# SPRAWOZDANIE

# PROJEKT Z PRZEDMIOTU „ALGORYTMY I STRUKTURY DANYCH”

# TEMAT

„Zaimplementuj sortowanie przez scalanie oraz sortowanie kopcowe.”

Bartłomiej Trzepacz, PRz, semestr 2020/2021,
Inżynieria i Analiza Danych

# Spis treści

# 1. Podstawy teoretyczne algorytmów sortowania

## 1.1 Sortowanie przez kopcowanie

Sortowanie przez kopcowanie– jeden z algorytmów sortowania, choć niestabilny, to jednak szybki i niepochłaniający wiele pamięci (złożoność czasowa wynosi O(nlogn) a pamięciowa O(n). Jest on w praktyce z reguły nieco wolniejszy od sortowania szybkiego, lecz ma lepszą pesymistyczną złożoność czasową (przez co jest odporny np. na atak za pomocą celowo spreparowanych danych, które spowodowałyby jego znacznie wolniejsze działanie.

### 1.1.1 Opis algorytmu

Podstawą algorytmu jest użycie kolejki priorytetowej zaimplementowanej w postaci binarnego kopca zupełnego. Zasadniczą zaletą kopców jest stały czas dostępu do elementu maksymalnego (lub minimalnego) oraz logarytmiczny czas wstawiania i usuwania elementów; ponadto łatwo można go implementować w postaci tablicy.

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.

### 1.1.2 Tworzenie kopca

Podstawową zaletą algorytmu jest to, że do stworzenia kopca wykorzystać można tę samą tablicę, w której początkowo znajdują się nieposortowane elementy. Dzięki temu uzyskuje się stałą złożoność pamięciową. 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.

```
57      for(i=2; i<=n; i++) //budujemy kopiec
58      {
59       j = i;
60       k = j/2;
61       x = tab[i];
62       while((k>0) && (tab[k]<x))
63       {
64         tab[j] = tab[k];
65         j=k;
66         k=j/2;
67       }
68       tab[j] = x;
69      }
```

*Rys. 1 Algorytm budowy kopca*

### 1.1.3 Sortowanie

Po utworzeniu kopca następuje właściwe sortowanie. Polega ono na usunięciu wierzchołka kopca, zawierającego element maksymalny (minimalny), 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.

```
71  // Rozbieramy kopiec
72
73    for(i=n; i>1; i--)
74    {
75      swap(tab[1], tab[i]);
76      j=1;
77      k=2;
78      while(k<i)
79      {
80        if((k+1<i) && (tab[k+1]>tab[k]))
81          m=k+1;
82        else
83          m=k;
84        if(tab[m] <= tab[j])
85          break;
86        swap(tab[j], tab[m]);
87        j=m;
88        k=j+j;
89      }
90    }
```

*Rys. 2 Algorytm rozbioru kopca*

## 1.2 Sortowanie przez scalanie

Algorytm sortowania przez scalanie opiera się na zasadzie „dziel i zwyciężaj”. Główna zasada działania polega na rekurencyjnym dzieleniu tablicy na podtablice. Dzielenie kończymy, w którym, każda z podtablic w danej grupie jest tablicą jednoelementową. Łączymy je kolejno porównując wartości ich elementów.

### 1.2.1 Opis algorytmu

Jeśli indeks prawej części tablicy (r) jest większy od indeksu lewej części tablicy wykonaj następujące kroki:

1. Znajdź środkowy indeks tablicy (m), ze wzoru m=(l+r)/2 (resztę z dzielenia zaokrąglamy w dół)
2. Wywołaj sortowanie przez scalanie dla podtablicy o indeksach od l do m
3. Wywołaj sortowanie przez scalanie dla podtablicy o indeksach od m+1 do r
4. Połącz podtablicę l – m i m+1 – r, przyrównując przy tym ich elementy – Jeśli element lewej tablicy na aktualnej pozycji jest mniejszy od tego na prawej, wstaw go do tablicy docelowej i zwiększ indeks lewej tablicy. W przeciwnym przypadku, wykonaj analogiczne działanie dla prawej tablicy.

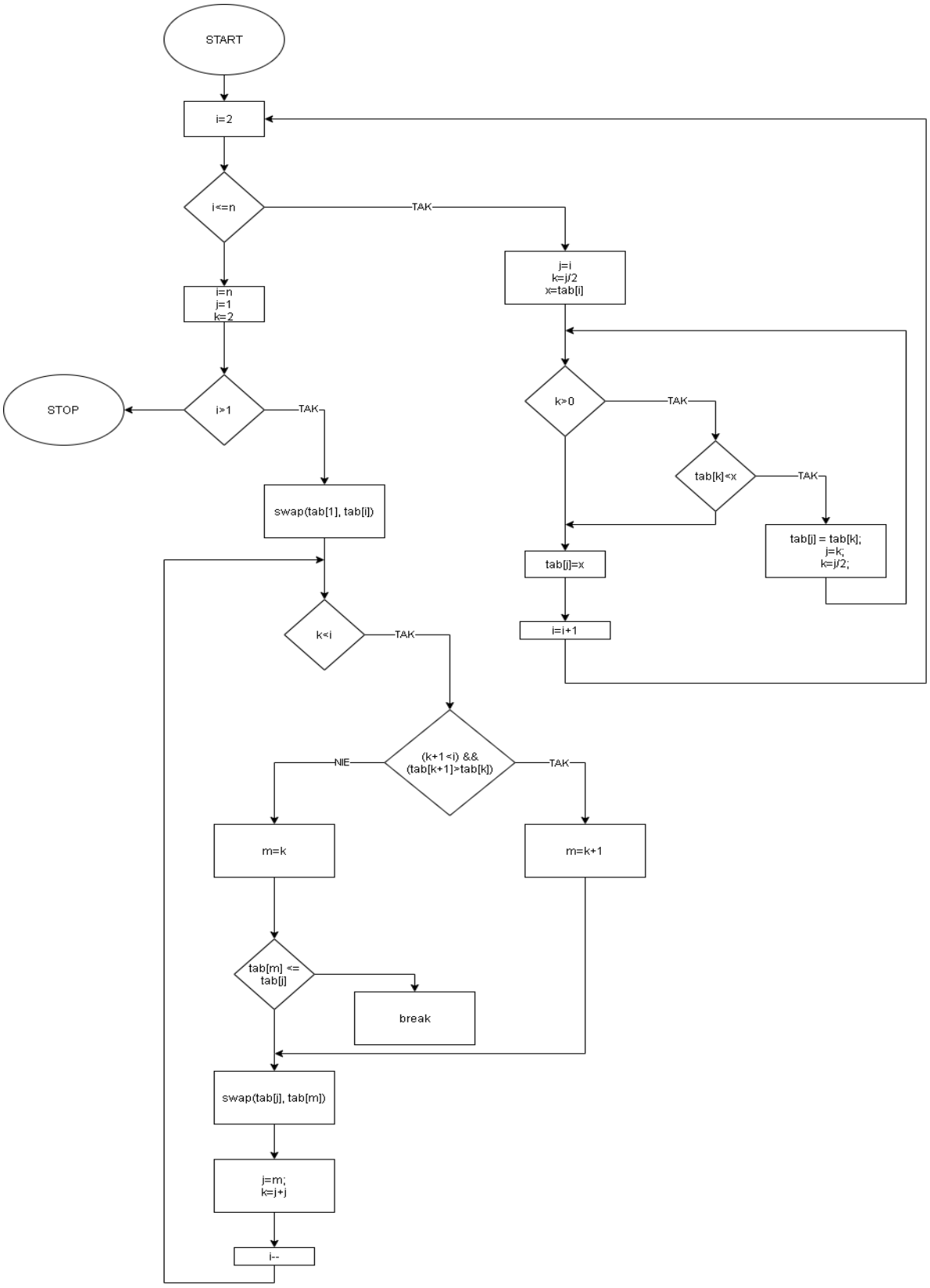
```
151  void sortowanie_przez_scalanie(int* tab, int l, int r)
152  {
153      if (r > l) {
154          int m = (l + r) / 2;
155          sortowanie_przez_scalanie(tab, l, m);
156          sortowanie_przez_scalanie(tab, m + 1, r);
157          merge(tab, l, m, r);
158      }
159  }
```

*Rys. 3 Algorytm sortowania_przez_scalanie*

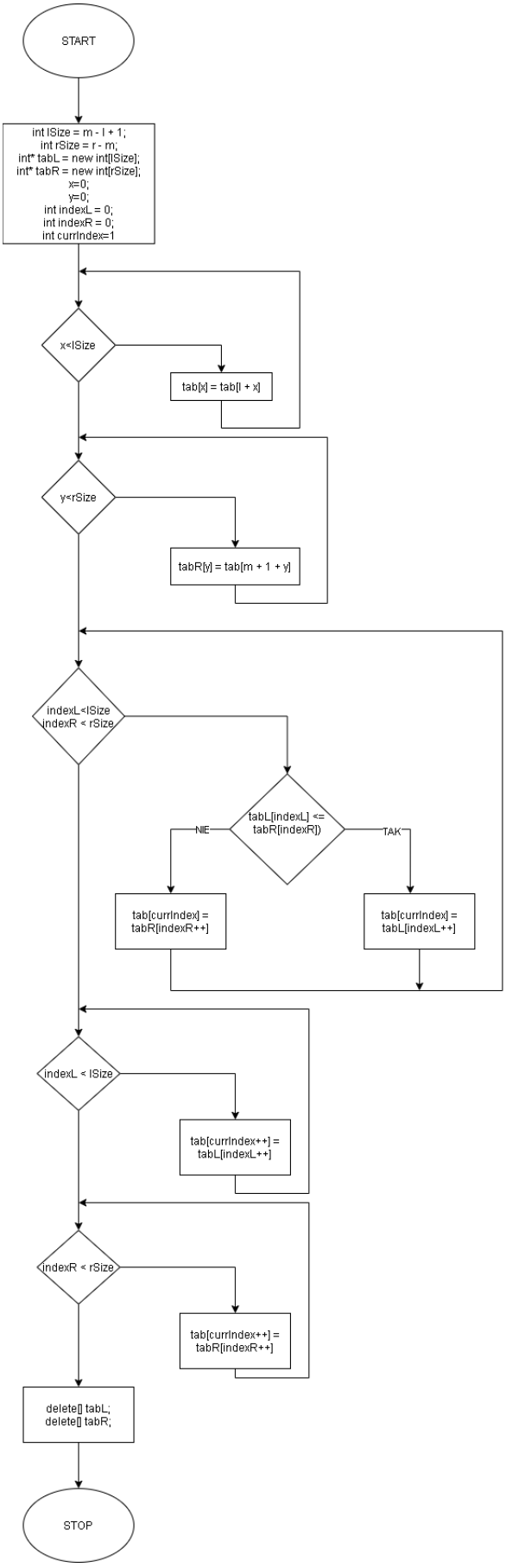
```
108  void merge(int* tab, int l, int m, int r)
109  {
110      int lSize = m - l + 1;
111      int rSize = r - m;
112
113      //Tablice pomocnicze
114      int* tabL = new int[lSize];
115      int* tabR = new int[rSize];
116
117      // Kopiowanie danych do tablic pomocniczych
118      for (int x = 0; x < lSize; x++)
119          tabL[x] = tab[l + x];
120      for (int y = 0; y < rSize; y++)
121          tabR[y] = tab[m + 1 + y];
122
123      int indexL = 0;
124      int indexR = 0;
125      int currIndex;
126
127      //Laczenie tablic R i L
128      for (currIndex = l; indexL < lSize && indexR < rSize; currIndex++)
129      {
130          if (tabL[indexL] <= tabR[indexR])
131              tab[currIndex] = tabL[indexL++];
132          else
133              tab[currIndex] = tabR[indexR++];
134      }
135
136      //Jeśli w tablicy tabL pozostały jeszcze jakieś elementy
137      //kopiujemy je
138      while (indexL < lSize)
139          tab[currIndex++] = tabL[indexL++];
140
141      //Jeśli w tablicy tabR pozostały jeszcze jakieś elementy
142      //kopiujemy je
143      while (indexR < rSize)
144          tab[currIndex++] = tabR[indexR++];
145
146      delete[] tabL;
147      delete[] tabR;
```

*Rys. 4 Algorytm scalania*

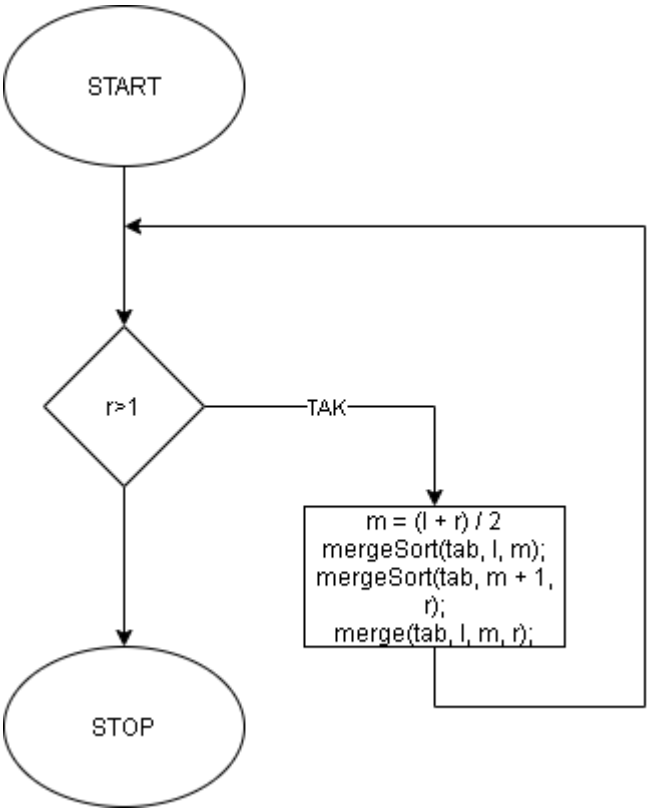
# 2.Schematy blokowe




*Schemat blokowy 1 Sortowanie przez kopcowanie*




*Schemat blokowy 2 Merge*




*Schemat blokowy 3 Sortowanie_przez_scalanie*

# 3. Pseudokod

## 3.1 Pseudokod algorytmu sortowania przez kopcowanie

### 3.1.1 Budowa kopca

K01: Dla i = 2, ..., n:
wykonuj kroki 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

### 3.1.2 Rozbiór kopca

K01: Dla i = n, n - 1, ..., 2:
wykonuj kroki K02...K08

K02: d [1] ↔ d[i]

K03: j ← 1; k ← 2

K04: Dopóki (k < i):
wykonuj kroki K05...K08

K05: Jeżeli (k + 1 < i) ∧ (d k + 1] > d[k]),
to m ← k + 1
inaczej m ← k

K06: Jeżeli d[m] ≤ d[j],
to wyjdź z pętli K04
i kontynuuj następny obieg K01

K07: d[j] ↔ d[m]

K08: j ← m; k ← j + j

K09: Zakończ

## 3.2 Pseudokod algorytmu sortowania przez scalanie

### 3.2.1 Algorytm scalający

K01: lSize ← m - l + 1;

rSize ← r - m;

K02: Dla x<lSize: wykonuj K03:

K03: tabL[x] = tab [l + x]

K04: Dla y<rSize: wykonuj K05:

K05: tabR[y] = tab[m + 1 + y]

K06: int indexL ← 0; int indexR ← 0; int currIndex ← l

K07: Dla indexL < lSize i indexR < rSize wykonuj od K08 do K10:

K08: Jeśli tabL[indexL] <= tabR[indexR] wykonuj K09, w przeciwnym wypadku przejdź do K10

K09: tab[currIndex] ← tabL[indexL++]

K10: tab[currIndex] ← tabR[indexR++]

K11: Dopóki indexR < rSize wykonuj K12:

K12: tab[currIndex++] = tabL[indexL++]

K13: Dopóki indexR < rSize wykonuj K14:

K14: tab[currIndex++] = tabR[indexR++]

K15: Usuń tablice dynamiczne i zakończ

### 3.2.2 Algorytm sortujący

K01: m ← (l + r) /2

K02: Jeśli r>1, to Sortowanie_przez_scalanie (tab, l,m)

K03: Jeśli r>1, to Sortowanie_przez_scalanie (tab, m+1, r)

K04: Scalaj (tab, l, m, r)

K05: Zakończ

# 4. Testy porównujące działanie obu metod na różnych próbkach danych

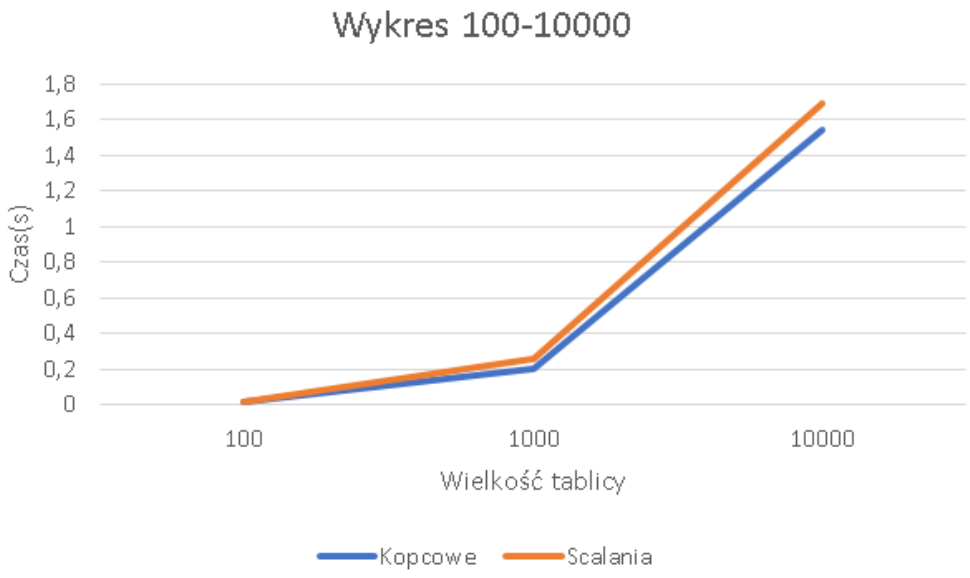
## 4.1 Liczby pseudolosowe (oczekiwany)

W pierwszym przypadku algorytm sortuje tablice wypełnioną liczbami pseudolosowymi. Wykorzystane zostały tablice dynamiczne co pozawala na sprawdzanie algorytmów na tablicach małych jak i dużych. Program będzie od nas wymagał podania wielkości tablicy.

Dzięki tablicy wypełnionej liczbami pseudolosowymi otrzymamy oczekiwaną złożoność czasową działania algorytmów.

| Rozmiar Tablicy | 100 | | 1000 | | 10000 | |
|---|---|---|---|---|---|---|
| Typ sortowania | Kopcowe | Scalania | Kopcowe | Scalania | Kopcowe | Scalania |
| Pomiar 1 [s] | 0,015 | 0,02 | 0,171 | 0,241 | 1,593 | 1,633 |
| Pomiar 2 [s] | 0,015 | 0,021 | 0,204 | 0,255 | 1,547 | 1,689 |
| Pomiar 3 [s] | 0,016 | 0,02 | 0,156 | 0,301 | 1,564 | 1,701 |
| Średnia pomiarów | 0,01533 | 0,02033 | 0,177 | 0,26567 | 1,568 | 1,67433 |

*Tabela 1 Pomiary czasów dla tablic 100-100000*

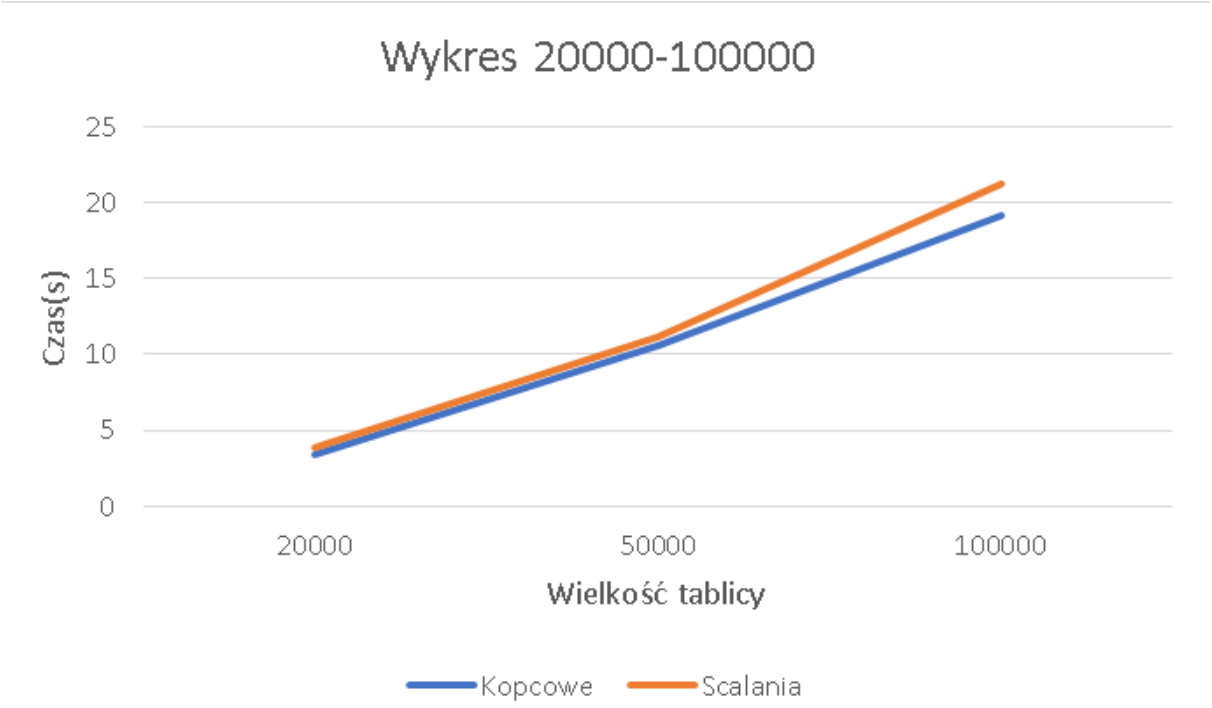



*Wykres 1 dla tablic 100-10000*

Jak przedstawia wykres powyżej od początku można zauważyć, że czasy obu algorytmów sortowania są podobne. Wynika to z faktu, że ich złożoność wynosi $O(n * logn)$.

| Rozmiar Tablicy | 20000 | | 50000 | | 100000 | |
|---|---|---|---|---|---|---|
| Typ sortowania | Kopcowe | Scalania | Kopcowe | Scalania | Kopcowe | Scalania |
| Pomiar 1 [s] | 3,157 | 3,902 | 10,609 | 11,253 | 18,884 | 20,512 |
| Pomiar 2 [s] | 3,415 | 3,821 | 10,601 | 11,115 | 19,142 | 21,214 |
| Pomiar 3 [s] | 3,213 | 3,802 | 10,355 | 11,632 | 18,924 | 20,701 |
| Średnia pomiarów | 3,26167 | 3,84167 | 10,5217 | 11,3333 | 18,9833 | 20,809 |

*Tabela 2 Pomiary czasów dla tablic 20000-100000*




*Wykres 2 dla tablic 20000-100000*

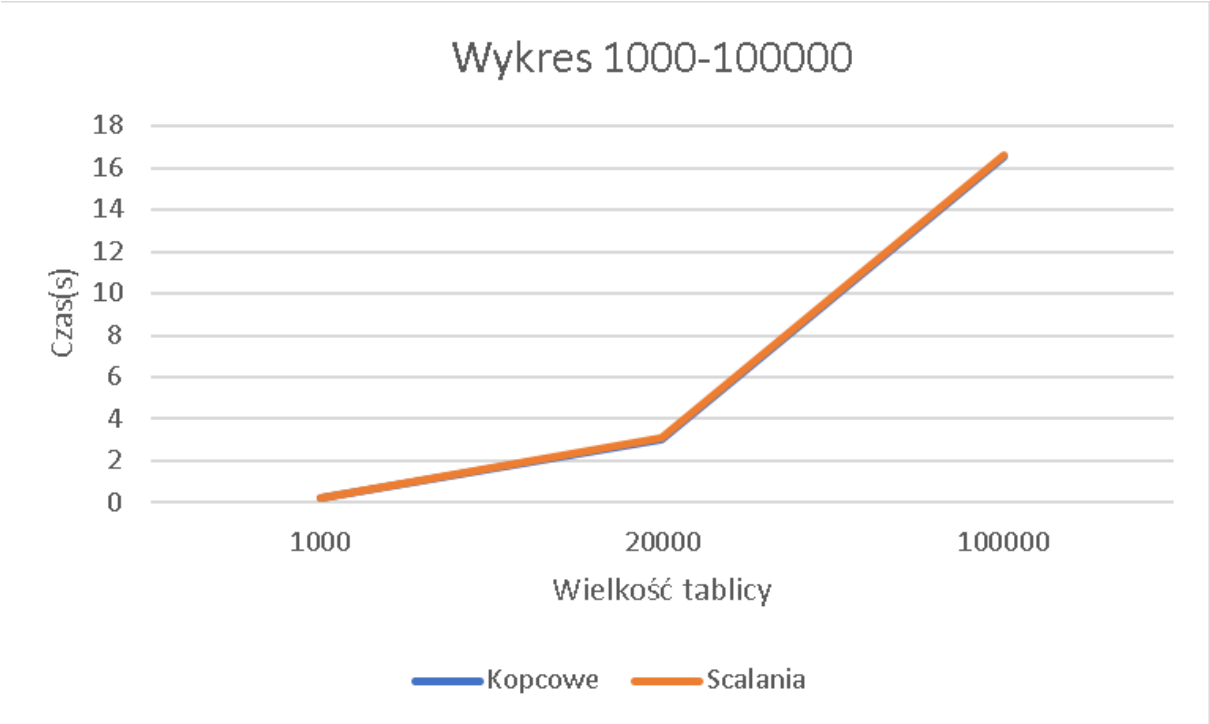
Dla większych tablic czas sortowania przebiega podobnie. Sortowanie przez kopcowanie z tablicą 100000- elementową poradziło sobie w czasie równym niespełna 19s, a sortowanie przez scalanie potrzebowało dla tej tablicy 2 sekundy więcej.

## 4.2 Przypadek optymistyczny dla obu sortowań

Optymistyczną złożoność czasową otrzymamy wtedy, gdy tablica, którą trzeba posortować, jest już posortowana. Do tego celu użyjemy możliwości pobrania danych do posortowania z pliku tekstowego.

| Rozmiar Tablicy | 1000 | | 20000 | | 100000 | |
|---|---|---|---|---|---|---|
| Typ sortowania | Kopcowe | Scalania | Kopcowe | Scalania | Kopcowe | Scalania |
| Pomiar 1 [s] | 0,156 | 0,191 | 3,031 | 3,04 | 16,5 | 16,733 |
| Pomiar 2 [s] | 0,187 | 0,198 | 2,984 | 3,102 | 16,537 | 16,619 |
| Pomiar 3 [s] | 0,156 | 0,16 | 2,996 | 3,067 | 16,524 | 16,701 |
| Średnia pomiarów | 0,1663 | 0,183 | 3,00367 | 3,0697 | 16,5203 | 16,684 |

*Tabela 3 Pomiary czasów dla tablic 1000-100000*




*Wykres 3 dla tablic 1000-100000*

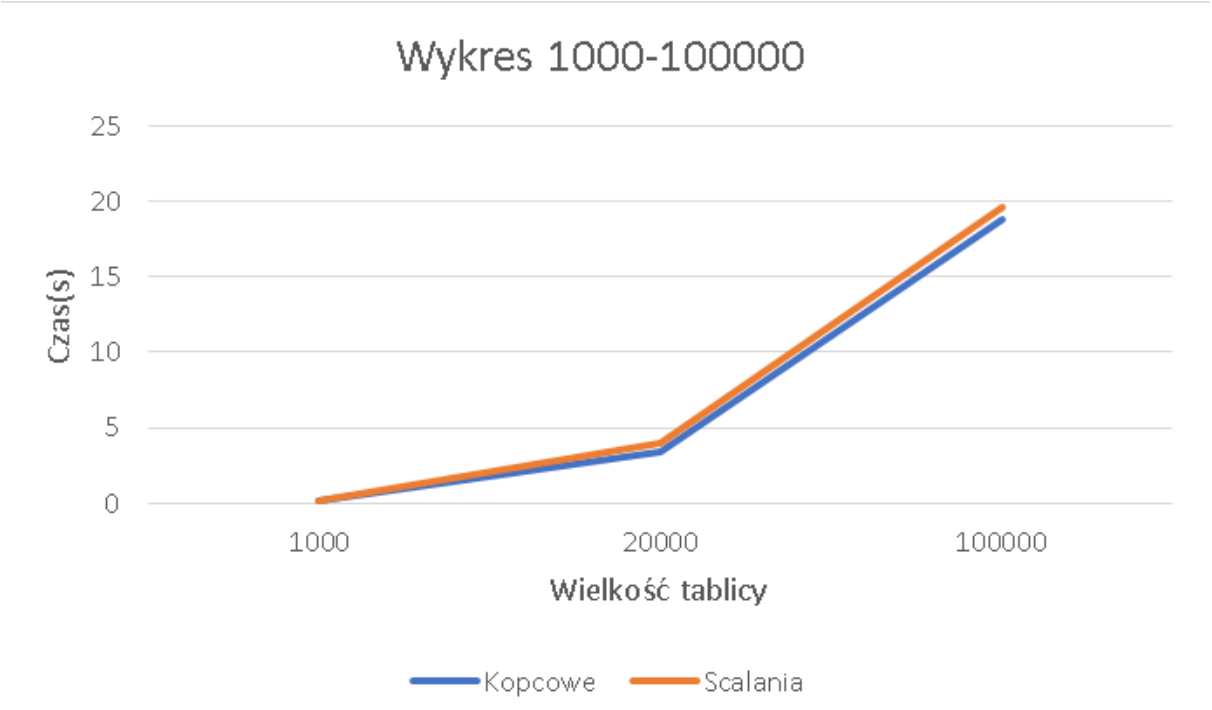
W tym przypadku, gdy tablice są już posortowane czasy pracy algorytmów uległ zmniejszeniu co widać w Tabela 3.

## 4.3 Przypadek pesymistyczny dla obu sortowań

Pesymistyczną złożoność czasową otrzymamy wtedy, gdy tablica, którą trzeba posortować, jest posortowana odwrotnie. Do tego celu użyjemy możliwości pobrania danych do posortowania z pliku tekstowego.

| Rozmiar Tablicy | 1000 | | 20000 | | 100000 | |
|---|---|---|---|---|---|---|
| Typ sortowania | Kopcowe | Scalania | Kopcowe | Scalania | Kopcowe | Scalania |
| Pomiar 1 [s] | 0,156 | 0,16 | 3,625 | 3,981 | 17,953 | 19,653 |
| Pomiar 2 [s] | 0,156 | 0,161 | 3,328 | 3,99 | 18,141 | 19,741 |
| Pomiar 3 [s] | 0,156 | 0,15 | 3,584 | 3,894 | 18,204 | 19,541 |
| Średnia pomiarów | 0,156 | 0,157 | 3,51233 | 3,955 | 18,0993 | 19,645 |

*Tabela 4 Pomiary czasów dla tablic 1000-100000*




*Wykres 4 dla tablic 1000-100000*

Jak widać w Tabela 4 czasy pomiarów zwiększyły się względem poprzednich badań. Różnice czasów sięgają nawet 10% przy większych tablicach.

# 5. Wnioski

Projekt został zrealizowany, działanie algorytmu jest poprawne. W programie główny algorytm został zaimplementowany w osobnej funkcji, która jest wywoływana w późniejszych etapach działania programu. Projekt posiada możliwość odczytu z pliku tekstowego jak i późniejsze zapisanie wyników do pliku tekstowego. W programie zostały umieszczone stosowane komentarze, które pomagają w zrozumieniu kodu również w sprawozdaniu zostały umieszczone elementy które ułatwiają nam zrozumienie działania algorytmu szukającego jest to pseudokod i schemat blokowy.