



POLITECHNIKA RZESZOWSKA  
im. Ignacego Łukasiewicza  
WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ

Damian Stechnij  
**Nr albumu: 173219**

PROJEKT  
ALGORYTMY I STRUKTURY DANYCH

kierunek studiów: inżynieria i analiza danych

Rzeszów, 2022

## Spis treści

1	TEMAT PRACY .....	3
2	PROJEKTOWANIE.....	4
2.1	Problem zagadnienia.....	4
2.2	Teoretyczne podstawy .....	4
2.2.1	Algorytm sortowania grzebieniowego .....	4
2.2.2	Algorytm sortowania kopcowego .....	4
3	SCHEMATY BLOKOWE I PSEUDOKODY .....	5
3.1	Pseudokod.....	5
3.1.1	Algorytm sortowania grzebieniowego .....	5
3.1.2	Algorytm sortowania kopcowego .....	5
3.2	Schemat blokowy .....	6
3.2.1	Algorytm sortowania grzebieniowego .....	6
3.2.2	Algorytm sortowania kopcowego .....	7
4	KOD ŹRÓDŁOWY ALGORYTMÓW SORTUJĄCYCH .....	9
4.1	Sortowanie grzebieniowe .....	9
4.1.1	Funkcja wykonująca znajdującą rozpiętość .....	9
4.1.2	Funkcja wykonująca sortowanie .....	9
4.2	Sortowanie kopcowe.....	10
4.2.1	Funkcja znajdującą większy element .....	10
4.2.2	Funkcja wykonująca sortowanie kopcowe.....	10
5	DZIAŁANIE PROGRAMU.....	11
5.1	Przykład.....	11
5.1.1	Dane wejściowe w pliku tekstowym.....	11
5.1.2	Wyniki wypisane w konsoli .....	11
5.1.3	Wyniki wypisane w plikach tekstowych.....	11
6	ZŁOŻONOŚĆ OBLICZENIOWA .....	12
6.1	Sortowanie kopcowe.....	12
6.2	Sortowanie grzebieniowe .....	13
7	WNIOSKI.....	14

# 1    TEMAT PRACY

Porównanie algorytmu sortowania kopcowego i algorytmu sortowania grzebieniowego.

Cechy jakie powinien zawierać program:

- program powinien odczytywać dane wejściowe z pliku tekstowego i zapisywać posortowany już ciąg do pliku tekstowego,
- w celu wykonania testów należy zaimplementować funkcję, która generuje pseudolosowy ciąg elementów o zadanej długości,
- założyć, że sortowanymi elementami są liczby całkowite z przedziału  $[0, N]$ , gdzie  $N$  powinno być „odpowiednio dużym” parametrem ustalonym wewnątrz programu,
- kod powinien zawierać komentarze ułatwiające zrozumienie programu.

## 2 PROJEKTOWANIE

### 2.1 Problem zagadnienia

W zadaniu trzeba porównać dwie metody sortowania: sortowanie przez kopcowanie oraz sortowanie grzebieniowe.

Aby wykonać testy trzeba utworzyć wygenerowaną tablicę z pseudolosowymi liczbami. Program docelowo ma wczytywać plik tekstowy i wpisywać dane do tablicy.

Później sortujemy liczby obiema metodami, porównujemy czasy sortowań oraz zapisujemy wszystko do pliku.

### 2.2 Teoretyczne podstawy

#### 2.2.1 Algorytm sortowania grzebieniowego

Przyjmuje długość tablicy za rozpiętość, którą dzielimy przez 1.3 i odrzucamy część ułamkową. Bada kolejno wszystkie pary obiektów odległych o wyliczoną wcześniej rozpiętość, gdy są ułożone niemonotonicznie zamieniamy je miejscami. Wykonuje się to w pętli, do momentu, gdy rozpiętość osiągnie wartość 1. Gdy rozpiętość osiągnęła wartość 1 zachowuje się tak jak sortowanie bąbelkowe.

#### 2.2.2 Algorytm sortowania kopcowego

Podstawą algorytmu jest wykorzystanie struktury danych typu kopiec binarny. Złożony jest z dwóch faz, w których w pierwszym elementy reorganizowane są w celu utworzenia kopca, a w drugiej wykonywane jest sortowanie właściwe.

Do utworzenia kopca można wykorzystywać początkową tablicę, w której znajdują się nieposortowane elementy. Na początku do kopca należy tylko pierwszy element tablicy, potem rozszerzamy go o kolejne elementy, sprawdzając czy przy każdym nowym wprowadzonym elemencie jest spełniony warunek kopca. Jeśli takowy nie jest to przemieszczamy elementy w górę kopca, by warunek został spełniony.

## 3 SCHEMATY BLOKOWE I PSEUDOKODY

### 3.1 Pseudokod

#### 3.1.1 Algorytm sortowania grzebieniowego

```
Combsort

wejscie(tab[], rozmiar)
gap <- rozmiar
zamiana <- true
dopóki gap != 1 || zamiana == true
    gap <- gap * 10 div 13
    jeśli gap < 1 wykonaj
        gap <- 1
    zamiana <- false
    i <- 0
    dopóki i + gap < rozmiar wykonuj
        jeśli tab[i + gap] < tab[i] wykonaj
            zamień (tab[i], tab[i + gap])
            zamiana <- true
        i <- i+1
Zakończ
```

#### 3.1.2 Algorytm sortowania kopcowego

```
Heap Sort

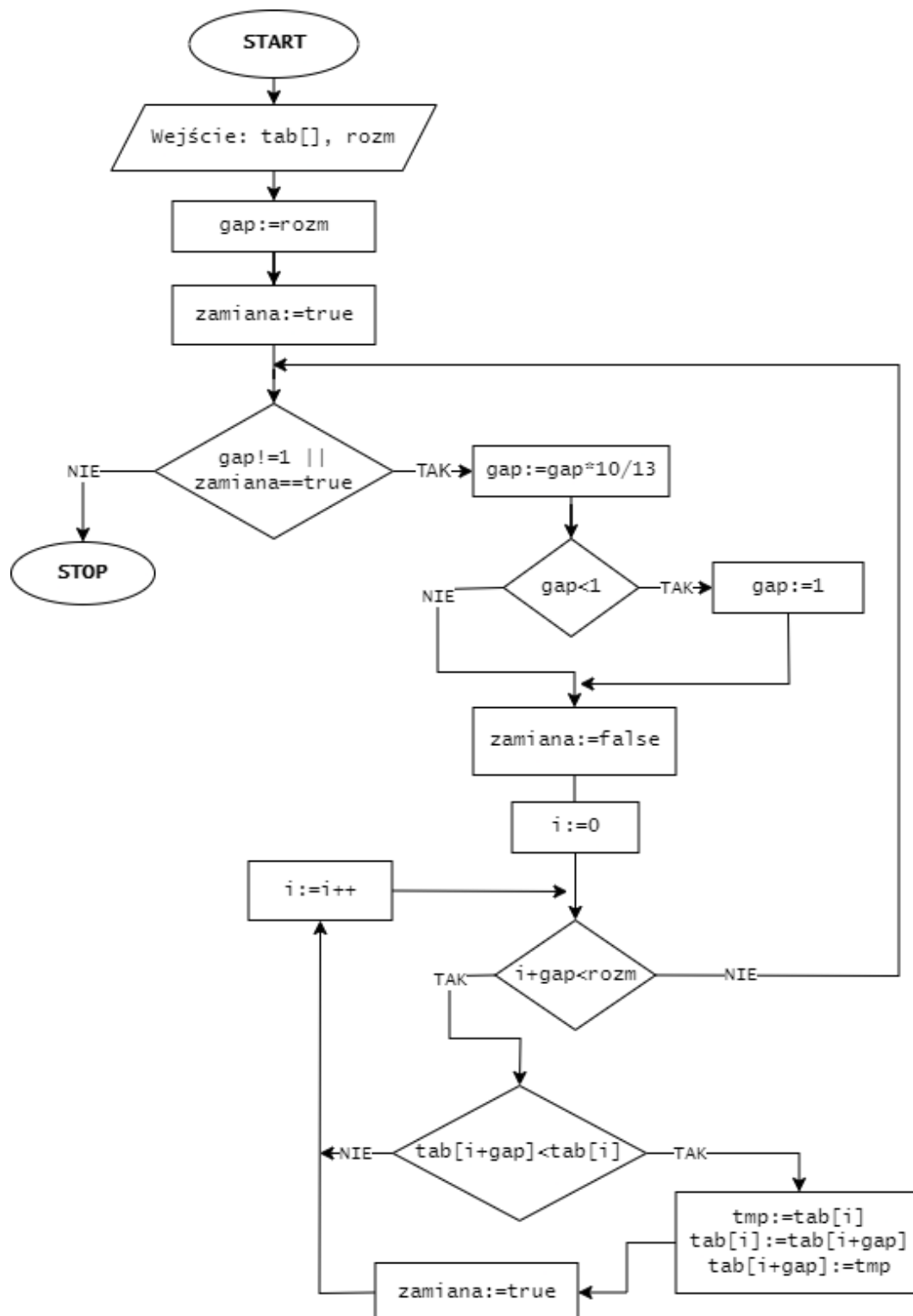
wejscie(tab[], rozmiar)
i <- rozmiar/2-1
dopóki i >= 0 wykonuj
    kopcowanie(tab, rozmiar, i)
    i <- i-1
i <- rozmiar-1
dopóki i > 0 wykonuj
    zamień(tab[0], tab[i])
    kopcowanie(tab, i, 0)
    i <- i-1
Zakończ

Kopcowanie

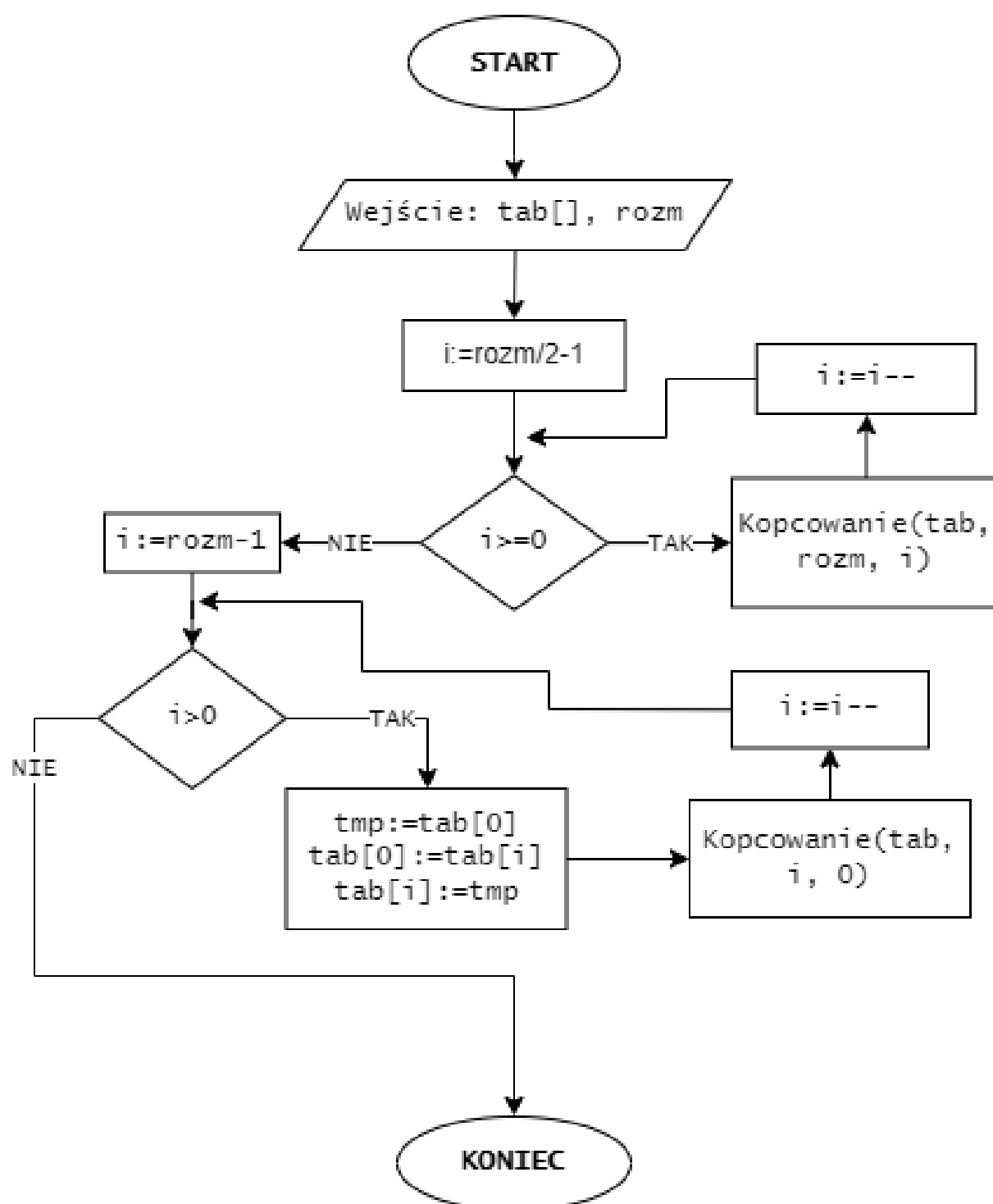
wejscie(tab[], rozmiar, i)
najwiekszy <- i
lewy <- 2 * i + 1
prawy <- 2 * i + 2
jeśli lewy < rozmiar && tab[lewy] > tab[najwiekszy] wykonaj
    najwiekszy <- lewy
jeśli prawy < rozmiar && tab[prawy] > tab[najwiekszy] wykonaj
    najwiekszy <- prawy
jesli najwiekszy != i wykonaj
    zamień (tab[i], tab[najwiekszy])
    kopcowanie(tab, rozm, naj)
Zakończ
```

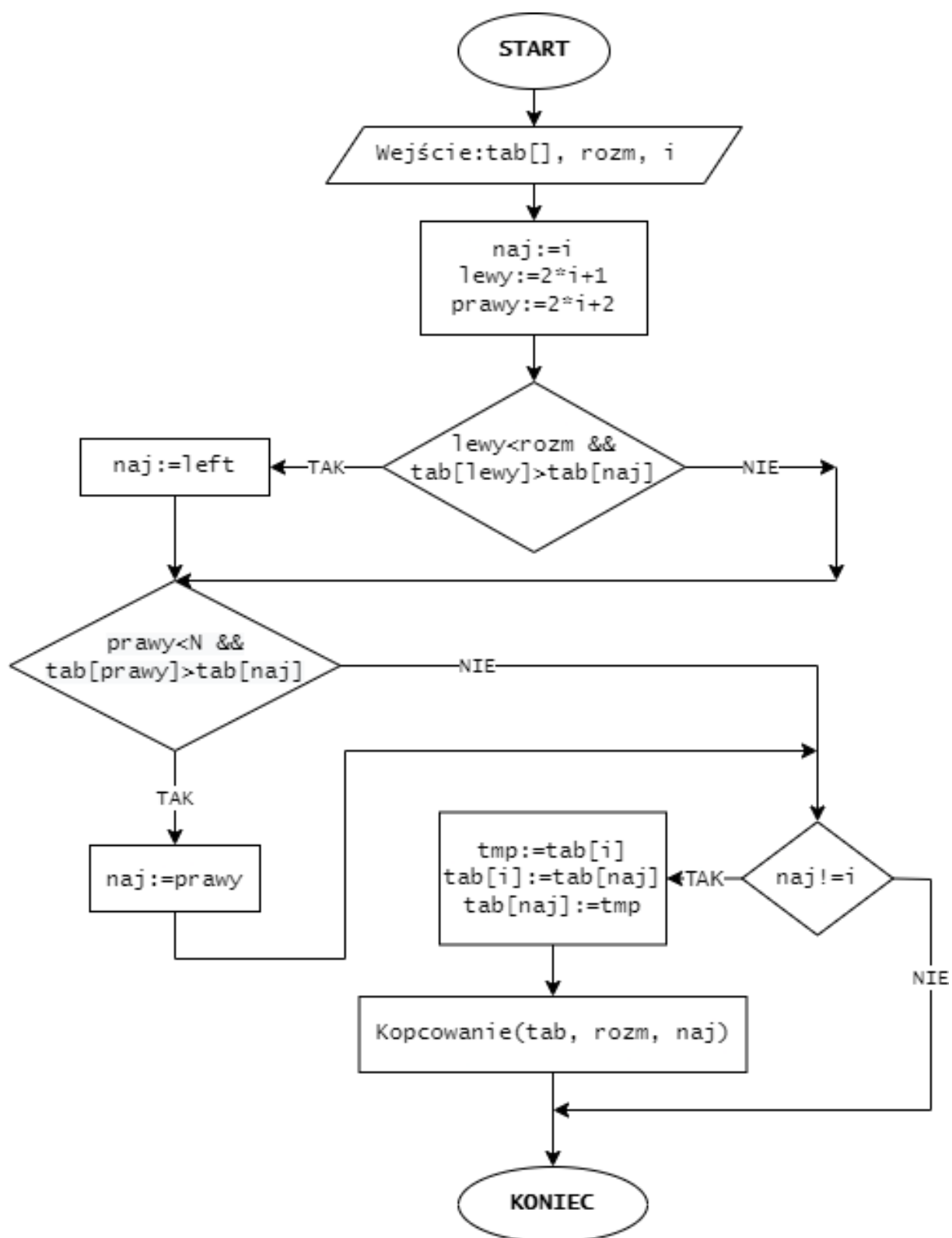
## 3.2 Schemat blokowy

### 3.2.1 Algorytm sortowania grzebieniowego



### 3.2.2 Algorytm sortowania kopcowego







## 4 KOD ŹRÓDŁOWY ALGORYTMÓW SORTUJĄCYCH

### 4.1 Sortowanie grzebieniowe

#### 4.1.1 Funkcja wykonująca znajdującą rozpiętość

```
25 // Funkcja znajdujaca odstep miedzy elementami
26 int getNextGap(int gap) {
27     // Zmniejszanie sie odstepu przez wspolczynnik
28     gap=(gap*10)/13;
29     if (gap < 1) // jesli gap<0 zwraca 1
30         return 1;
31     return gap;
32 }
```

#### 4.1.2 Funkcja wykonująca sortowanie

```
34 // Sortowanie grzebieniowe
35 void combSort(int* tab, int size) {
36     int gap=size, tmp;
37     bool swapped=true; //zainicjowanie zmiennej prawda/falsz
38     while (gap!=1 || swapped) { // jesli gap=1 lub nie dokonano zamiany - wyjscie z petli
39         gap=getNextGap(gap);
40         swapped=false;
41         // wykonuje od 0 do ostatniego elementu tablicy
42         for (int i=0; i+gap<size; i++) {
43             if (tab[i + gap] < tab[i]) { // porownanie elementow odleglych o rozpietosc
44                 swap(tab[i], tab[i+gap]); // zamiana elementow
45                 swapped = true;
46             }
47         }
48     }
49 }
```

## 4.2 Sortowanie kopcowe

### 4.2.1 Funkcja znajdujący większy element

```
26 // Ułożenie poddrzewa zakorzenionego w węzle i
27 // który jest indeksem w arr[].
28 // N jest rozmiarem tablicy
29 void heapify(int arr[], int size, int i) {
30     int largest=i; // Zainicjowanie largest jako korzenia
31     int left=2*i+1; // lewy potomek = 2*i + 1
32     int right=2*i+2; // prawy potomek = 2*i + 2
33     // Jeśli lewy potomek jest większy niż korzeń
34     if (left<size && arr[left]>arr[largest])
35         largest=left;
36     // Jeśli prawy potomek jest większy niż korzeń
37     if (right<size && arr[right]>arr[largest])
38         largest=right;
39     // Jeśli największy nie jest korzeniem
40     if (largest!=i)
41     {
42         swap(arr[i], arr[largest]);
43         // Rekursywne kopcowanie
44         heapify(arr, size, largest);
45     }
46 }
```

### 4.2.2 Funkcja wykonująca sortowanie kopcowe

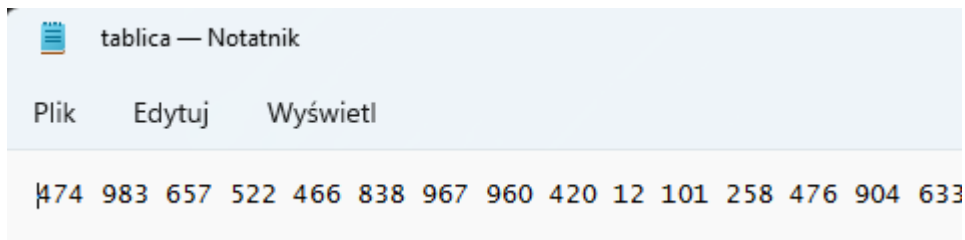
```
48 // Główna funkcja wykonująca sortowanie kopcowe
49 void heapSort(int arr[], int size) {
50     // Budowanie kopca
51     for (int i=size/2-1; i>=0; i--)
52         heapify(arr, size, i);
53     // Wyodrębnianie elementu jeden po drugim z kopca
54     for (int i=size-1; i>0; i--)
55     {
56         // Przeniesienie korzenia na koniec
57         swap(arr[0], arr[i]);
58         // Wywołanie max heapify na zredukowanym kopcu
59         heapify(arr, i, 0);
60     }
61 }
```

## 5 DZIAŁANIE PROGRAMU

Program wczytuje plik, w którym znajduje się tablica (tablica.txt), oraz zapisuje posortowaną już tablicę do pliku wyniki1.txt dla kopcowego i wyniki2.txt dla grzebieniowego. W konsoli podajemy rozmiar tablicy jaki chcemy wczytać, a po wykonaniu wyświetla się czas sortowania tablic dla każdej z metod.

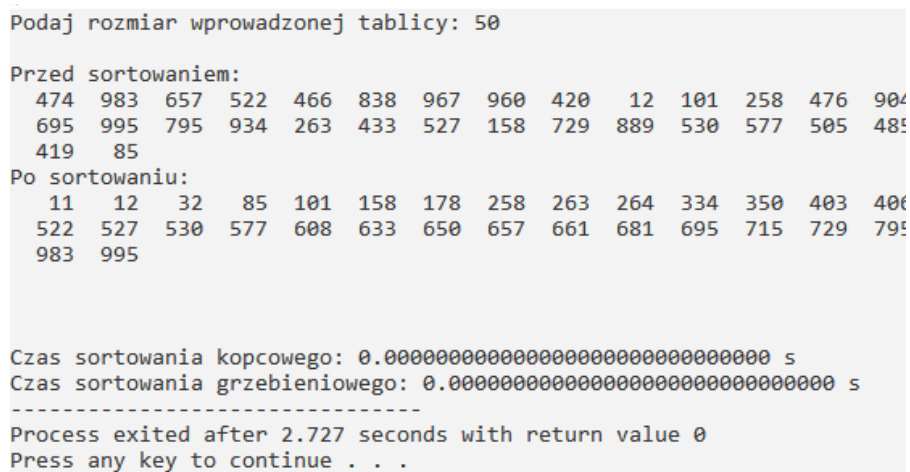
### 5.1 Przykład

#### 5.1.1 Dane wejściowe w pliku tekstowym



```
tablica — Notatnik
Plik  Edytuj  Wyświetl
474 983 657 522 466 838 967 960 420 12 101 258 476 904 633
```

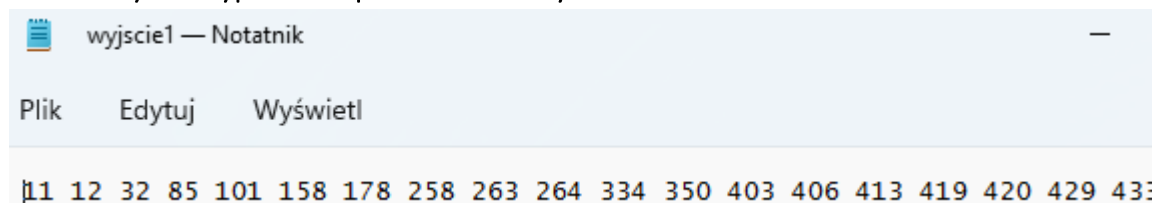
#### 5.1.2 Wyniki wypisane w konsoli



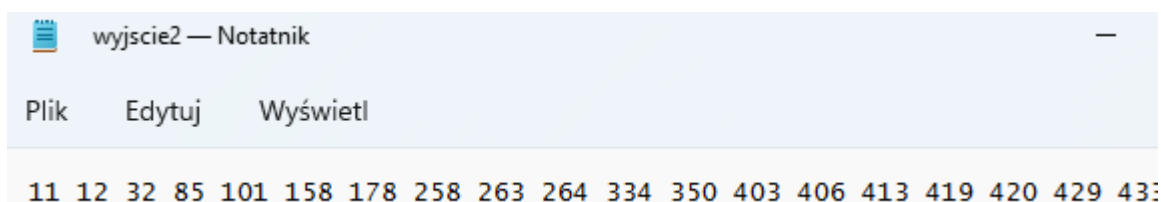
```
Podaj rozmiar wprowadzonej tablicy: 50
Przed sortowaniem:
474 983 657 522 466 838 967 960 420 12 101 258 476 904 633
695 995 795 934 263 433 527 158 729 889 530 577 505 485
419 85
Po sortowaniu:
11 12 32 85 101 158 178 258 263 264 334 350 403 406 413
522 527 530 577 608 633 650 657 661 681 695 715 729 795
983 995

Czas sortowania kopcowego: 0.000000000000000000000000000000 s
Czas sortowania grzebieniowego: 0.000000000000000000000000000000 s
-----
Process exited after 2.727 seconds with return value 0
Press any key to continue . . .
```

#### 5.1.3 Wyniki wypisane w plikach tekstowych



```
wyjscie1 — Notatnik
Plik  Edytuj  Wyświetl
11 12 32 85 101 158 178 258 263 264 334 350 403 406 413
```

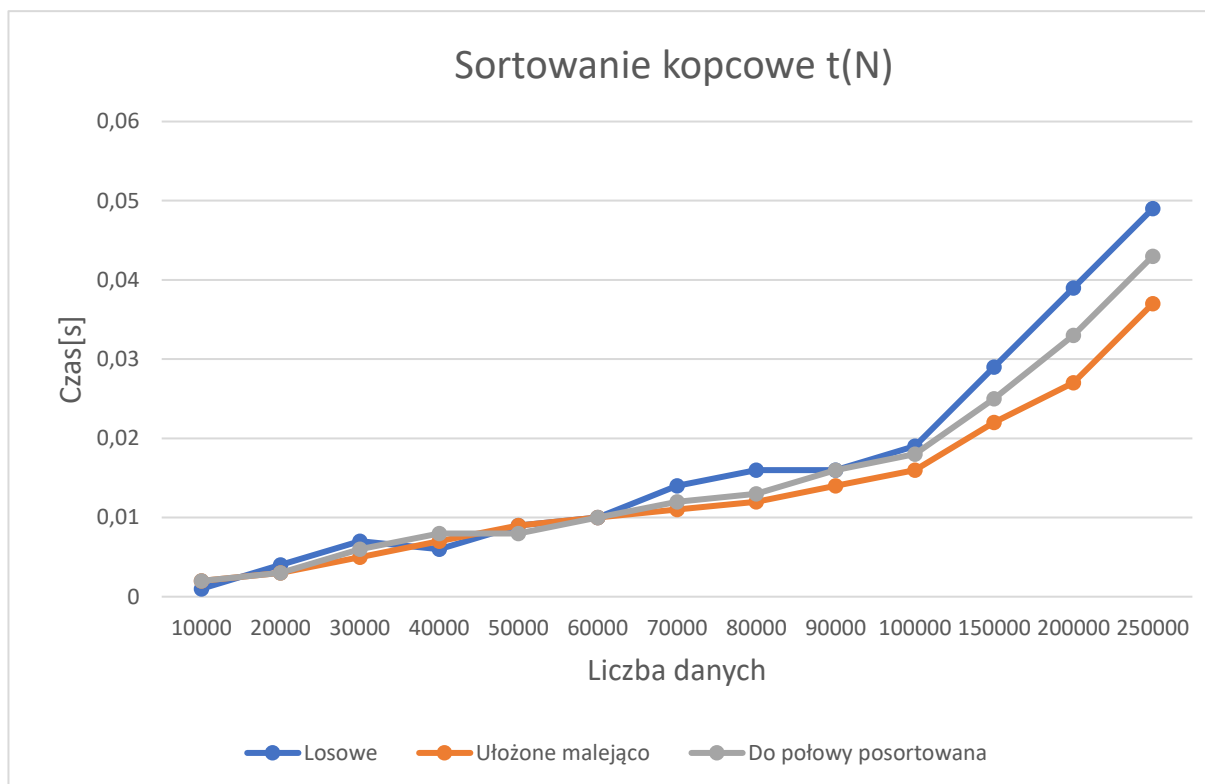


```
wyjscie2 — Notatnik
Plik  Edytuj  Wyświetl
11 12 32 85 101 158 178 258 263 264 334 350 403 406 413
```

## 6 ZŁOŻONOŚĆ OBLICZENIOWA

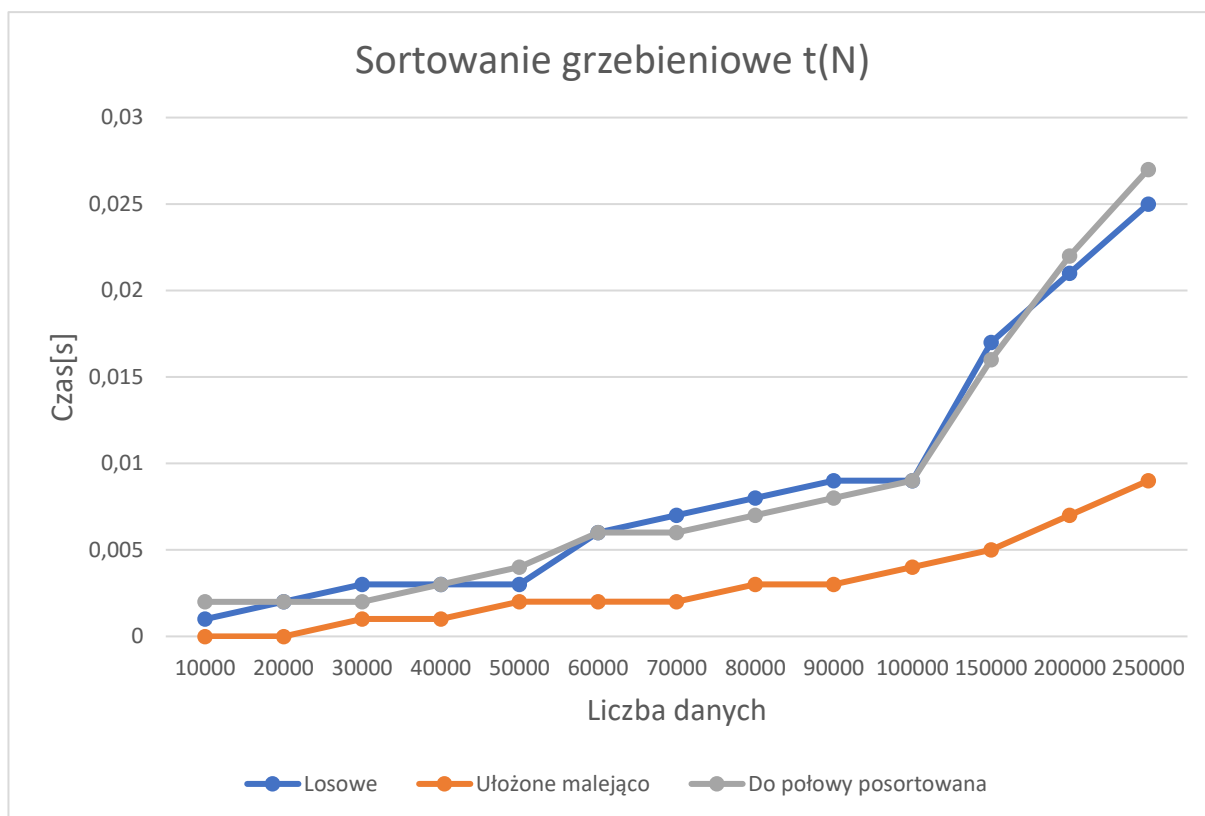
### 6.1 Sortowanie kopcowe

Algorytm sortowania przez kopcowanie jest niestabilnym, ale mimo to szybkim i niepochłaniającym wiele pamięci. Jego złożoność czasowa wynosi  $O(n \log n)$ , a pamięciowa  $O(n)$ , złożoność pamięciowa dodatkowych struktur wynosi  $O(1)$ , czyli wykonuje się "w miejscu". Jest odporny na tak spreparowane dane powodujące wolniejsze działanie, co widać na wykresie. Jego działanie na różnych próbkach danych jest mniej więcej podobne czasowo.



## 6.2 Sortowanie grzebieniowe

Algorytm sortowania grzebieniowego oparty jest na sortowaniu bąbelkowym. Jego złożoność czasowa prawdopodobnie wynosi  $O(n \log n)$ . Współczynnik 1.3 wyznaczono doświadczalnie. Działa o wiele szybciej dla tablicy, w której to elementy są ułożone malejąco niż dla pozostałych próbek danych.



## 7 WNIOSKI

Algorytm grzebieniowy szybszy niż algorytm sortowania przez kopcowanie.

Dla tablicy, w której elementy są ułożone malejąco sortowanie grzebieniowe jest o wiele szybsze niż dla tablicy, w której elementy są pseudolosowe lub do połowy posortowane.

W sortowaniu przez kopcowanie nie ma większych różnic czasowych w różnie zadanych próbkach danych.

Program może pobierać dane wejściowe z pliku jak i wypisywać dane wyjściowe do pliku.

Czas jest mierzony dla obu sposobów sortowania.

Kod zawiera komentarze, które pomogą zrozumieć działanie programu.

Schemat blokowy i pseudokod zostały sporządzone do obu algorytmów.

Wersje programu po zmianach były umieszczane na serwer GitHub, do którego link został wysłany mailowo.