2021

Projekt

PORÓWNANIE SORTOWAŃ: PRZEZ SCALANIE ORAZ PRZEZ KOPCOWANIE

JAKUB KUŹNIAR GRUPA 3

Spis treści

1. Wstęp

Porównanie dwóch algorytmów sortowania przez scalanie oraz kopcowanie.

1.1. Sortowanie przez scalanie

Jest to algorytm rekurencyjny, wykorzystujący zasadę dziel i zwyciężaj, która polega na podzieleniu problemu na mniejsze części przez co jest łatwiejszy do rozwiązania. Algorytm sortujący dzieli zbiór na kolejne połowy dopóki podział jest możliwy. Następnie algorytm sortuje rekurencyjnie otrzymane zbiory, później je łączy za pomocą scalania, przez co zbiór końcowy jest posortowany.

1.1.1. Złożoność obliczeniowa sortowania przez scalanie

Złożoność obliczeniowa to $O(n * log(n))$ (z indukcji matematycznej)

1.1.2. Szczegóły implementacji

Algorytm dzieli dany zbiór na połówki, następnie je sortuje i scala.
Funkcja MergeSort(int i_p, int i_k, int tab[], int pomoc[]) przyjmuje 4 argumenty: i_p to index pierszego elementu w młodszym podzbiorze, i_k to index ostatniego elementu w starszym podzbiorze, tab[] to posortowany zbiór, pomoc[] to zbiór pomocniczy.

```
void MergeSort(int i_p, int i_k, int tab[], int pomoc[])
{
    int i_s, i1, i2, i;
    //i_s to index pierwszego elemnetu w starszym podzbiorze, i1 to index elementó
    w w młodszej połówce zbioru
    // i2 to index elementów w starszej połówce zbioru, i to index elementów w zbi
    orze pomocniczym
    i_s = (i_p + i_k + 1) / 2; // wyznaczamy index który jest wykorzysytwany do po
    działu zbioru na dwie części
    if (i_s - i_p > 1) MergeSort(i_p, i_s - 1, tab, pomoc);//sprawdzamy czy jest w
    ięcej niż jeden element, jeżeli tak to sortujemy go tym samym algorytmem
    if (i_k - i_s > 0) MergeSort(i_s, i_k, tab, pomoc); // sortujemy drugą połówkę
    //
    i1 = i_p;
    i2 = i_s;
    // sprawdzamy czy i1 i i2 wskazują elemnety podzbiorów, jeżeli któryś z nich
    wyszedł poza dopuszczalny zakres to zbiór jest wyczerpany
    // w takim przypadku do tablicy pomoc przypisujemy elementy drugiego zbioru
    // jeżeli żaden z podzbiorów nie jest wyczeprany to porównujemy kolejne
    elementy podzbiorów według index'ów i1 i i2
    // do tablicy pomoc zapisujemy zawsze mniejszy element
    // pętla jest kontynuowana aż do zapełnienia tablicy pomoc[]
    for (i = i_p; i <= i_k; i++)
        pomoc[i] = ((i1 == i_s) || ((i2 <= i_k) && (tab[i1] > tab[i2]))) ?
        tab[i2++] : tab[i1++];
    for (i = i_p; i <= i_k; i++) tab[i] = pomoc[i];
    //przypisujemy wartości tablicy pomoc do tablicy tab
}
```

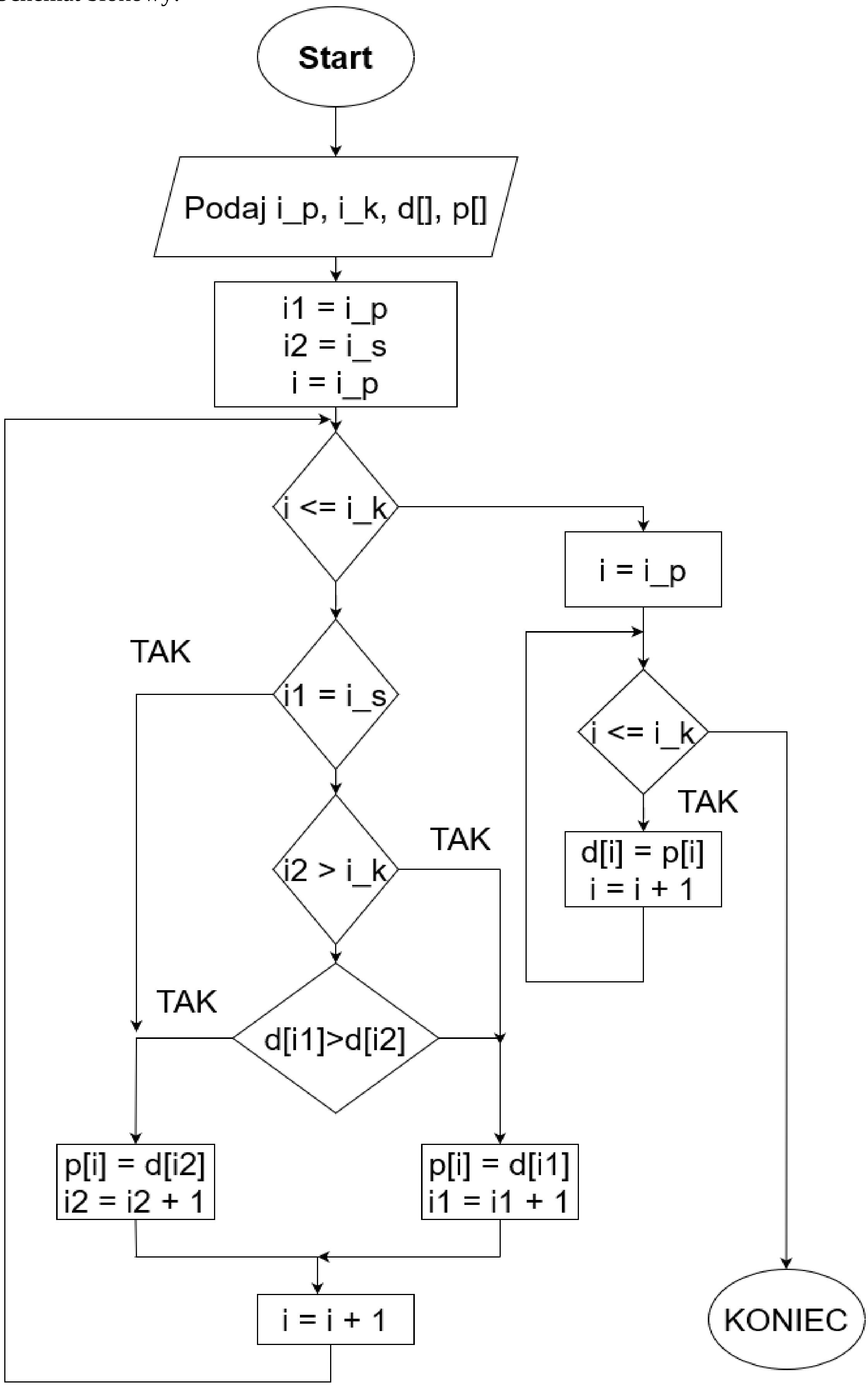
Rys. 1 Algorytm do dzielenia, sortowania oraz scalania

Program zawiera także możliwość odczytywania z pliku, losowania do pliku oraz zapisywania wyników do innego pliku.

```
// funckja służąca do otworzenia pliku i zapełnienia go losowyi liczbami
// arguemnty funkcji następująco to nazwa pliku, tablica do której mamy przepisać
dane z pliku, żeby wykonywać kolejne operacje oraz rozmiar czyli ile liczb losujem
y do pliku
void openAndPopulateFile(string file_name, int tab[], int rozmiar) {
    fstream plik;
    int value; //zmienna pomocnicza
    plik.open(file_name + ".txt", ios::in | ios::out); //otworzenie pliku
    if (plik.good() == true)
    {
        for (int i = 0; i < rozmiar; i++) {
            value = rand() % 31; // losowanie liczb
            plik << value << endl; // wpisywanie liczb do pliku
            //cout <<value<< endl;
            tab[i] = value; // przypisywanie danych z pliku do tablicy
        }

        plik.close(); // zamknięcie pliku
    }
}
// funkcja służąca do wypisania posortowanych danych oraz zapisanie ich do pliku
// argumenty to nazwa pliku, tablica która będzie przechowywać posortowane dane, r
ozmiar tablicy
void saveResultsToFile(string file_name, int tab[], int rozmiar) {
    fstream plik;
    plik.open(file_name + ".txt", ios::in | ios::out);//otworzenie pliku
    if (plik.good() == true) {
        cout << "Po sortowaniu:\n\n";
        for (int i = 0; i < rozmiar; i++) {
            plik << tab[i] << endl; // wpisujemy posortowane dane do pliku
            cout << setw(4) << tab[i];// wypisujemy posortowane dane
        }
        plik.close();// zamknięcie pliku
    }
}
```

Rys. 2 Algorytm służący do zapisu oraz odczytu plików

Schemat blokowy:



Rys. 3 Schemat blokowy algorytmu sortowania przez scalanie

Pseudokod:

```
K01:
        i 1= i_p; i2 = i_s, i = i_p
K02:
Dla i = i_p, i_p + 1, ..., i_k: wykonuj
                jeśli (i1 = i_s) ∨ (i2 ≤ i_k i d[i1] > d[i2]), to
                        p[i] ← d[i2]; i2 ← i2 + 1
                inaczej
                        p[i] ← d[i1]; i1 ← i1 + 1
K03:
        Dla i = i_p, i_p + 1,...,i_k: d[i] ← p[i]
K04:
Zakończ
```

1.1.3. Wykres oraz test dla algorytmu sortowania poprzez scalanie

Algorytm posiada jedną złożoność obliczeniową dla każdego z 3 przypadków dlatego wykonam jeden wykres, który będzie dotyczył pesymistycznego, oczekiwanego oraz optymistycznego przypadku.

- Dla 20 elementów

```
 25   1  27   9  17   4  23  28  14  17  27  18  26   0  30  18  28   5   4  10
Po sortowaniu:

  0   1   4   4   5   9  10  14  17  17  18  18  23  25  26  27  27  28  28  30
Czas w mikrosekundach: 31984
```

Rys. 4 Test dla 20 elementów

- Dla 100 elementów

```
  9   4   0  12   5  29  18  19  27  27  15  14   2   7  27  14   6   4  30  20  20   3  26  11  26  19  20  30  19  23
 29   3  12  23  17   9   1  30   6  23   7  24   9  21  11  27  28  23   3   6  30  30   9  18  10  22  25  30   9   2
 29   9  18  26  13   6  26  28   2   7  19  24  25  21   6  28  14   4  17   9  12  21  19   6   6   8  15  16  28  24
  9  17  17   2  14  27   5   3  14  19
Po sortowaniu:

  0   1   2   2   2   2   3   3   3   3   4   4   4   5   5   6   6   6   6   6   6   6   7   7   7   8   9   9   9   9
  9   9   9   9  10  11  11  12  12  12  13  14  14  14  14  14  15  15  16  17  17  17  17  18  18  18  19  19  19  19
 19  19  20  20  20  21  21  21  22  23  23  23  23  24  24  24  25  25  26  26  26  26  27  27  27  27  27  28  28  28
 28  29  29  29  30  30  30  30  30  30
Czas w mikrosekundach: 23973
```

Rys. 5 Test dla 100 elementów

- Dla 1000 elementów (dla zaoszczędzenia papieru oraz miejsca będę uwzględniał tylko czas wykonania algorytmu)

```
 29  29  29  30  30  30  30  30  30  30  30  30
 30  30  30  30  30  30  30  30  30  30
Czas w mikrosekundach: 210023
```

Rys. 6 Test dla 1000 elementów

- Dla 10000 elementów

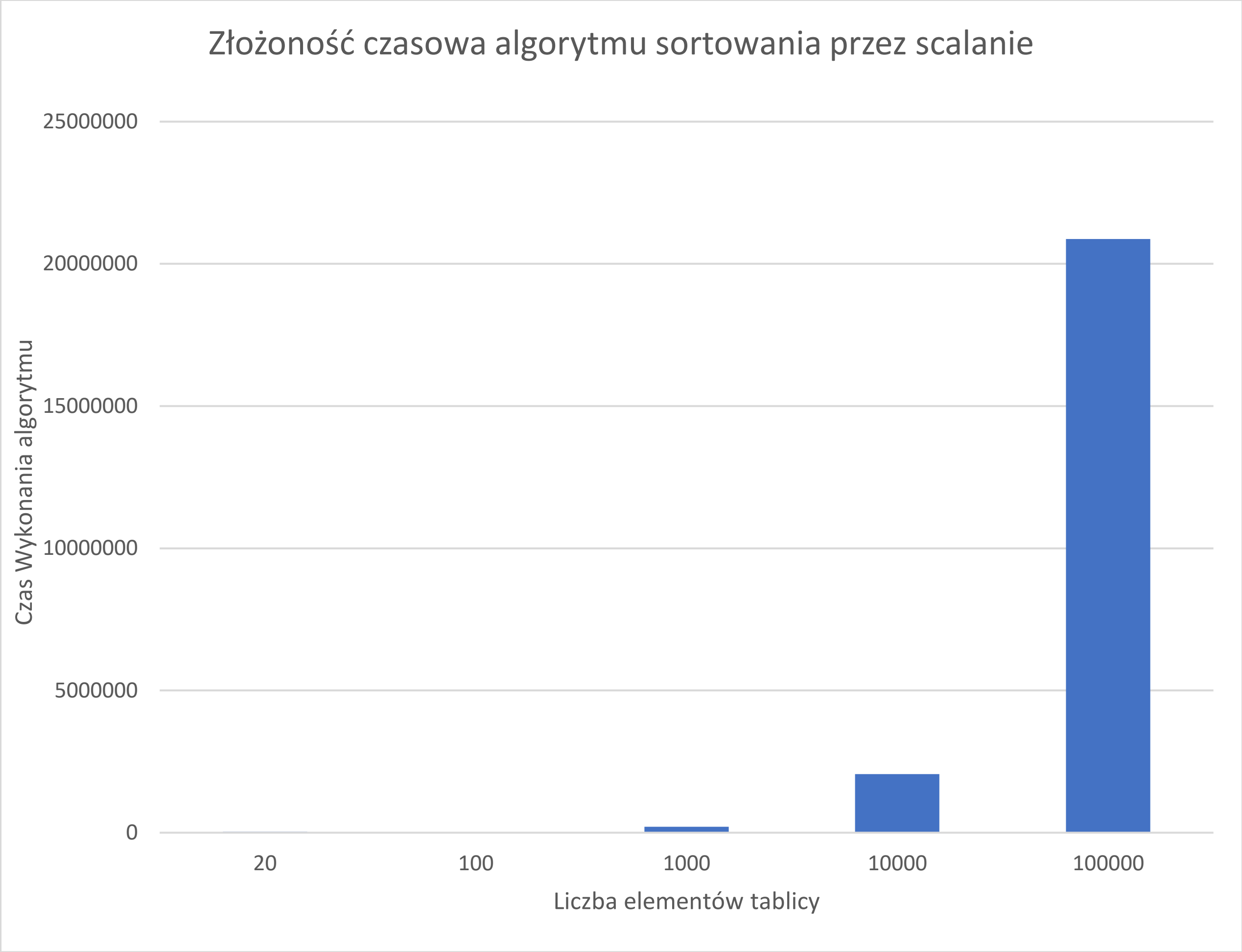
```
 30  30  30  30  30  30  30  30  30  30
Czas w mikrosekundach: 2056708
```

Rys. 7 Test dla 10000 elementów

- Dla 100000 elementów

```
 30  30  30  30  30  30  30  30  30  30
Czas w mikrosekundach: 20869472
```

Rys. 8 Test dla 100000 elementów



Wykres 1 Złożoność czasowa dla algorytmu sortowania przez scalanie

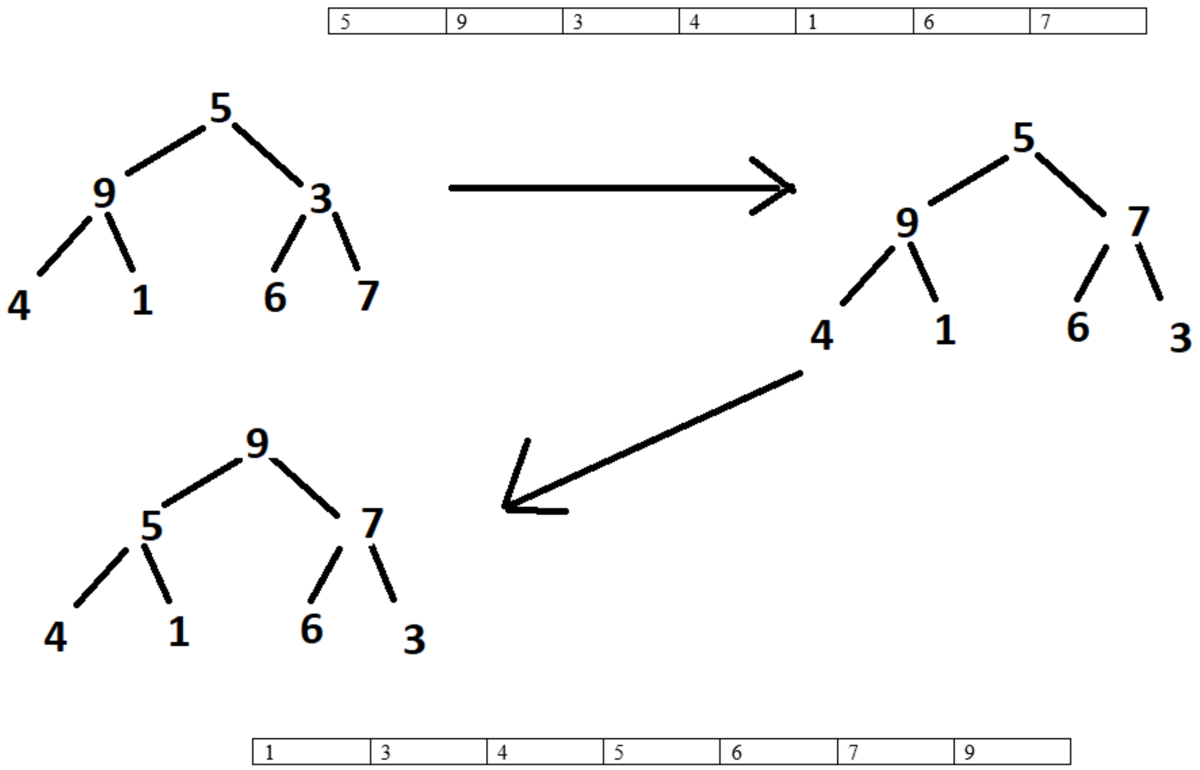
1.2. Sortowanie przez kopcowanie

Zadaniem algorytmu jest zbudowanie kopca, posortowanie danych z kopca i rozebranie go.
Kopiec jest drzewem binarnym, w którym wszystkie węzły spełniają następujący warunek:

- Węzeł nadrzędny jest większy lub równy węzłom potomnym

Kopiec jest bardziej skomplikowany niż drzewo binarne ponieważ po dołączeniu nowego elementu musimy sprawdzać czy zachodzi warunek kopca.
Charakterystyczną cechą kopca jest to, iż korzeń zawsze jest największym (w porządku malejącym) elementem z całego drzewa.
Rozbiór kopca jest kolejna czynnością którą musimy wykonać. Zamieniamy miejscami korzeń z ostatnim liściem, który wyłączamy ze struktury kopca. Elementem pobieranym jest zawsze największy czyli jest korzeniem. Należy pamiętać o warunku i sprawdzać czy zachodzi. Powtarzamy aż kopiec będzie pusty.

Przykładowe działanie algorytmu:



Rys. 9 Przykładowe działanie algorytmu sortowania przez kopcowanie

1.2.1. Złożoność obliczeniowa algorytmu sortowania przez kopcowanie

Złożoność obliczeniowa dla sortowania przez kopcowanie:

- Przypadek pesymistyczny - $O(n\ log\ n)$
- Przypadek optymistyczny - $O(n)$ dla zbioru tych samych elementów

1.2.2. Przedstawienie kodu oraz schematów blokowych dla sortowania przez kopcowanie

Ponieważ cały algorytm składa się z dwóch różnych funkcji (budowania oraz rozbioru kopca) przedstawię dwie funkcje, dwa schematy blokowe oraz dwa pseudokody.

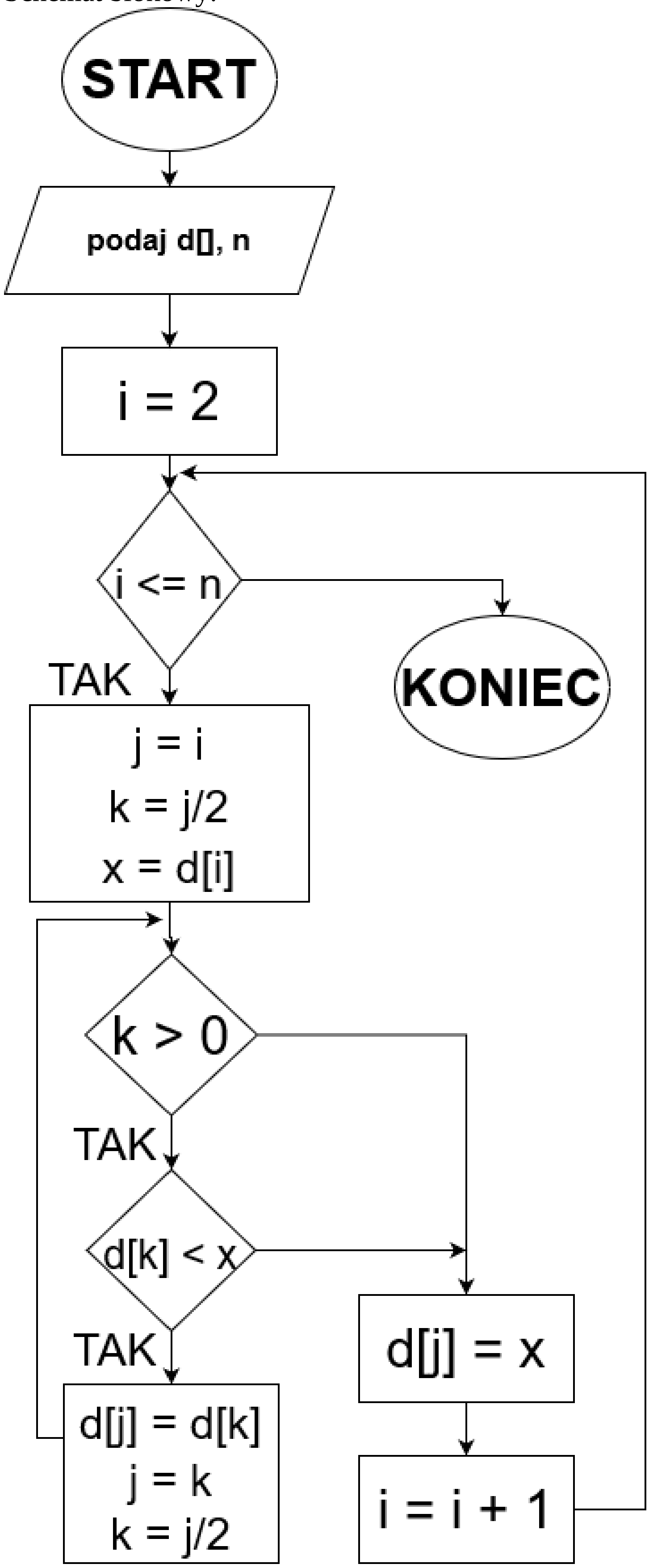
1) Budowa kopca

a) Kod źródłowy:

```
void buildHeap(int tab[], int rozmiar) {
    int j, k, x;
    for (int i = 2; i <= rozmiar; i++) {//ta pętla służy do wyznaczania kolejnych
    elementów wstawianych do kopca, i = 2 ponieważ pierwszy element zostaje na
    swoim miejscu
        j = i; // pozycja wstawianego elementu
        k = j / 2; // pozycja elementu nadrzędnego (przodka)
        x = tab[i]; // zapamiętuje wstawiany element
        while ((k > 0) && (tab[k] < x)) { //zadaniem tej pętli jest znalezienie w
        kopcu miejsca żeby wstawić zapamiętany element
          //wykonuje się do osiągnięcia korzenia kopca(k = 0) lub znalezienia więk
        szego przodka od zapamiętanego elementu
          // przesuwamy przodka na mejsce potomka, aby zachować warunek kopca, a
        następnie przesuwamy pozycję j na zajmowana wcześniej przez przodka, k
        staje się pozycją nowego przodka
            tab[j] = tab[k];
            j = k;
            k = j / 2;
        }
        tab[j] = x; //tablica z miejscem gdzie należy umieścić element x
    }
}
```

Rys. 10 Kod źródłowy budowy kopca

b) Schemat blokowy:



Rys. 11 Schemat blokowy budowy kopca

c) Pseudokod:

K01: Dla i = 2, ..., n: wykonuj K02...K05
K02: j ← i; k ← j div 2
K03: x ← d[i]
K04: Dopóki (k > 0) ∧ (d[k] < x): wykonuj
d[j] ← d[k]
j ← k
k ← j div 2
K05: d[j] ← x
K06: Zakończ

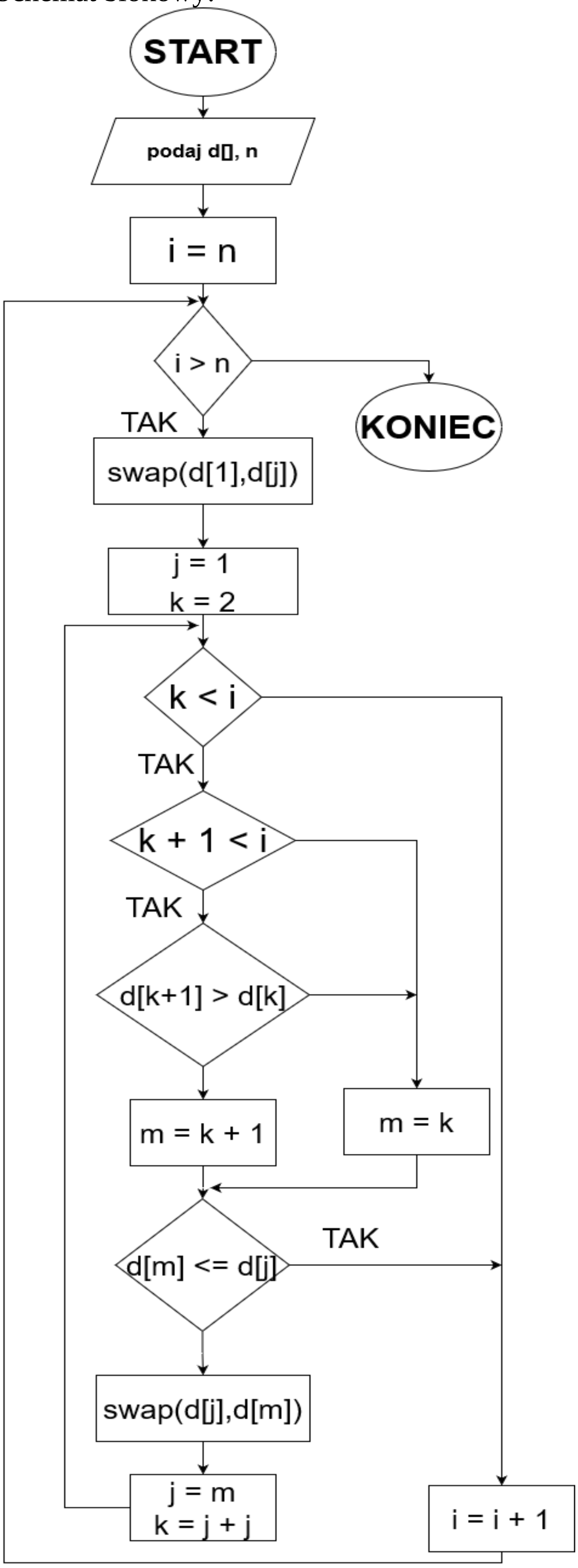
2) Rozbiór kopca

a) Kod źródłowy:

```
void deconstructHeap(int tab[], int rozmiar) {
    int j, k, m, i;
    for (i = rozmiar; i > 1; i--
) {//pętla zamienia miejscami kolejne liście ze spodu drzewa z korzeniem
        swap(tab[1], tab[i]);
        j = 1;
        k = 2;
        while (k < i) {// zadaniem tej pętli jest przywrócenie warunków kopca
            if ((k + 1 < i) && (tab[k + 1] > tab[k]))
                m = k + 1; // wyznaczamy indeks większego z dwóch węzłów potomnych
            else
                m = k;
            if (tab[m] <= tab[j]) break; // sprawdzamy czy jest zachowany warunek
kopca
            swap(tab[j], tab[m]);//zamiana miejscami węzeł nadrzędny j-
ty z jego największym potomkiem
m-tym i za nowy węzeł nadrzędny przyjmujemy węzeł m-ty
            j = m;//przechowuje indeks przodka
            k = j + j;//przechowuje indeks lewego przodka
            //pętla kończy wykonywanie się  w momencie gdy węzeł j-
ty nie posiada elementów
        }
    }
}
```

Rys. 12 Kod źródłowy rozbioru kopca

b) Schemat blokowy:



Rys. 13 Schemat blokowy rozbioru kopca

c) Pseudokod:

K01: Dla i = n, n - 1, ..., 2: wykonuj K02...K08
K02: $d[1] \leftrightarrow d[i]$
K03: $j \leftarrow 1$; $k \leftarrow 2$
K04: Dopóki ($k < i$) wykonuj K05...K08
K05: Jeżeli $(k + 1 < i) \wedge (d[k + 1] > d[k])$, to $m \leftarrow k + 1$ inaczej $m \leftarrow k$
K06: Jeżeli $d[m] \leq d[j]$, to wyjdź z pętli K04 i kontynuuj następny obieg K01
K07: $d[j] \leftrightarrow d[m]$
K08: $j \leftarrow m$; $k \leftarrow j + j$
K09: Zakończ

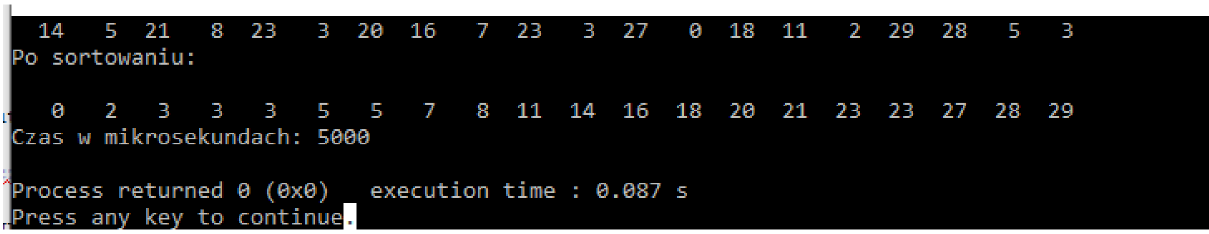
3) Kod źródłowy dla dwóch funkcji obsługujących operacje na plikach

```
//otwieranie pliku oraz zapisywanie losywch liczb
void openAndPopulateFile(string file_name, int tab[], int rozmiar) {
    fstream plik;
    int value;
    plik.open(file_name + ".txt", ios::in | ios::out);
    if (plik.good() == true)
    {
        for (int i = 1; i <= rozmiar; i++) {
            value = rand() % 31;
            plik << value << endl;
            cout << setw(4) << value;
            tab[i] = value;
        }

        plik.close();
    }
}
// zapisywanie posortowanego wynkiu do innego pliku
void saveResultsToFile(string file_name, int tab[], int rozmiar) {
    fstream plik;
    plik.open(file_name + ".txt", ios::in | ios::out);

    if (plik.good() == true) {
        cout << "Po sortowaniu:\n\n";
        for (int i = 1; i <= rozmiar; i++) {
            plik << tab[i] << endl;
            cout << setw(4) << tab[i];
        }
        plik.close();
    }
}
```

Rys. 14 Obsługa plików tekstowych

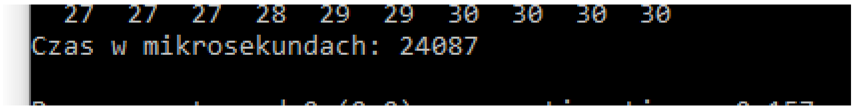
1.2.3. Testy i wykresy

1. Przypadek pesymistyczny
   - Dla 20 elementów



Rys. 15 Test dla 20 elementów pesymistycznie

   - Dla 100 elementów

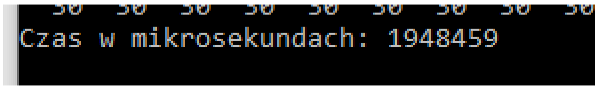


Rys. 16 Test dla 100 elementów pesymistycznie

   - Dla 1000 elementów

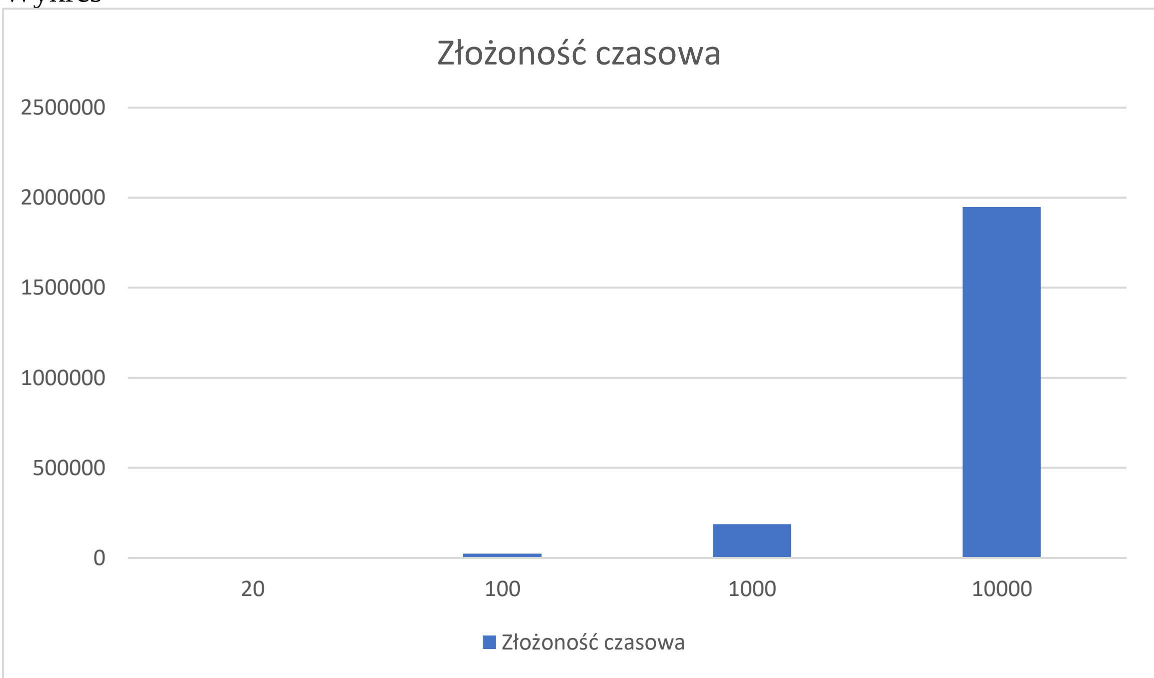
Rys. 17 Test dla 1000 elementów pesymistycznie

   - Dla 10000 elementów



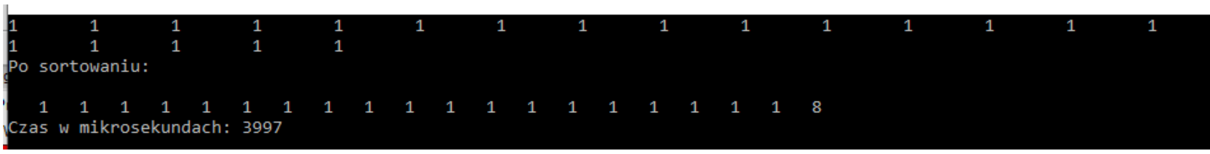
Rys. 18 Test dla 10000 elementów pesymistycznie

   - Wykres



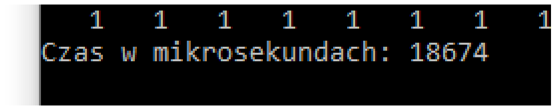
Wykres 2 Sortowanie przez kopcowanie pesymistyczne

2. Przypadek optymistyczny
   - Dla 20 elementów



Rys. 19 Test dla 20 elementów optymistycznie

- Dla 100 elementów



Rys. 20 Test dla 100 elementów optymistycznie

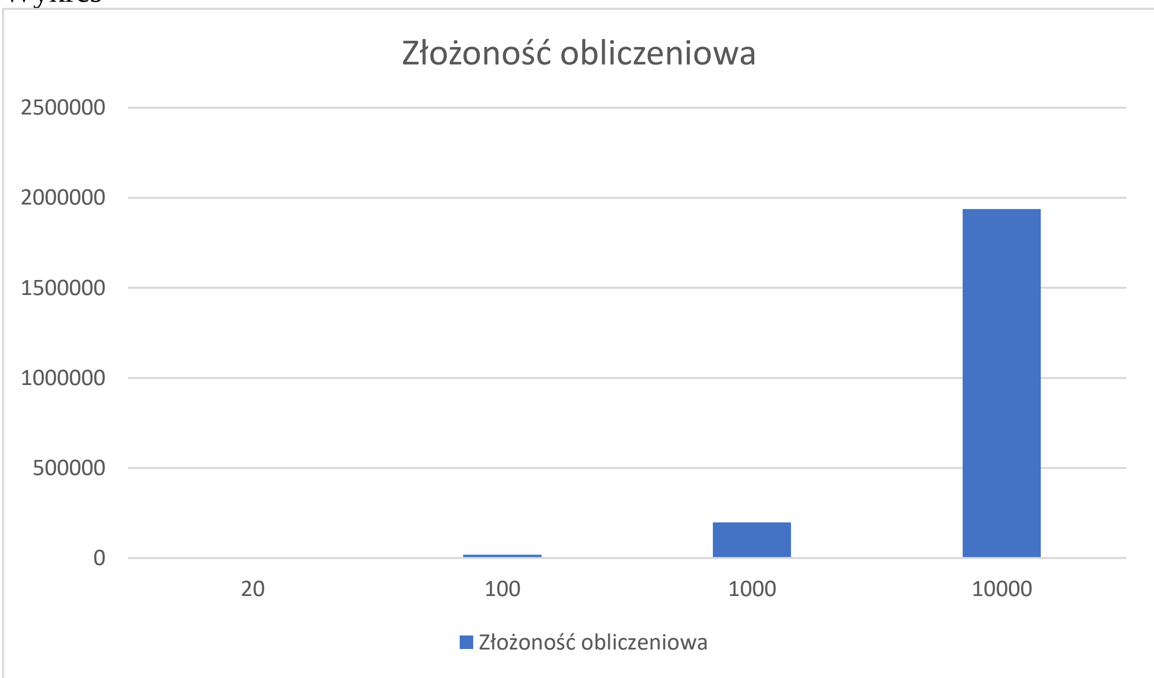
- Dla 1000 elementów

Rys. 21 Test dla 1000 elementów optymistycznie

- Dla 10000 elementów

Rys. 22 Test dla 10000 elementów optymistycznie

- Wykres



Wykres 3 Wykres sortowania przez kopcowanie optymistyczne

Link do GitHub z kodem źródłowym: https://github.com/Tyfytyfy/projektSortowanie