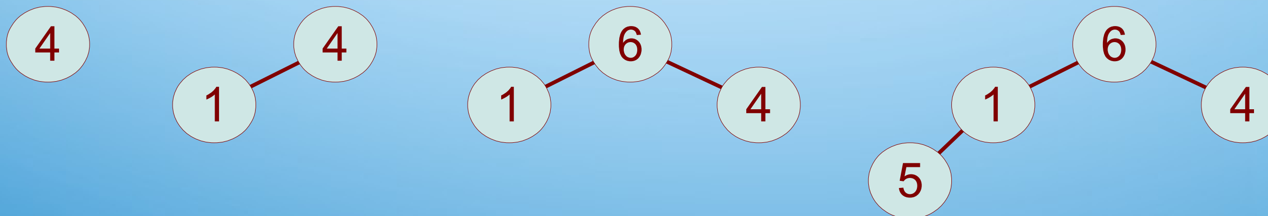
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

6	1	4	5	8	12	3	11	9	2



Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

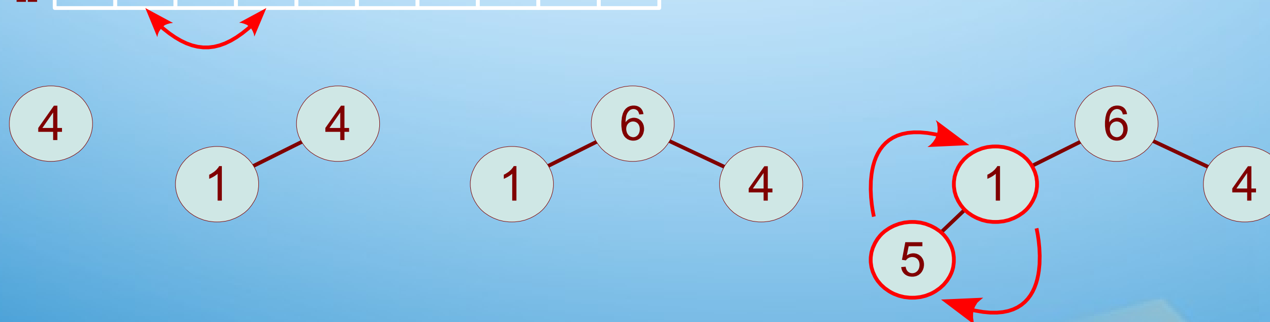
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

6	1	4	5	8	12	3	11	9	2



Etap 1 – budowa kopca.

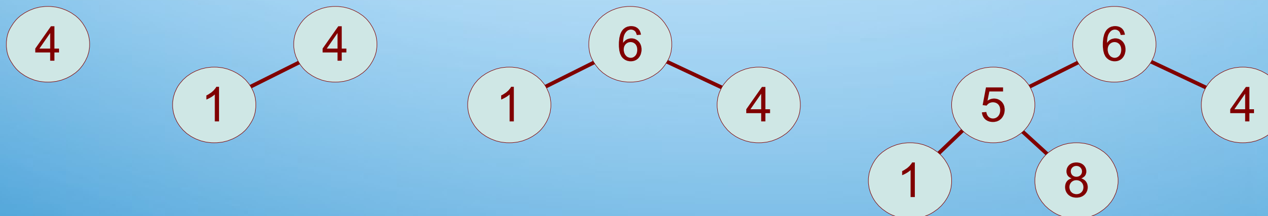
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:	6	5	4	1	8	12	3	11	9	2



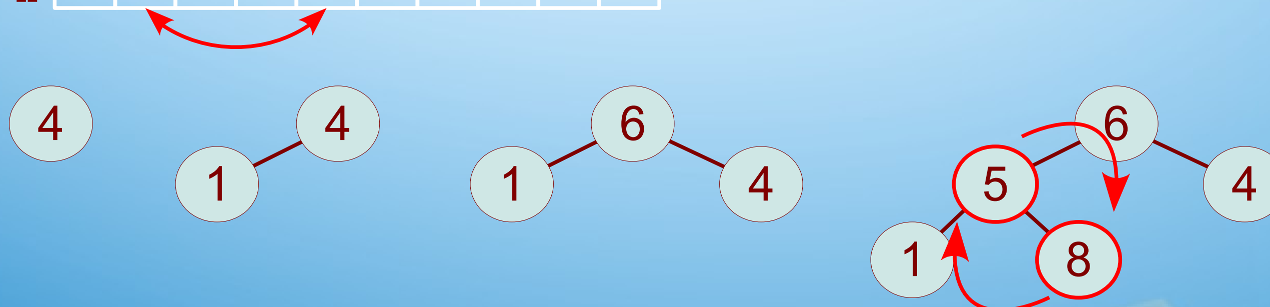
Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:



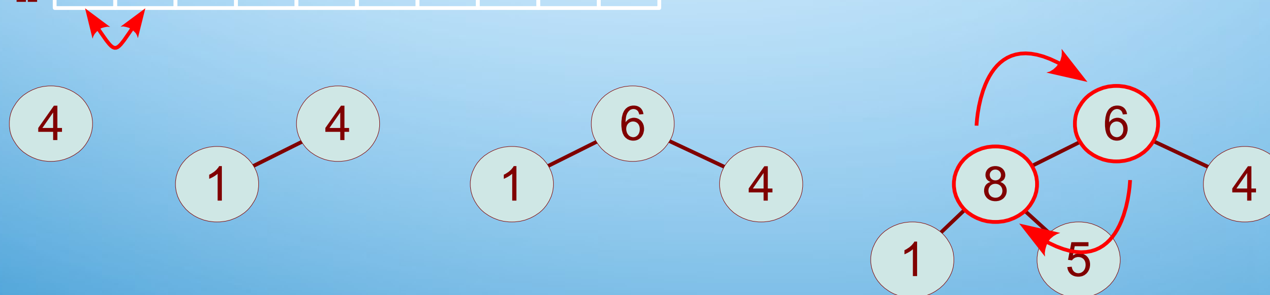
Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(*n* log *n*).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log *n*).

Przykład:



Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

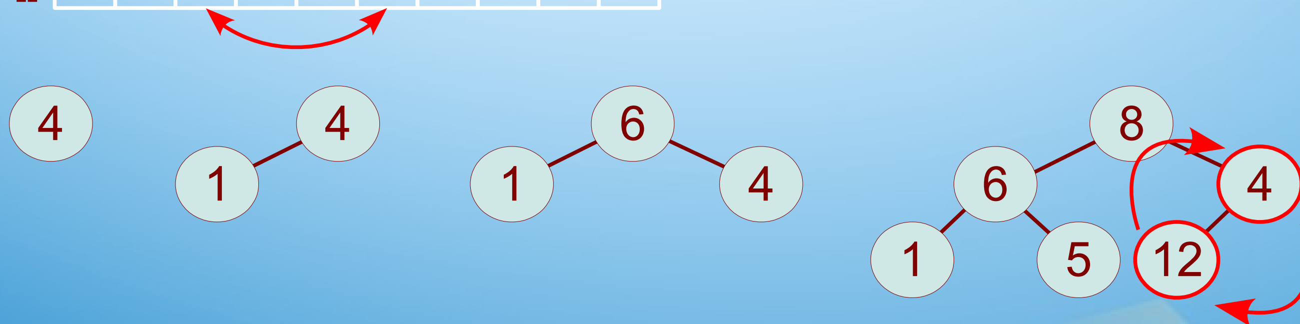
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

8	6	4	1	5	12	3	11	9	2



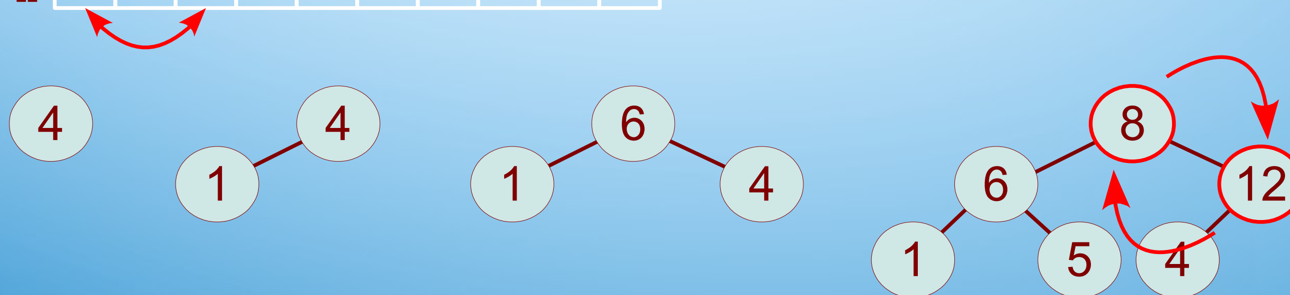
Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:



Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

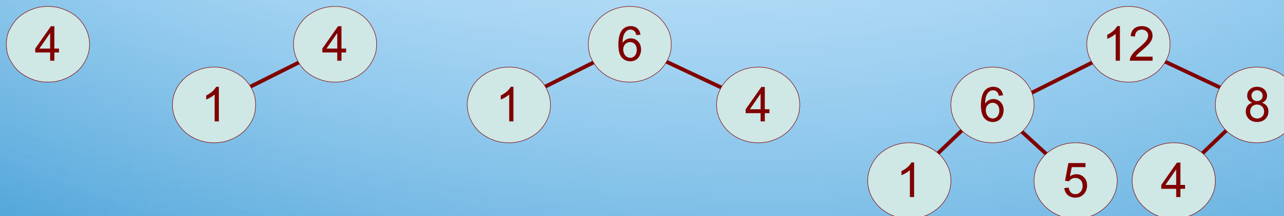
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

12	6	8	1	5	4	3	11	9	2



Etap 1 – budowa kopca.

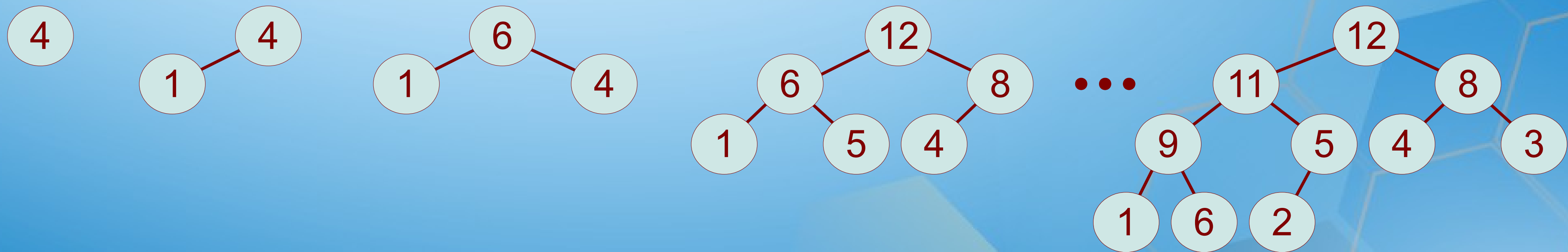
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:	12	11	8	9	5	4	3	1	6	2



Etap 1 – budowa kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

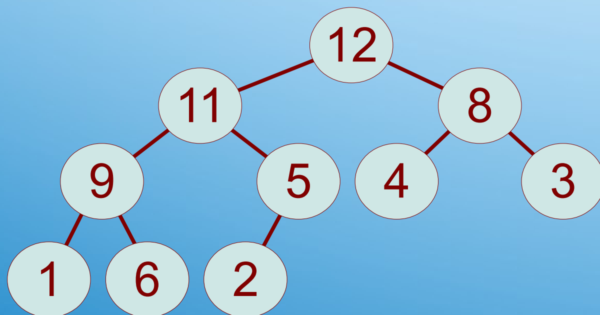
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

12	11	8	9	5	4	3	1	6	2



Etap 2 – sortowanie.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

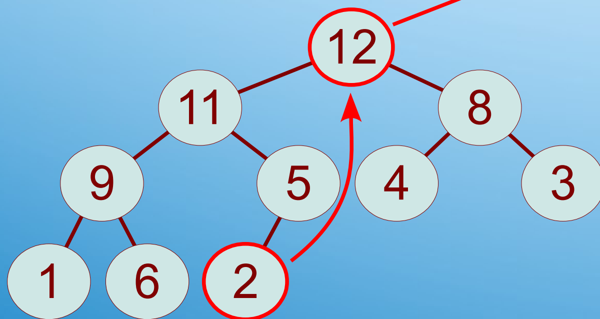
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

12	11	8	9	5	4	3	1	6	2



Etap 2 – sortowanie: przeniesienie korzenia drzewa na koniec tablicy kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

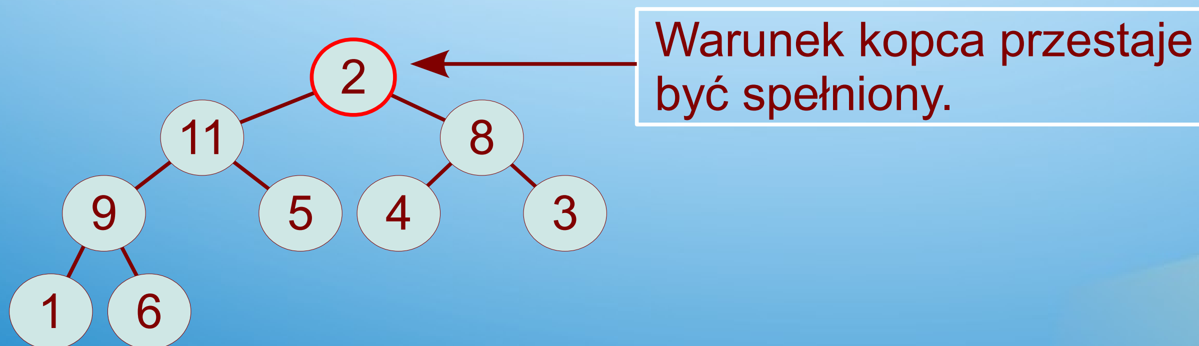
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

2	11	8	9	5	4	3	1	6	12



Etap 2 – sortowanie: przywracanie kopca.

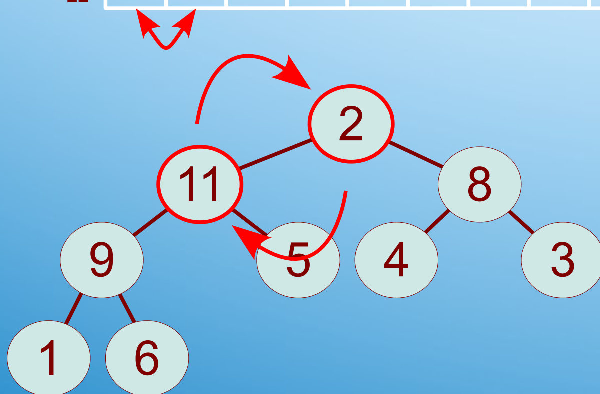
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:	2	11	8	9	5	4	3	1	6	12



Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

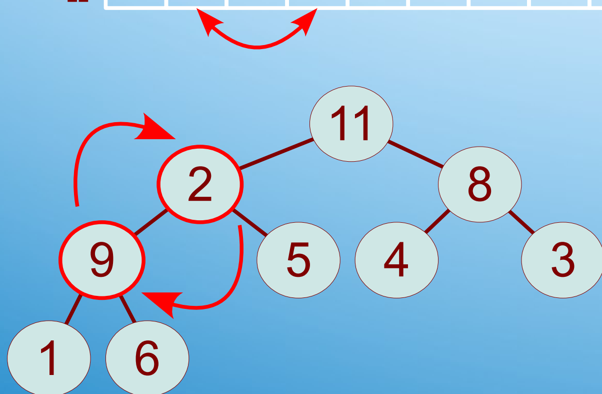
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

11	2	8	9	5	4	3	1	6	12



Etap 2 – sortowanie: przywracanie kopca.

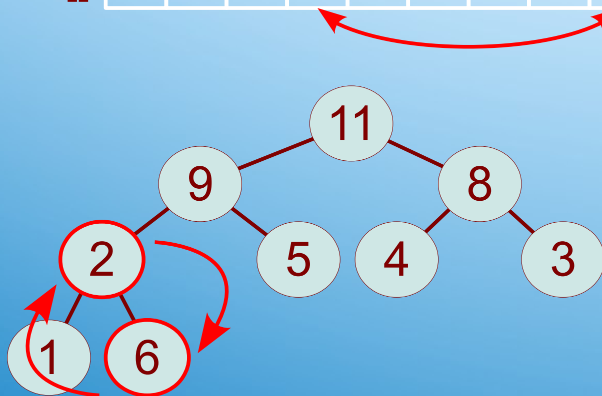
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:	11	9	8	2	5	4	3	1	6	12



Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

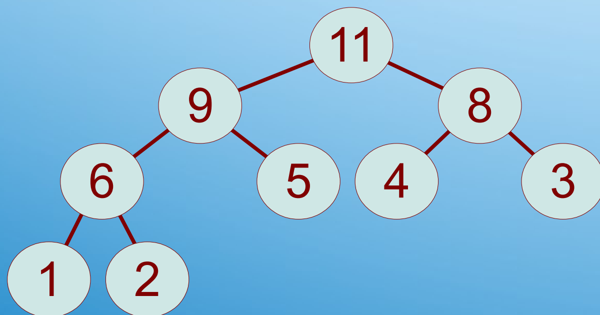
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

11	9	8	6	5	4	3	1	2	12



Kopiec został przywrócony.

Etap 2 – sortowanie.

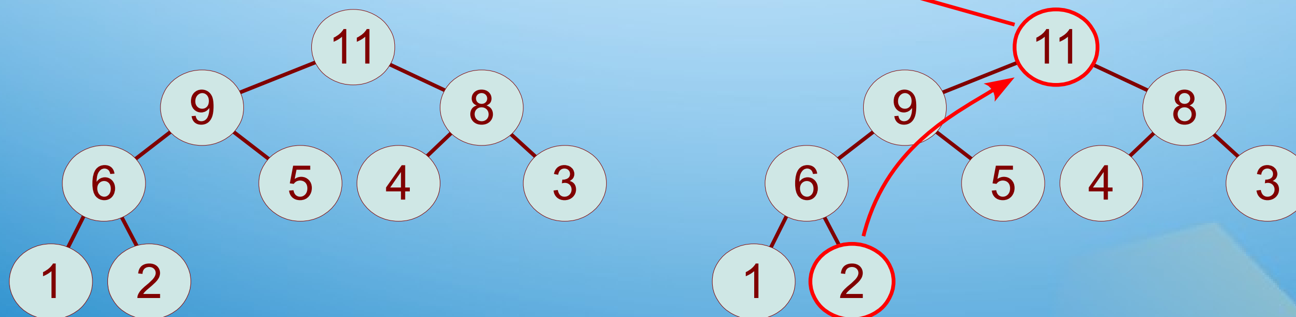
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:	11	9	8	6	5	4	3	1	2	12



Etap 2 – sortowanie: przeniesienie korzenia drzewa na koniec tablicy kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

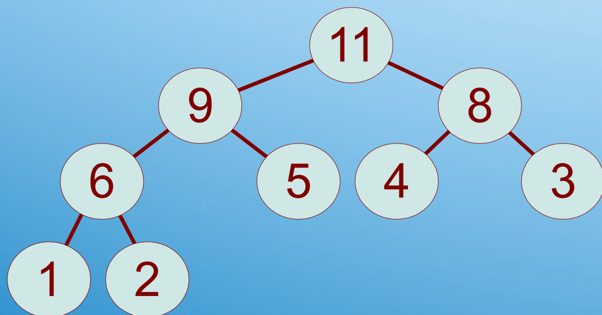
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

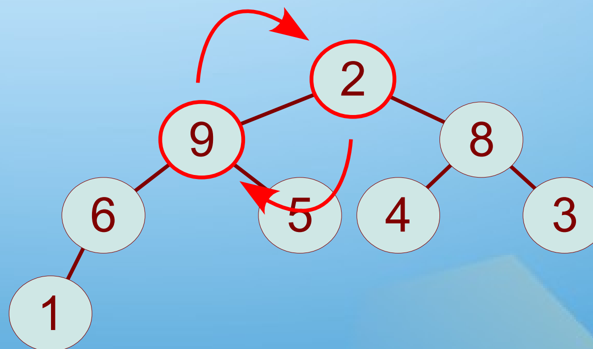
Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

2	9	8	6	5	4	3	1	11	12





Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

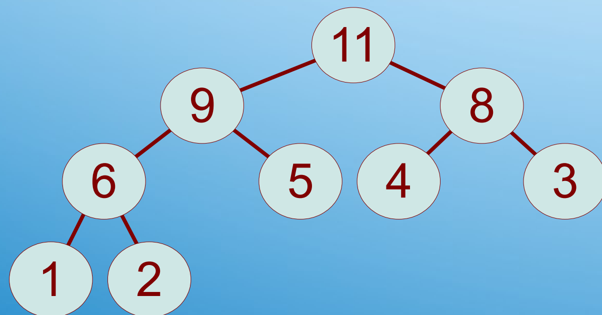
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

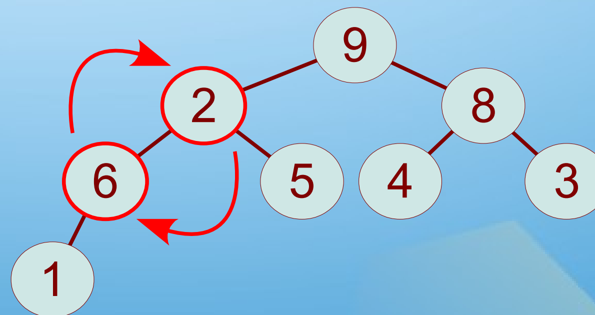
Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

9	2	8	6	5	4	3	1	11	12





Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

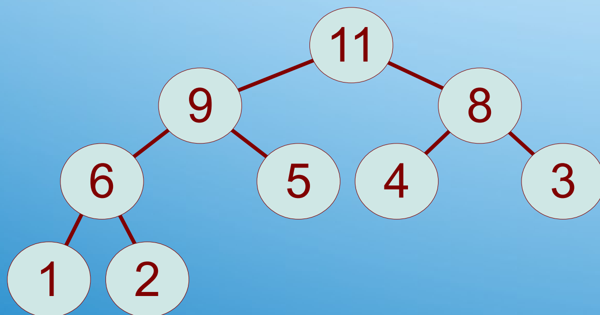
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

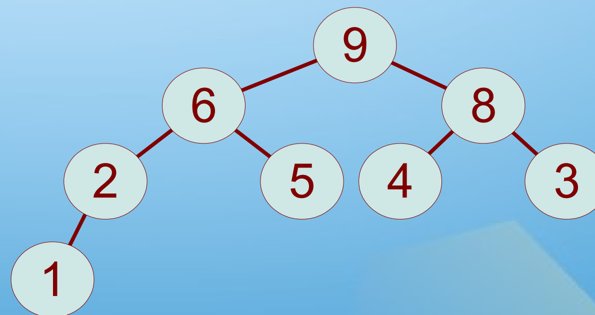
Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

9	6	8	2	5	4	3	1	11	12





Etap 2 – sortowanie.

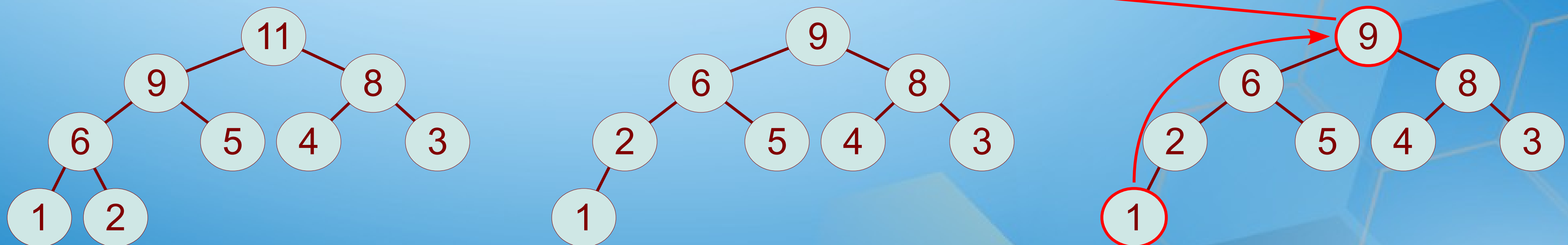
Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:	9	6	8	2	5	4	3	1	11	12



Etap 2 – sortowanie: przeniesienie korzenia drzewa na koniec tablicy kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

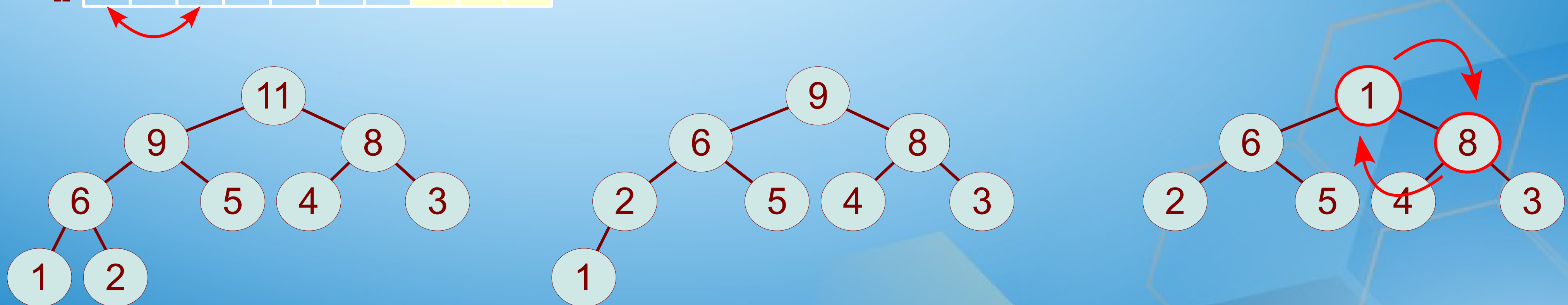
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

Przykład:

A[]:

1	6	8	2	5	4	3	9	11	12



Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

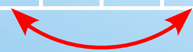
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

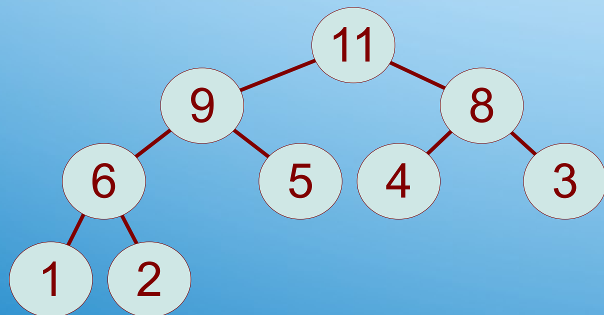
Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

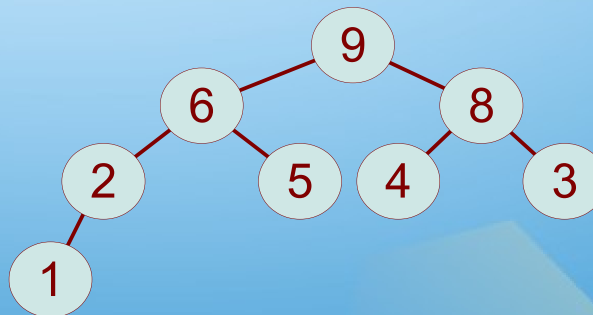
Przykład:

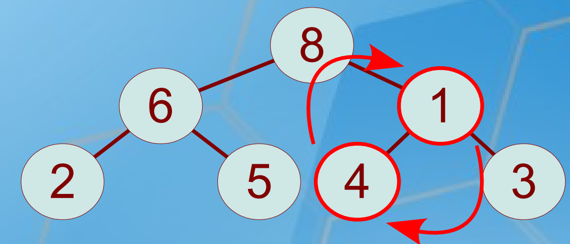
A[]:

8	6	1	2	5	4	3	9	11	12









Etap 2 – sortowanie: przywracanie kopca.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

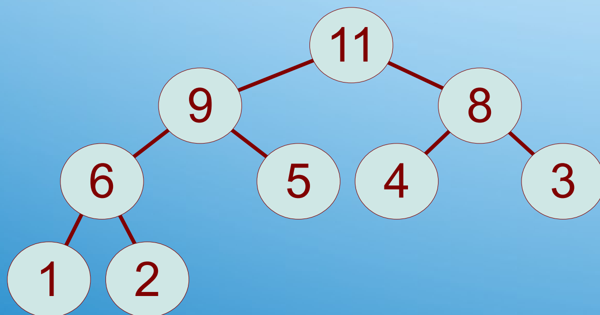
Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

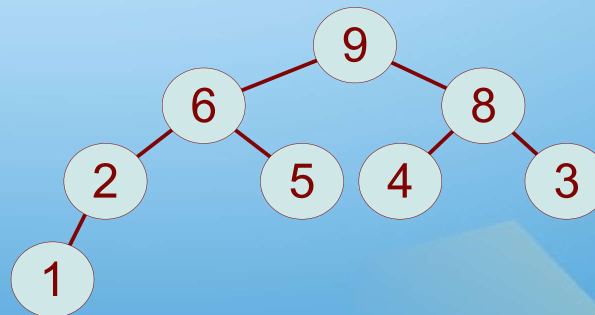
Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi O(n log n).
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi O(n^2) oraz pamięciowa O(log n).

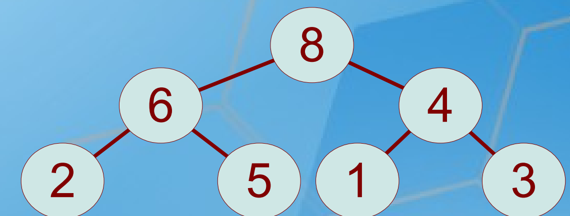
Przykład:

A[]:

8	6	4	2	5	1	3	9	11	12







Etap 2 – sortowanie.

Sortowanie przez kopcowanie – algorytm sortujący stosujący kopiec binarny.

Algorytm:
1. Reorganizacja danych w celu utworzenia kopca.
2. Właściwe sortowanie.

Cechy:
- algorytm ma stałą złożoność pamięciową O(1) – kopiec tworzy się w tablicy z danymi,
- złożoność obliczeniowa pesymistyczna wynosi $O(n \log n)$.
- algorytm stanowi doskonałą alternatywę dla quicksort'a, którego złożoność pesymistyczna jest gorsza i wynosi $O(n^2)$ oraz pamięciowa $O(\log n)$.

Przykład:

A[]:

8	6	4	2	5	1	3	9	11	12

Etap 2 – sortowanie: ...

Czynności wykonywane do końca etapu sortowania:
- przenoszenie korzenia na koniec tablicy kopca,
- wstawienie ostatniego elementu w miejsce korzenia,
- przewracanie kopca.

Koniec wykładu