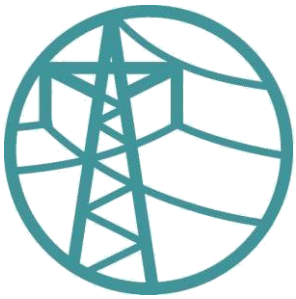


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Projekt #2 Algorytmy i struktury danych

Autor: Dawid Stachiewicz

Numer albumy: 173218

Temat: Porównanie algorytmów sortujących.

Kierunek: Inżynieria i analiza danych

Rzeszów 2022

SPIS TREŚCI

1. Sortowanie przez scalanie

1. Podstawy teoretyczne sortowania przez scalanie

Sortowanie przez scalanie (ang. merge sort) jest popularnym algorytmem sortowania danych. Jest on efektywny i zazwyczaj uważany za dobry wybór w wielu sytuacjach.

Sortowanie przez scalanie jest algorytmem dziel i zwyciężaj. W tym algorytmie dane są najpierw dzielone na coraz mniejsze części, aż osiągną rozmiar jednego elementu, co oznacza, że każdy taki element jest posortowany. Następnie posortowane elementy są scalane w coraz większe części, aż osiągną rozmiar całego zbioru danych.

Sortowanie przez scalanie jest stabilnym algorytmem, co oznacza, że elementy o takiej samej wartości pozostają w tej samej kolejności po sortowaniu, jak miało to miejsce w oryginalnej sekwencji. Jest to ważna cecha algorytmu, ponieważ pozwala na zachowanie relacji między elementami o takiej samej wartości w oryginalnej sekwencji danych.

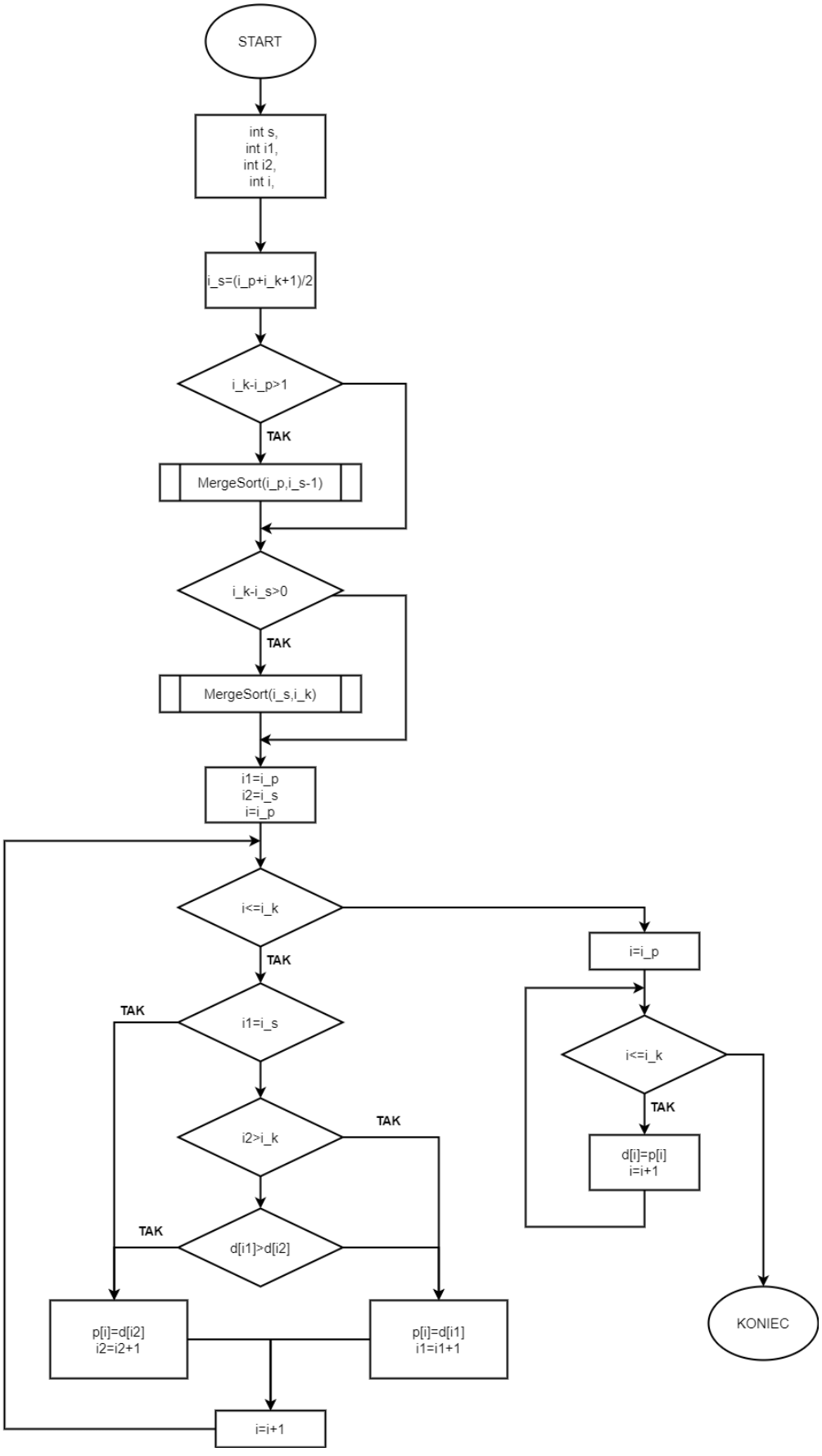
Sortowanie przez scalanie jest również wydajne, ponieważ wykorzystuje podział danych na mniejsze części, co pozwala na szybkie sortowanie dużych zbiorów danych. Algorytm jest złożonościowy $\boldsymbol{n}(\boldsymbol{\log n})$,co oznacza, że wydajność jest dobra w przypadku dużych zbiorów danych.

2. Kod funkcji MergeSort()

```
int d[N],p[N];

//p[ ]  - zbiór pomocniczy, który zawiera tyle samo elementów, co zbiór d[ ].

// Procedura sortująca
//--------------------

void MergeSort(int i_p, int i_k)
{
  int i_s,i1,i2,i;

//i1    - indeks elementów w młodszej połówce zbioru d[ ],
//i2    - indeks elementów w starszej połówce zbioru d[ ]
//i - indeks elementów w zbiorze pomocniczym p[ ]

//ip    - indeks pierwszego elementu w młodszym podzbiorze
//ik    - indeks ostatniego elementu w starszym podzbiorze

  i_s = (i_p + i_k + 1) / 2;
  if(i_s - i_p > 1) MergeSort(i_p, i_s - 1);
  if(i_k - i_s > 0) MergeSort(i_s, i_k);
  i1 = i_p; i2 = i_s;
  for(i = i_p; i <= i_k; i++)
    p[i] = ((i1 == i_s) || ((i2 <= i_k) && (d[i1] > d[i2]))) ? d[i2++] : d[i1++];
  for(i = i_p; i <= i_k; i++) d[i] = p[i];
}
```

Rysunek 1 - kod funkcji MergeSort()

3. Schemat blokowy sortowania przez scalanie



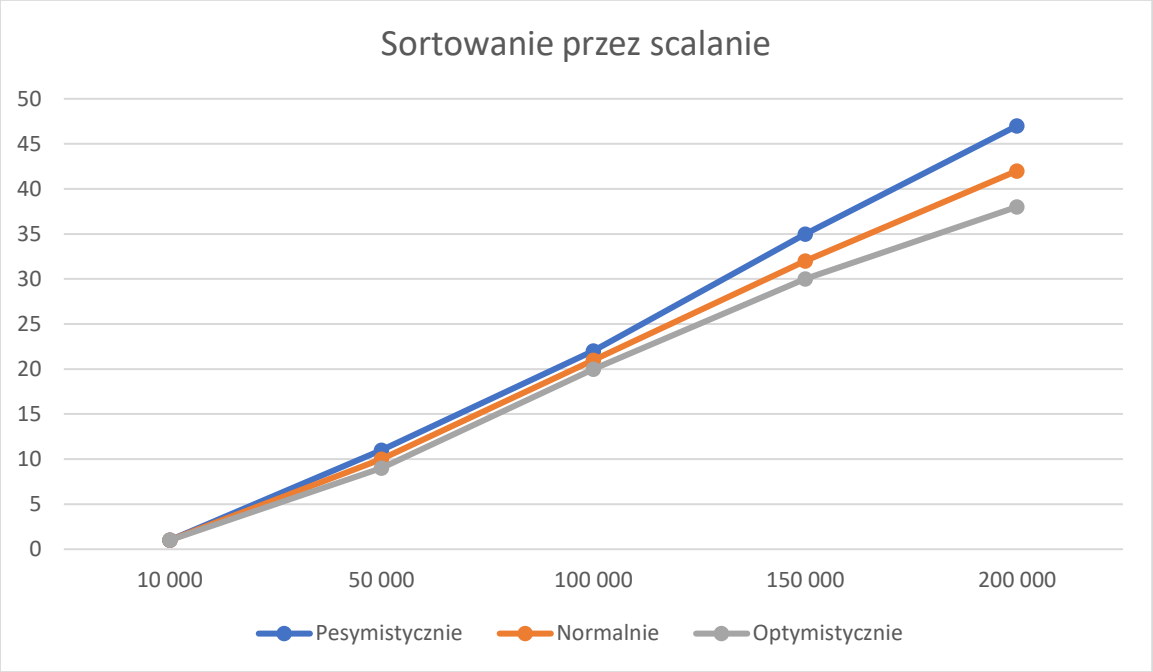
4. Pseudokod funkcji sortowania przez scalanie

```
Przypisz zmiennej i_s wartość [(i_p+i_k+1)/2]
Jeśli i_s - i_p > 1, to Sortuj_przez_scalanie(i_p, i_s - 1)
Jeśli i_k - i_s > 0, to Sortuj_przez_scalanie(i_s, i_k)
Przypisz zmiennym i1 wartość i_p; i2 wartość i_s; i wartość i_p
Scalaj(i_p, i_s, i_k)
Dla i = ip, ip + 1, ..., ik:
    jeśli (i1 = is) ? (i2 ? ik  i d[i1] > d[i2]),
    to p[i] ‹ d[i2];   i2 ‹ i2 + 1
    inaczej p[i] ‹ d[i1];   i1 ‹ i1 + 1
Dla i = ip, ip + 1,...,ik:
    d[i] ‹ p[i]
Zakończ
```

Rysunek 2 - pseudokod sortowania przez scalanie

5. Wykres złożoności obliczeniowej sortowania przez scalanie.

Wykres przedstawia zależność czasu od ilości rekordów w tabeli sortowanej. Czas przedstawiony jest w sekundach.



Wykres sortowania przez scalanie

2. Złożoność obliczeniowa w przypadku pesymistycznym.

Złożoność obliczeniowa sortowania przez scalanie w przypadku pesymistycznym to również **O(n log n).** Sortowanie przez scalanie jest algorytmem divide-and-conquer, co oznacza, że dzieli on dane wejściowe na mniejsze części, sortuje je niezależnie, a następnie scalaj je w całość

3. Złożoność obliczeniowa w przypadku optymistycznym:

Złożoność obliczeniowa sortowania przez scalanie w przypadku optymistycznym również wynosi **O(n log n).** W przypadku optymistycznym mówimy o sytuacji, w której dane wejściowe są już w pełni posortowane, co oznacza, że algorytm nie musi przeprowadzać żadnych operacji scalania, aby je posortować.

2. Sortowanie kopcowe

1. Podstawy teoretyczne sortowania kopcowego

Sortowanie kopcowe to algorytm sortujący, który działa na zasadzie budowania kopca. Kopiec to specjalny rodzaj struktury danych, w której każdy węzeł posiada co najwyżej dwóch potomków, ale każdy potomek jest większy (lub mniejszy, w zależności od kierunku sortowania) niż jego rodzic.

Sortowanie kopcowe jest jednym z popularnych algorytmów sortowania, który ma kilka zalet. Jedną z najważniejszych jest to, że jest ono bardzo efektywne w przypadku dużych zbiorów danych.

Sortowanie kopcowe jest jednym z popularnych algorytmów sortowania, który ma kilka zalet w porównaniu do innych algorytmów. Oto kilka z tych zalet:

1. **Wydajność** - sortowanie kopcowe jest wydajnym algorytmem, który działa w czasie O(n * log(n)). Oznacza to, że wydajność tego algorytmu zwiększa się wraz z rozmiarem danych w liniowym tempie, co czyni go dobrym wyborem dla dużych zbiorów danych.

2. **Prostota implementacji** - sortowanie kopcowe jest łatwe do zaimplementowania, a jego kod jest łatwy do zrozumienia i debugowania.

3. **Niezawodność** - sortowanie kopcowe jest niezawodne i zawsze zwraca posortowany zbiór danych, niezależnie od tego, jakie dane są na wejściu.

4. **Możliwość dodawania elementów** - sortowanie kopcowe pozwala na dynamiczne dodawanie elementów do posortowanego zbioru danych, bez konieczności ponownego sortowania całego zbioru.

2. Kod funkcji heapify() [kopcowania]

```
void heapify(int tab[], int N, int i)   //N to rozmiar kopca
{

    int najwiekszy = i;     // Inicjalizujemy 'największy' jako korzeń

    int le = 2 * i + 1;     // lewy = 2i + 1

    int pr = 2 * i + 2;     // prawy = 2i + 2

    if (le < N && tab[le] > tab[najwiekszy])    // Jeśli lewe 'dziecko' jest większe niż korzeń
        najwiekszy = le;

    if (pr < N && tab[pr] > tab[najwiekszy])    // Jeśli prawe 'dziecko' jest większe niż korzeń
        najwiekszy = pr;

    if (najwiekszy != i) {      //Jeśli 'największy' nie jest korzeniem
        swap(tab[i], tab[najwiekszy]);

        heapify(tab, N, najwiekszy);     // Rekurencyjnie tworzymy kopiec w zmienionym pod-drzewie
    }
}
```

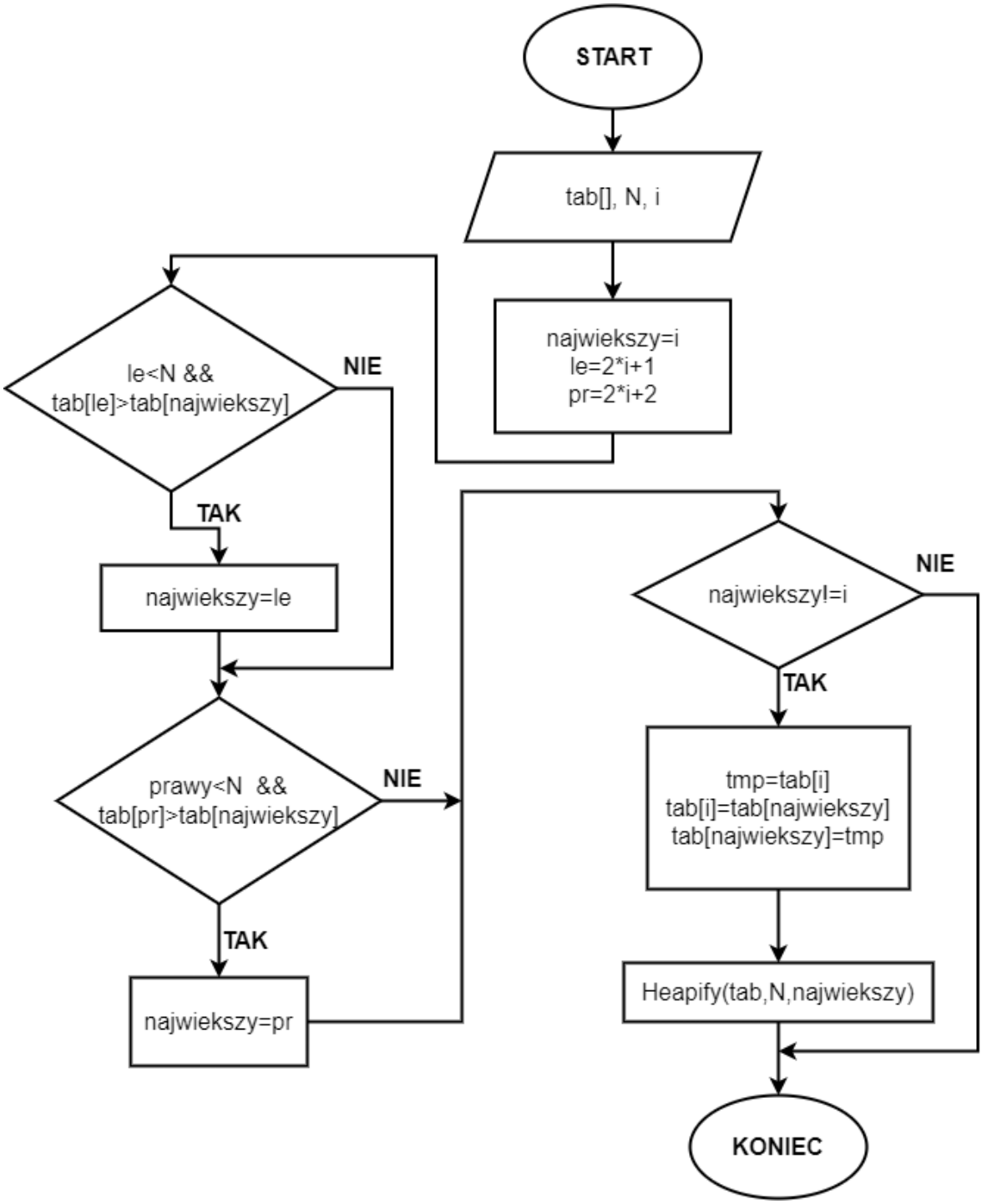
Rysunek 4 - kod funkcji heapify()

3. Kod funkcji heapSort()

```
void heapSort(int tab[], int N)
{

    for (int i = N / 2 - 1; i >= 0; i--)    //Budowanie kopca
    {
        heapify(tab, N, i);     //wywołanie funkcji heapify
    }
    for (int i = N - 1; i > 0; i--) //Sortowanie tablicy
    {
        swap(tab[0], tab[i]);   //Zamiana pierwszego elementu w tablicy na ostatni

        heapify(tab, i, 0); //Wywołanie funkcji heapify() dla nowego pierwszego elementu tablicy
    }
}
```

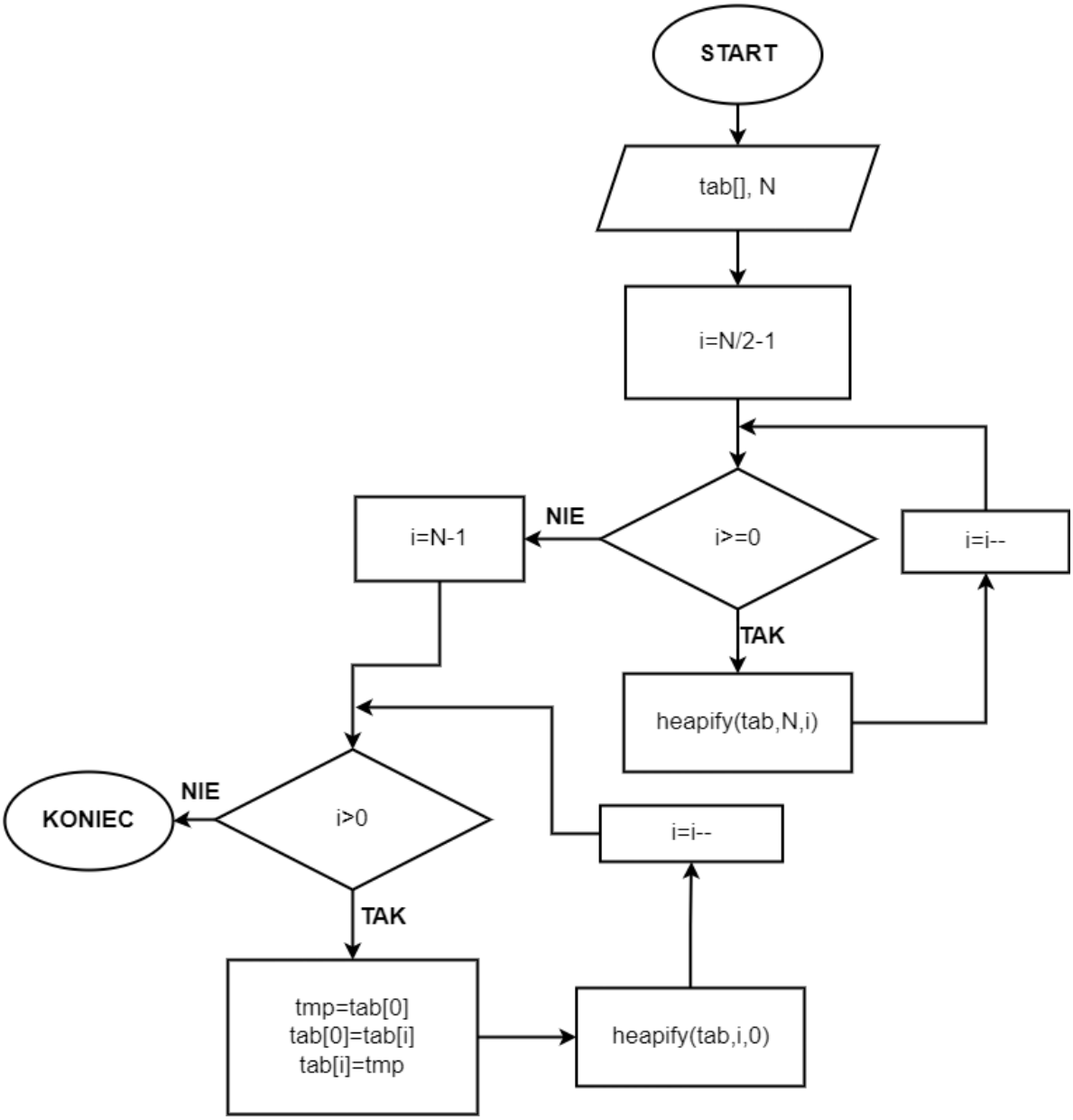
Rysunek 5 Kod funkcji heapSort()

4. Schemat blokowy funkcji heapify()



Rysunek 6 - schemat blokowy funkcji heapify()

4. Schemat blokowy funkcji heapSort()



Rysunek 7 Schemat blokowy funkcji heapSort()

5. Pseudokod funkcji heapify()

```
void heapify(int tab[], int N, int i)
{
// N to rozmiar kopca
// Inicjalizujemy 'największy' jako korzeń
int najwiekszy = i;


// lewy = 2i + 1
int le = 2 * i + 1;

// prawy = 2i + 2
int pr = 2 * i + 2;

// Jeśli lewe 'dziecko' jest większe niż korzeń
if (le < N && tab[le] > tab[najwiekszy])
    najwiekszy = le;

// Jeśli prawe 'dziecko' jest większe niż korzeń
if (pr < N && tab[pr] > tab[najwiekszy])
    najwiekszy = pr;

// Jeśli 'największy' nie jest korzeniem
if (najwiekszy != i) {
    // Zamieniamy 'największy' z korzeniem
    swap(tab[i], tab[najwiekszy]);

    // Rekurencyjnie tworzymy kopiec w zmienionym pod-drzewie
    heapify(tab, N, najwiekszy);
}
}
```

Rysunek 7 - pseudokod funkcji heapify()

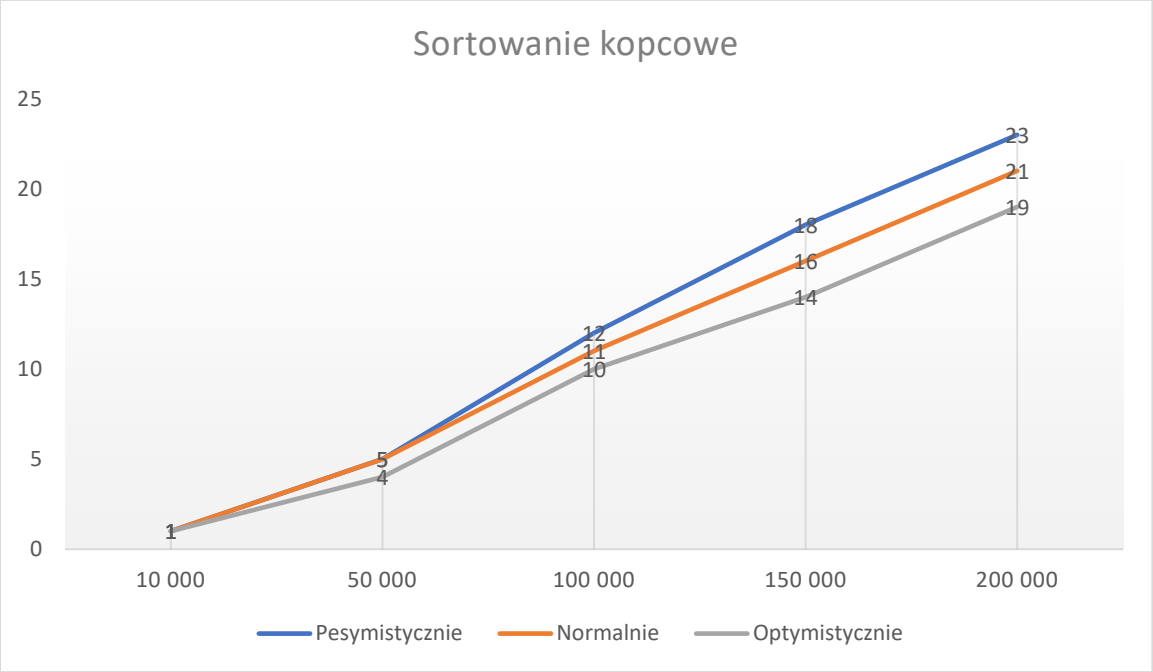
6. Pseudokod funkcji heapSort()

```
funkcja void heapSort(int tab[], int N):
    dla i od N/2 - 1 do 0:
        wywołaj funkcję heapify(tab, N, i)

    dla i od N - 1 do 0:
        zamień pierwszy element tablicy z ostatnim
        wywołaj funkcję heapify(tab, i, 0)
```

Rysunek 8 – Pseudokod funkcji heapSort()

7. Wykres złożoności czasowej sortowania kopcowego

Wykres przedstawia zależność czasu od ilości rekordów w tabeli sortowanej. Czas przedstawiony jest w sekundach.



Wykres sortowania kopcowego

1. Złożoność obliczeniowa w przypadku pesymistycznym.

Złożoność obliczeniowa sortowania kopcowego w przypadku pesymistycznym jest **O(n log n)**, gdzie n to liczba elementów do posortowania.

2. Złożoność obliczeniowa w przypadku optymistycznym:

Złość obliczeniowa sortowania kopcowego w przypadku optymistycznym to **O(n log n).** W tym przypadku mówimy o sytuacji, w której kopiec jest zawsze w pełni zbalansowany, co oznacza, że wszystkie węzły w drzewie są wypełnione zgodnie z algorytmem budowania kopca.

3. Wnioski

Analizując obie metody sortowania, można dojść do następujących wniosków:

Sortowanie kopcowe i sortowanie przez scalanie to dwa różne algorytmy sortowania danych. Algorytm sortowania kopcowego wykorzystuje strukturę danych zwaną kopcem, aby posortować dane. Algorytm sortowania przez scalanie natomiast wykorzystuje podejście dziel i zwyciężaj, tworząc kilka mniejszych posortowanych ciągów i scalając je w jeden finalny posortowany ciąg.

Jedną z głównych różnic między tymi algorytmami jest sposób działania. Sortowanie kopcowe jest algorytmem in-place, co oznacza, że nie potrzebuje dodatkowej pamięci do sortowania danych. Sortowanie przez scalanie natomiast jest algorytmem out-of-place, co oznacza, że wymaga dodatkowej pamięci do przechowywania posortowanych danych.

Kolejną ważną różnicą jest szybkość działania tych algorytmów. Sortowanie kopcowe jest zazwyczaj szybsze od sortowania przez scalanie dla dużych zbiorów danych, ponieważ wymaga mniejszej liczby operacji porównania i przestawienia danych.