POLITECHNIKA RZESZOWSKA

Wydział Matematyki i Fizyki Stosowanej

Zadanie projektowe 2 - sprawozdanie

Algorytmy i struktury danych

Oliwia Konefał

Nr albumu: 173157

1. Wstęp

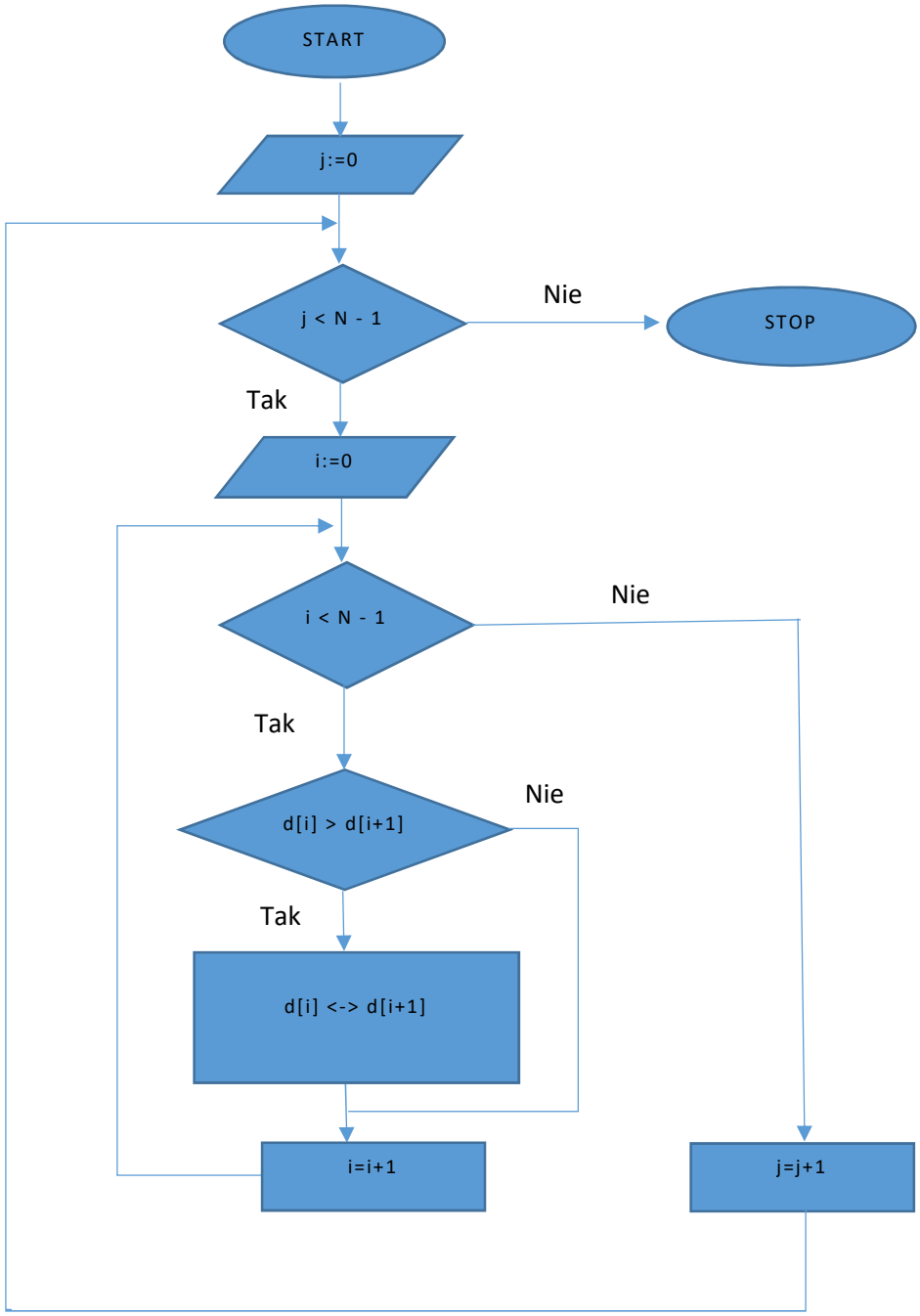
Celem zadania projektowego jest porównanie dwóch algorytmów służących do sortowania danych – **bąbelkowego** oraz **kopcowania**.

2. Sortowanie bąbelkowe – podstawy teoretyczne

Algorytm sortowania bąbelkowego jest jednym z najstarszych algorytmów sortujących. Zasada działania opiera się na cyklicznym porównywaniu par sąsiadujących elementów i zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operację tę wykonujemy dotąd, aż cały zbiór zostanie posortowany. Można go stosować tylko dla niewielkiej liczby elementów w sortowanym zbiorze. Przy większych zbiorach czas sortowania może być zbyt długi.

Algorytm opiera się na zasadzie maksimum, tj. każda liczba jest mniejsza lub równa od liczby maksymalnej. Porównując kolejno liczby, można wyznaczyć największą z nich. Następnie ciąg częściowo posortowany (mający liczbę maksymalną) można skrócić o tę liczbę i ponowić szukanie maksimum, już bez elementów odrzuconych i tak długo, aż zostanie nam jeden element.

3. Sortowanie bąbelkowe – schemat blokowy oraz pseudokod



N - liczba elementów w zbiorze

D[] - zbiór n-elementowy, który będzie sortowany

i, j - indeksy tablicy

Pseudokod:

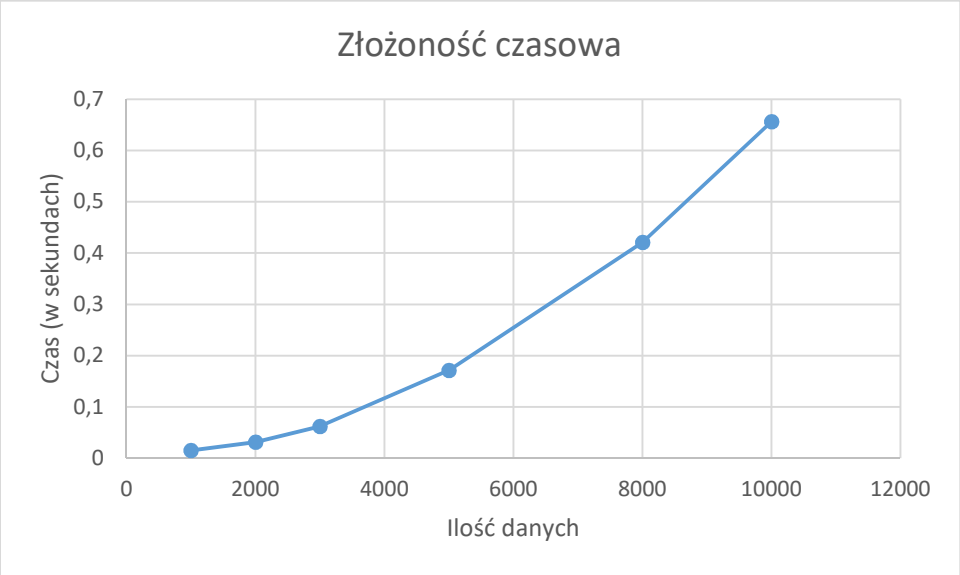
```
Wczytaj d[N], i, j;
        dla (j = 0; j < N - 1; j=j+1)
                dla(i = 0; i < N - 1; i=i+1)
                        jeśli(d[i] > d[i-1]), to
                        zamien (d[1], d[i-1]);
```

4. Sortowanie bąbelkowe - złożoność obliczeniowa

Sortowanie bąbelkowe jest jednym z najprostszych w implementacji algorytmów porządkujących dane. Złożoność tego sposobu sortowania jest rzędu **$O(n^2)$.**

Oznacza to, że sortowanie bąbelkowe nie poradzi sobie z porządkowaniem większych zbiorów.

5. Sortowanie bąbelkowe - złożoność czasowa



Liczba instrukcji algorytmu rośnie proporcjonalnie do kwadratu rozmiaru danych wejściowych.

Warto zauważyć, że ilość wymaganych porównań nie zależy od stopnia początkowego ułożenia elementów w ciągu. Nawet jeśli będziemy sortowali ciąg posortowany już na początku, to algorytm i tak będzie musiał wykonać wszystkie porównania. Średnia złożoność czasowa tego algorytmu jest zatem równa pesymistycznej.

6. Sortowanie przez kopcowanie – podstawy teoretyczne

Algorytm sortowania przez kopcowanie składa się z dwóch faz. W pierwszej sortowane elementy reorganizowane są w celu utworzenia kopca. W drugiej zaś dokonywane jest właściwe sortowanie, poprzez rozbiór kopca.

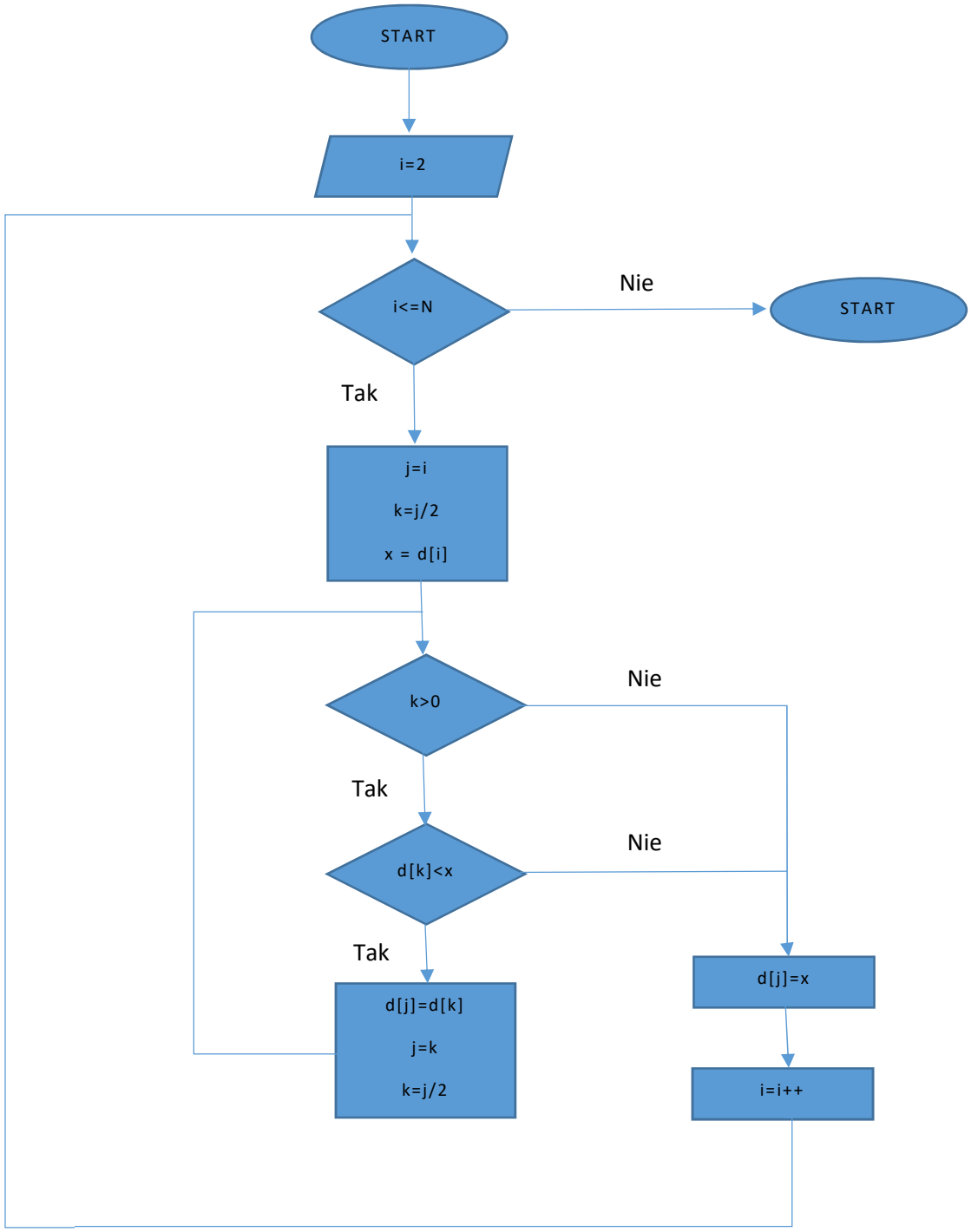
Tworzenie kopca:

Początkowo do kopca należy tylko pierwszy element w tablicy. Następnie kopiec rozszerzany jest o drugą, trzecią i kolejne pozycje tablicy, przy czym przy każdym rozszerzeniu, nowy element jest przemieszczany w górę kopca, tak aby spełnione były relacje pomiędzy węzłami. Kopiec rozrasta się, aż do wyczerpania nieposortowanej części tablicy.

Rozbiór kopca:

Po utworzeniu kopca następuje właściwe sortowanie. Polega ono na usunięciu wierzchołka kopca, zawierającego element maksymalny, a następnie wstawieniu w jego miejsce elementu z końca kopca i odtworzenie porządku kopcowego. W zwolnione w ten sposób miejsce, zaraz za końcem zmniejszonego kopca wstawia się usunięty element maksymalny. Operacje te powtarza się aż do wyczerpania elementów w kopcu.

7. Sortowanie przez kopcowanie – schemat blokowy – budowa kopca

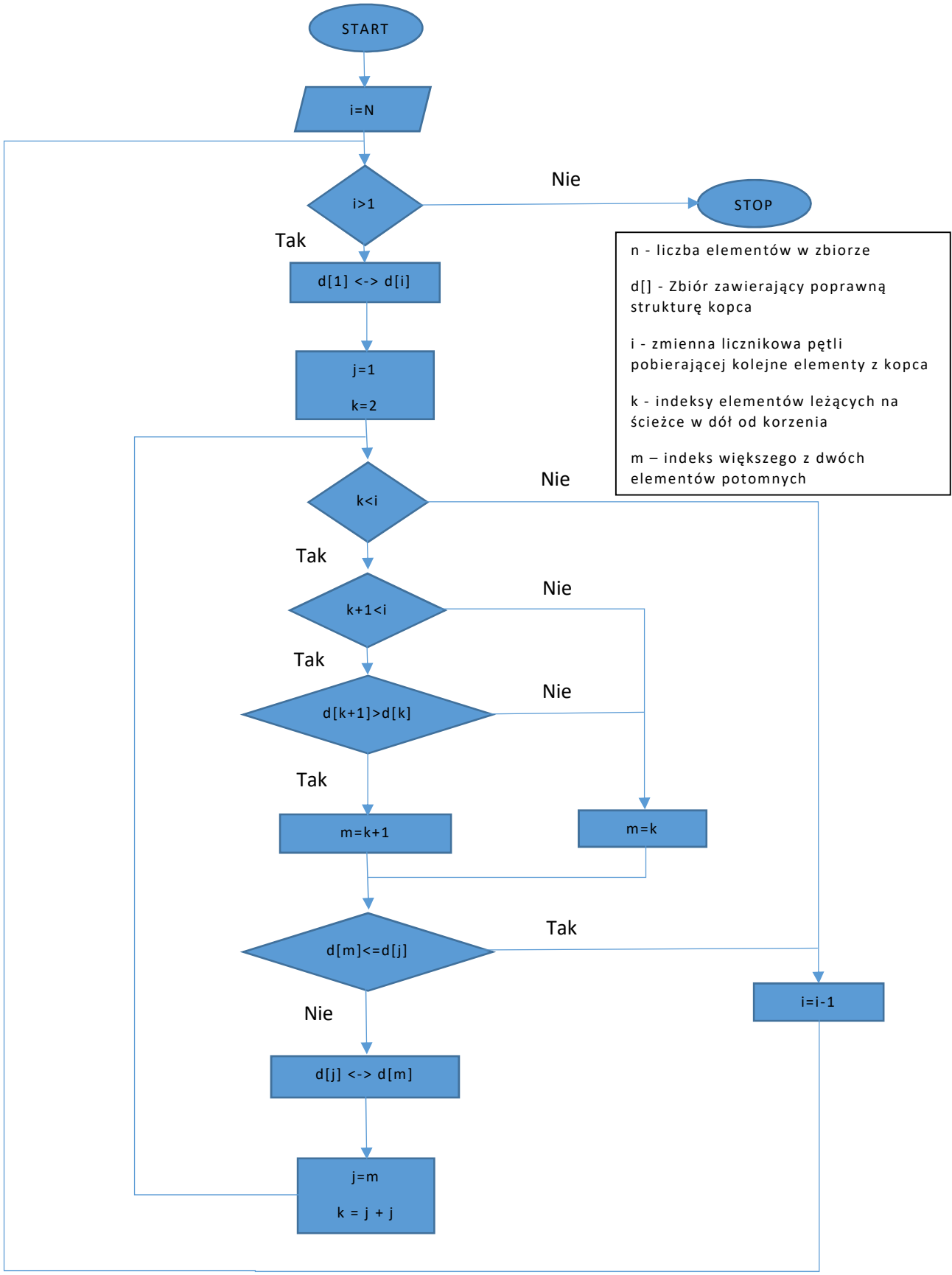


j - pozycja wstawianego elementu (liścia)

k - pozycja elementu nadrzędnego (przodka)

x - zapamiętuje wstawiany element

8. Sortowanie przez kopcowanie – schemat blokowy – rozbiór kopca



9. Sortowanie przez kopcowanie – pseudokod

```
// Budowa kopca:
dla(i = 2; i <= N; i++)
 {
   j = i; k = j / 2;
   x = d[i];
   dopoki((k > 0) oraz (d[k] < x))
   {
     d[j] = d[k];
     j = k; k = j / 2;
   }
   d[j] = x;  }
// Rozbior kopca
dla(i = N; i > 1; i--)
 {
   zamien(d[1], d[i]);
   j = 1; k = 2;
   dopoki(k < i)
   {
     jesli((k + 1 < i) oraz (d[k + 1] > d[k])), to
       m = k + 1;
     w przeciwnym wypadku
       m = k;
     jesli(d[m] <= d[j]), to
     zamien(d[j], d[m]);
     w przeciwnym wypadku
     przerwij petle;
     j = m; k = j + j;
   }
}
```

10. Sortowanie przez kopcowanie – złożoność czasowa i obliczeniowa

Sortowanie przez kopcowanie składa się z dwóch następujących bezpośrednio po sobie operacji o klasie czasowej złożoności obliczeniowej O(n log n), czyli iloczynu funkcji liniowej oraz logarytmicznej.

Dla klasy optymistycznej, typowej i pesymistycznej czasy są proporcjonalne do $n\log_2 n$, zatem wnioskujemy, iż algorytm sortowania przez kopcowanie posiada klasę czasowej złożoności obliczeniowej równą **O(n log n)**.

Najdłużej trwa sortowanie zbioru nieposortowanego. Czasy nie różnią się wiele od siebie, co sugeruje, iż algorytm jest mało czuły na postać danych wejściowych.

Ciekawostką jest to, iż czas sortowania zbiorów posortowanych jest dłuższy od sortowania zbioru posortowanego odwrotnie (jest to najkrótszy czas z otrzymanych, zatem możemy przyjąć, iż dla algorytmu sortowania przez kopcowanie przypadkiem optymistycznym jest właśnie sortowanie zbioru posortowanego odwrotnie).

11. Podsumowanie

Nazwa algorytmu	Kl. Zł. Optymistyczna	Kl. Zł. Typowa	Kl. Zł. Pesymistyczna	Stabilność	Sortowanie w miejscu	Zalecane
Sortowanie bąbelkowe	$O(n^2)$	$O(n^2)$	$O(n^2)$	TAK	TAK	TAK
Sortowanie przez kopcowanie	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	NIE	TAK	NIE

12. Kod programu:

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <time.h>
#include <windows.h>
#include <stdio.h>
#include <sys/time.h>

using namespace std;

void SortowanieBabelkowe(int arr[], int dlugoscciagu)
{

   int i,j;
   for(int j = 0; j < dlugoscciagu - 1; j++)
   for(int i = 0; i < dlugoscciagu - 1; i++)
   if(arr[i] > arr[i + 1])swap(arr[i], arr[i + 1]);

}
```

```
void SortowaniePrzezKopcowanie(int arr[], int dlugoscciagu){

int i,j,k,m,x;

// Tworzymy kopiec
 for(i = 2; i <= dlugoscciagu; i++)
 {
  j = i; k = j / 2;
  x = arr[i];
  while((k > 0) && (arr[k] < x))
  {
   arr[j] = arr[k];
   j = k; k = j / 2;
  }
  arr[j] = x;
 }

// Rozbieramy kopiec
 for(i = dlugoscciagu; i > 1; i--)
 {
  swap(arr[1], arr[i]);
  j = 1; k = 2;
  while(k < i)
  {
   if((k + 1 < i) && (arr[k + 1] > arr[k]))
    m = k + 1;
    else
    m = k;
   if(arr[m] <= arr[j]) break;
   swap(arr[j], arr[m]);
   j = m; k = j + j;
```

```
    }
  }

}
int main () {
    int iloscciagow = 3;
    int dlugoscciagu = 20;
    int ciag[dlugoscciagu];
    int ciag2[dlugoscciagu];
    unsigned long long a;
    srand (time (NULL));

    FILE *array1;
    FILE *bubble;
    FILE *heap;
    array1 = fopen("wygenerowaneciagi.txt", "a");
    bubble = fopen("sortowaniebabelkowe.txt", "a");
    heap = fopen("sortowanieprzezkopcowanie.txt", "a"); //otwarcie plikow

    for(int i = 0; i < iloscciagow; i++)
    {for(int j = 0; j < dlugoscciagu; j++)
    {
        ciag[j]=rand()%100;
        fprintf(array1, "%d ", ciag[j]);
    }
    fprintf (array1, "%s", "\n");
    }  //generowanie losowych ciagow do pliku

    for(int k=0; k < iloscciagow; k++)
    {
        for(int j = 0; j < dlugoscciagu; j++)
```

```
        {
            fscanf(array1, "%d", &ciag[j]);
             ciag2[j] = ciag[j];
        } //wczytanie ciagow w pliku

        SortowanieBabelkowe(ciag, dlugoscciagu);
        SortowaniePrzezKopcowanie(ciag2, dlugoscciagu);
        for (int i=0; i < dlugoscciagu; i++)
        {
            fprintf (bubble, "%d ", ciag[i]);
            fprintf (heap, "%d ", ciag2[i]); //zapis posortowanego ciagu do plikiu
        }
            fprintf (bubble,"%s", "\n");
            fprintf (heap, "%s", "\n");  //znak nowej lini
    }
    fclose (bubble);
    fclose (array1);
    fclose (heap);  //zamkniecie plikow

    return 0;
}
```