Politechnika Rzeszowska, 13.01.2021r.

Zadanie projektowe nr 2 Algorytmy i Struktury Danych

Inżynieria i analiza danych, I rok

Wiktor Barć
Numer indeksu: 166632
Grupa: P01

Spis treści

1. Wstęp

W niniejszym sprawozdaniu porównane zostały dwa algorytmy sortujące: sortowanie przez scalanie (ang. merge sort), oraz sortowanie szybkie (ang. quicksort). Są one jednymi z wielu algorytmów pozwalających sortować dane liczbowe w językach programowania. Na podstawie wykonanych testów oraz wykresów, została dokonana ocena, który z algorytmów jest szybszy i wydajniejszy w zależności od wprowadzonych danych.

2. Opis problemu

Zadane było zaimplementowanie obu algorytmów w języku C++, wykonanie testów na danych różnej wielkości, a także odpowiednia preparacja i porównanie wyników czasowych sortowania tych algorytmów, na których podstawie dokonana miała zostać ocena wydajności algorytmów.

3. Podstawy teoretyczne

3.1. Sortowanie przez scalanie (ang. merge sort)

Algorytm sortowania przez scalanie jest algorytmem rekurencyjnym. Opiera się na zasadzie **dziel i zwyciężaj**, która polega na podziale zadania głównego na mniejsze dotąd, aż zostanie ono rozwiązane. Algorytm sortujący dzieli porządkowany zbiór na kolejne połowy dopóki taki podział jest możliwy (tzn. podzbiór zawiera co najmniej dwa elementy). Następnie uzyskane w ten sposób części zbioru sortuje tym samym algorytmem. Posortowane części łączy ze sobą za pomocą scalania tak, aby wynikowy zbiór był posortowany.

3.2. Sortowanie szybkie (ang. quicksort)

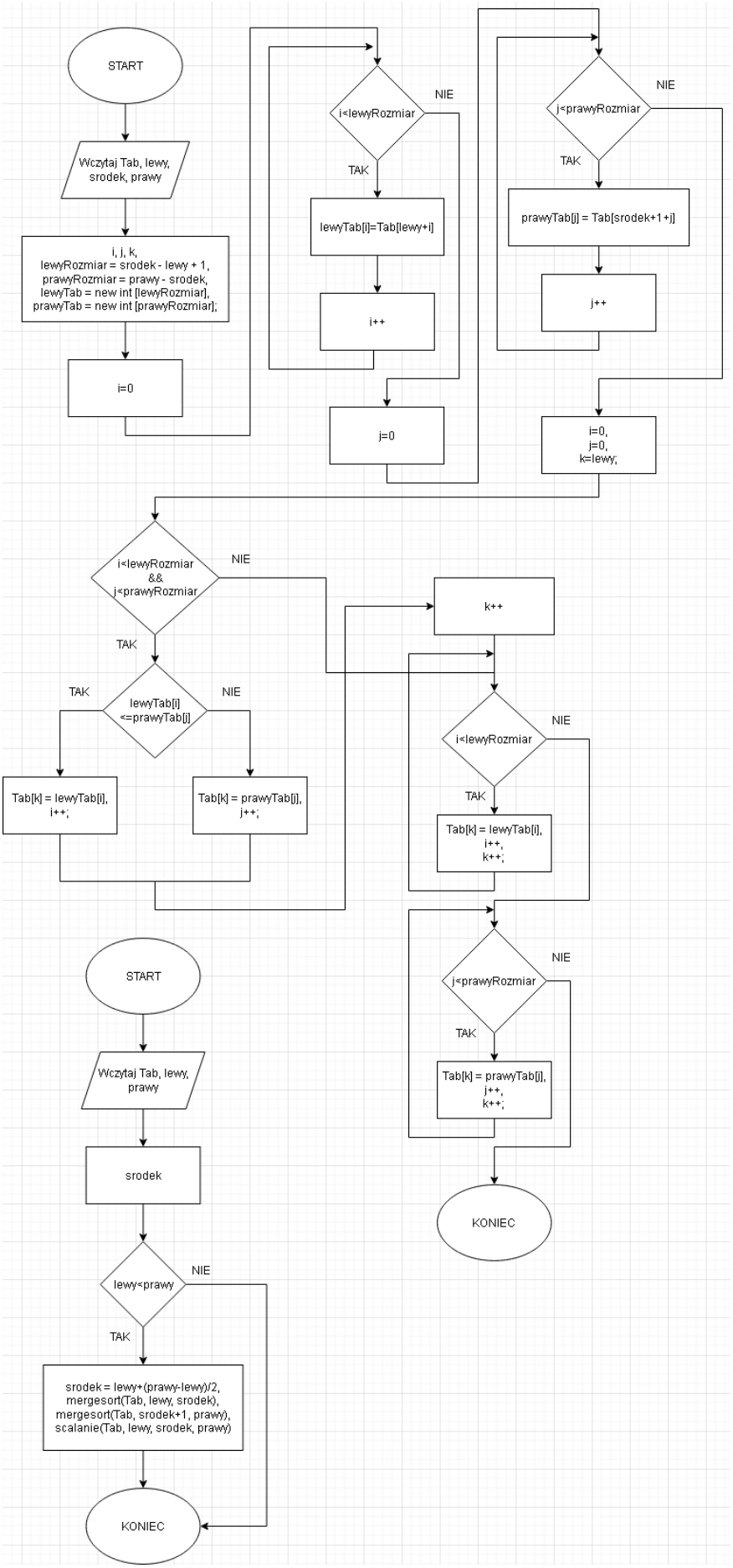
Algorytm sortowania szybkiego również opiera się na zasadzie **dziel i zwyciężaj.** Z tablicy wybiera się element rozdzielający (tzw. piwot), po czym tablica jest dzielona na dwa fragmenty: do lewej partycji przenoszone są wszystkie elementy nie większe od rozdzielającego, do prawej zaś wszystkie większe. Potem sortuje się osobno lewą i prawą część tablicy. Sortowanie kończy się, gdy kolejny fragment uzyskany z podziału zawiera pojedynczy element, jako że jednoelementowa tablica nie wymaga sortowania.

4. Szczegóły implementacji problemu

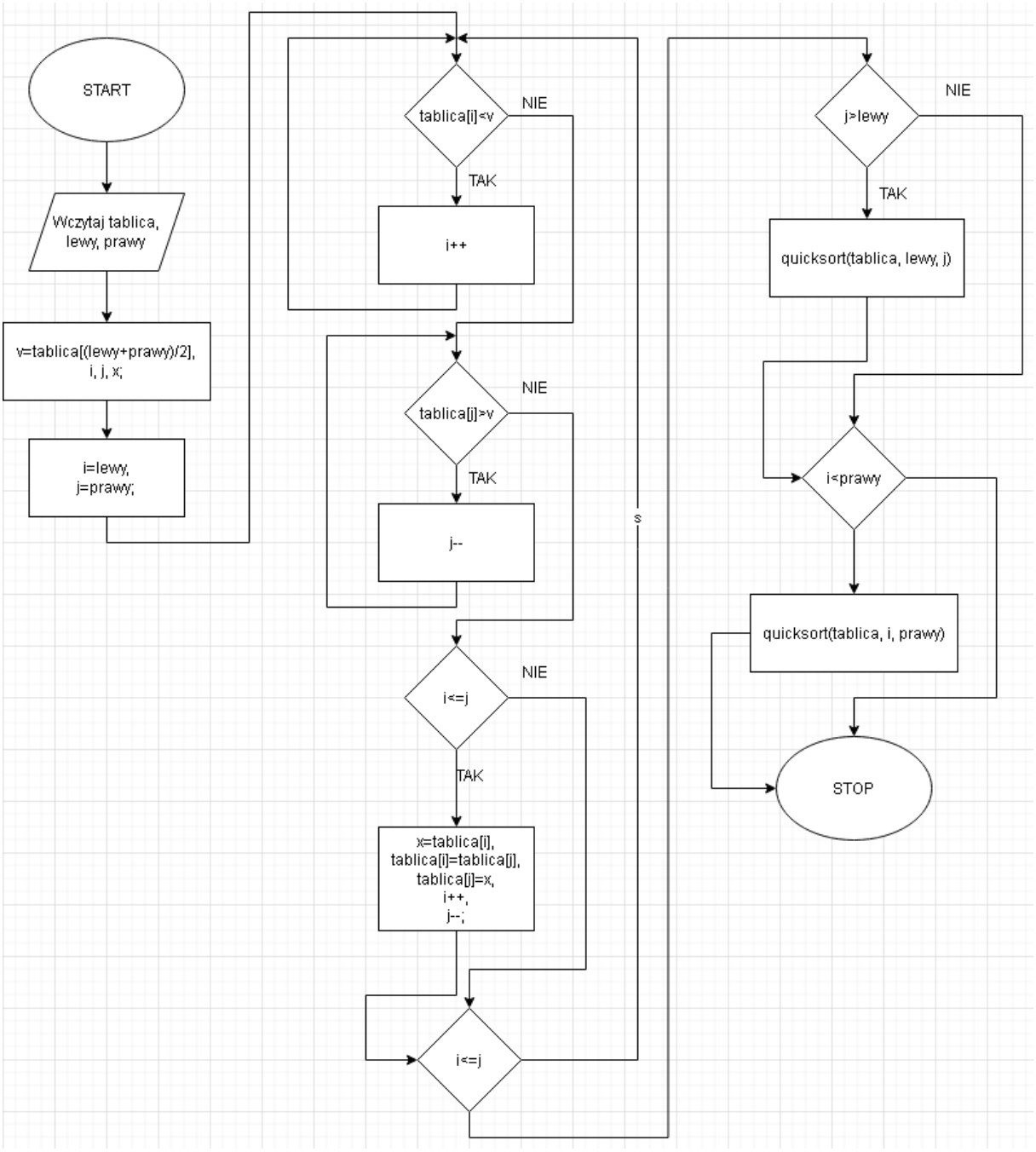
Implementację rozpocząłem od napisania prostych funkcji: losującej tablice (z możliwością kontrolowania losowanych danych, czyli zakresu wielkości i ilości liczb w tablicy), oraz wyświetlającej je w konsoli. Były one potrzebne do sprawdzenia poprawności działania algorytmów sortujących. Następnie do mojego programu wdrożyłem algorytmy sortujące (przez scalanie i szybkie). Na algorytm sortowania przez scalanie składały się dwie funkcje: scalająca podtablice, oraz realizująca sortowanie przez scalanie. Następnie zaimplementowałem pętlę for wczytującą losowe liczby do tablicy, oraz drugą przepisujące te same dane do drugiej tablicy, która była oddzielną dla drugiego algorytmu sortującego. Taki zabieg umożliwił wykonywanie sortowań dwoma algorytmami w jednym czasie, co z kolei przełożyło się na możliwość dodania kolejnej funkcji porównującej czasy sortowań obu algorytmów.

5. Schemat blokowy

5.1. Schemat sortowania przez scalanie



5.2. Schemat sortowania szybkiego



6. Pseudokod

6.1. Sortowanie przez scalanie

```
*Tab, lewy, srodek, prawy
i, j, k
lewyRozmiar=srodek-lewy+1
prawyRozmiar=prawy-srodek
*lewyTab=new int [lewy rozmiar]
*prawyTab=new int[prawy rozmiar]

dla i←0, i<lewyRozmiar, i++ wykonuj
      lewyTab[i]=Tab[lewy+i] (koniec warunku)
dla j←0, j<prawyRozmiar, j++ wykonuj
      prawyTab[j]=Tab[srodek+1+j] (koniec warunku)
i←0
j←0
k←lewy

kiedy i<lewyRozmiar&&j<prawyRozmiar
      jeżeli lewyTab[i]<=prawyTab[j] wykonuj
            Tab[k]=lewyTab[i]
            inkrementuj i (koniec warunku)
      w przeciwnym razie wykonuj
            Tab[k]=prawyTab[j]
            inkrementuj j (koniec warunku)
      inkrementuj k (koniec warunku)

kiedy i<lewyRozmiar wykonuj
      Tab[k]=lewyTab[i]
      inkrementuj i
      inkrementuj k (koniec warunku)

kiedy j<prawyRozmiar wykonuj
      Tab[k]=prawyTab[j]
      inkrementuj j
      inkrementuj k (koniec warunku)

*Tab, lewy, prawy
srodek

jeżeli lewy<prawy wykonuj
      srodek=lewy+(prawy-lewy)/2
      mergesort(Tab, lewy, srodek)
      mergesort(Tab, srodek+1, prawy)
      scalanie(Tab, lewy, srodek, prawy) (koniec warunku)
```

6.2. Sortowanie szybkie

```
*tablica, lewy, prawy
v←tablica[(lewy+prawy)/2]
i, j, x
i←lewy
j←prawy

wykonuj
    kiedy tablica[i]<v wykonuj
        inkrementuj i (koniec warunku)
    kiedy tablica[j]>v wykonuj
        dekrementuj j (koniec warunku)

jeżeli i<=j wykonuj
    x=tablica[i]
    tablica[i]=tablica[j]
    tablica[j]=x
    inkrementuj i
    dekrementuj j (koniec warunku)

kiedy i<=j wykonuj
    jeżeli j>lewy wykonuj
        quicksort(tablica, lewy, j) (koniec warunku)
    jeżeli i<prawy wykonuj
        quicksort(tablica, i, prawy) (koniec warunku)
```

7. Złożoność obliczeniowa

7.1. Sortowanie przez scalanie

Średnia złożoność obliczeniowa dla sortowania przez scalanie to O(n log n) dla wszystkich złożoności.

7.2. Sortowanie szybkie

Średnia złożoność obliczeniowa dla sortowania szybkiego to również O(n log n) dla wszystkich złożoności.

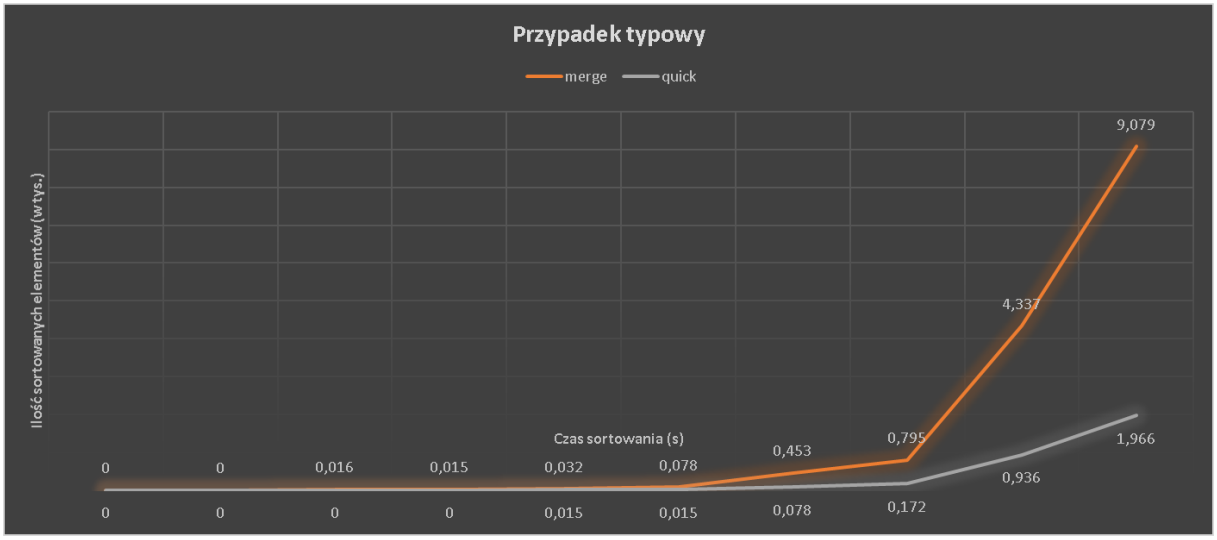
8. Testy

8.1. Przypadek typowy

Ilość sortowanych elementów (w tys.):
0,1; 1; 5; 10; 50; 100; 500; 1000; 5000; 10000;

Rozmiar sortowanych elementów: od 1 do 100tys.

Czasy sortowań widoczne w punktach przegięcia wykresów.



Dla przypadku typowego (tj. losowo pozycjonowanych sortowanych liczb) oba algorytmy w przypadku mniejszej ilości sortowanych liczb (tzn. od 100 do 10 tys.) radzą sobie bardzo podobnie. Ich czasy sortowań są tak krótkie, że program analizujący uznaje wynik za niemalże zerowy i taką też zwraca wartość.

W przypadku większej ilości liczb niż 10 tys. elementów, czas sortowania przez scalanie widocznie się wydłuża, podczas gdy sortowanie szybkie osiąga wyniki podobne do poprzednich.

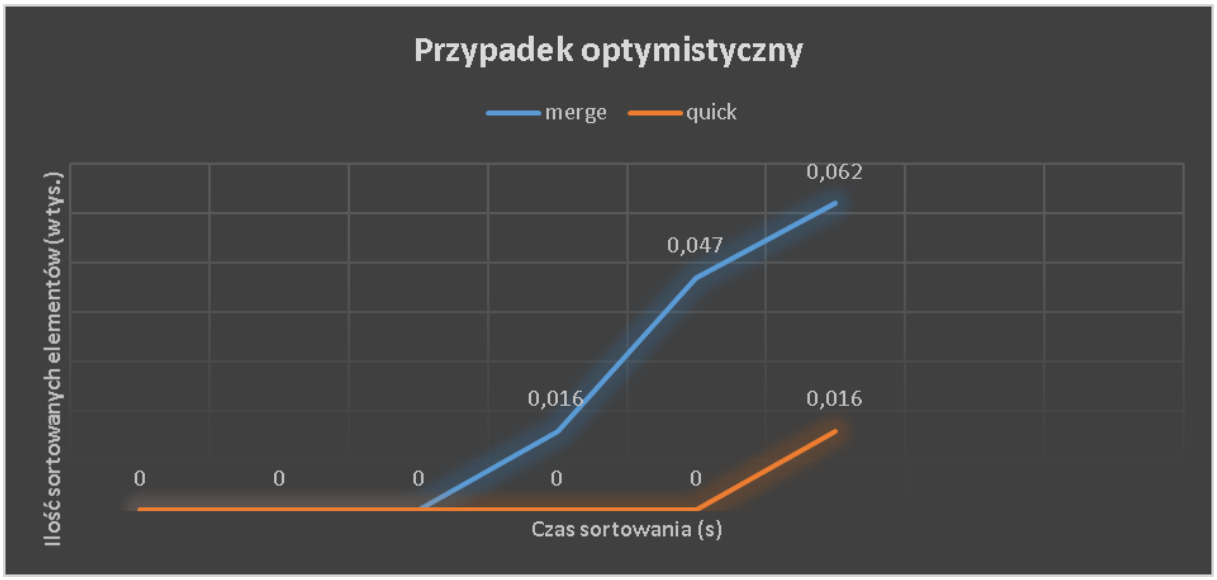
Jeżeli chodzi o sortowanie bardzo dużej liczby elementów (tzn. powyżej 500 tys.) różnice pomiędzy algorytmami zwiększają się, ze zdecydowaną przewagą sortowania szybkiego, które nawet dla 5 mln sortowanych elementów, uzyskuje czas niecałych 2 sekund, podczas gdy posortowanie identycznego zbioru liczb zajęło przeciwnikowi ponad 9 sekund.

8.2. Przypadek optymistyczny

Ilość sortowanych elementów (w tys.):
0,1; 1; 5; 10; 50; 100;

Rozmiar sortowanych elementów: od 1 do 100tys.

Czasy sortowań widoczne w punktach przegięcia wykresów.



Dla przypadku optymistycznego (tj. liczb uporządkowanych rosnąco, z niewielką liczbą zamienionych miejscami elementów) czasy sortowania elementów o ilości do 5 tys. są bliskie zeru.

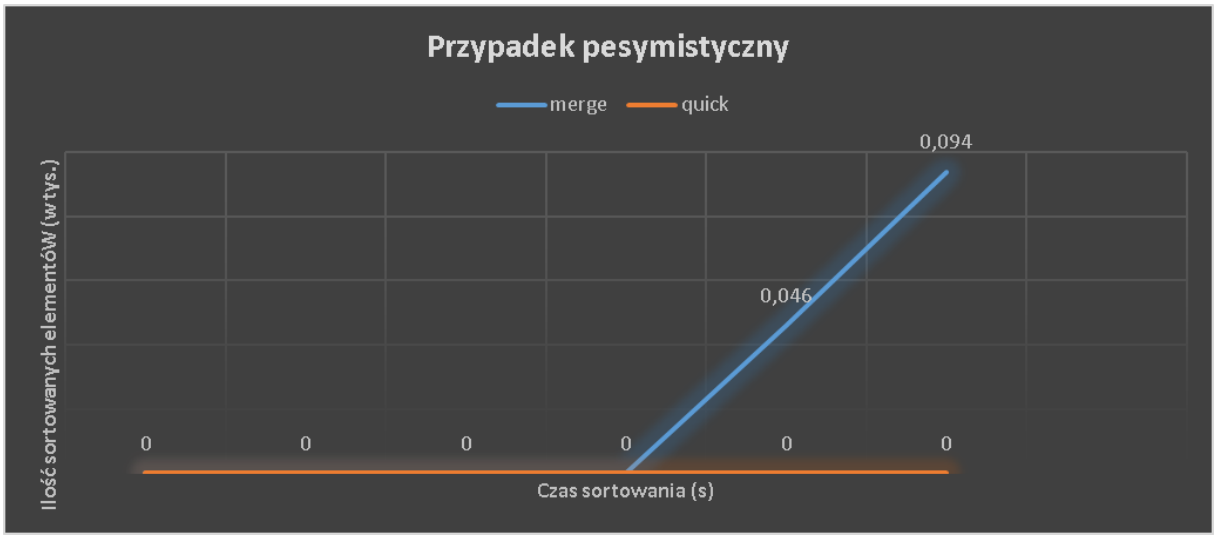
W przypadku większej ilości sortowanych elementów (powyżej 5 tys.) czas sortowania obu algorytmów są bardzo niskie (na poziomie poniżej 1/10 sekundy), jednak z przewagą sortowania szybkiego.

8.3. Przypadek pesymistyczny

Ilość sortowanych elementów (w tys.):
0,1; 1; 5; 10; 50; 100;

Rozmiar sortowanych elementów: od 1 do 100tys.

Czasy sortowań widoczne w punktach przegięcia wykresów.



Dla przypadku pesymistycznego (tj. liczb uporządkowanych malejąco) czas sortowania elementów o ilości do 100 tys. dla algorytmu sortowania szybkiego wynosi niemalże zero, w przeciwieństwie do algorytmu sortowania przez scalanie, który wynikami znacznie odbiega od przeciwnika, choć te są równie nieznaczne (na poziomie poniżej 1/10 sekundy).

9. Wnioski i podsumowanie

Dane i sporządzone na ich podstawie wykresy dają nam niejasny werdykt. Czasy sortowań obu algorytmów w identycznych warunkach odbiegają od siebie nieznacznie. Dopiero ogromna liczba sortowanych elementów daje nam wyniki, w których jasno widać przewagę algorytmu sortowania szybkiego nad algorytmem sortowania przez scalanie.

Zarówno jeden jak i drugi algorytm jest prosty do zrozumienia. Oba opierają się na tej samej zasadzie **dziel i zwyciężaj** z niewielkimi różnicami, na przewagę algorytmu szybkiego, który jest bardzo uniwersalny i bardzo dobrze spełniający się w każdym przypadku. Nie bez powodu jego nazwa mówi nam o faktycznej przewadze nad pozostałym algorytmami sortującymi.

Załącznik – kod programu

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <time.h>
#include <fstream>

using namespace std;

int ile, rozmiar;
clock_t start,stop;
double czas;

void scalanie(int *Tab, int lewy, int srodek, int prawy) // funkcja scalajaca podtablice
{
   int i, j, k; // zmienne iteracyjne do petli "for"
   int lewyRozmiar = srodek-lewy+1; // rozmiar lewej podtablicy
   int prawyRozmiar = prawy-srodek; // rozmiar prawej podtablicy
   int *lewyTab = new int[lewyRozmiar]; // deklaracja lewej podtablicy o odpowiednim rozmiarze
   int *prawyTab = new int[prawyRozmiar]; // deklaracja prawej podtablicy o odpowiednim rozmiarze
   for(i = 0; i<lewyRozmiar; i++)
   {
      lewyTab[i] = Tab[lewy+i]; // wypelnienie lewej podtablicy odpowiednimi liczbami z tablicy
   }
   for(j = 0; j<prawyRozmiar; j++)
   {
      prawyTab[j] = Tab[srodek+1+j];  // wypelnienie prawej podtablicy odpowiednimi liczbami z tablicy
   }
   i = 0;
   j = 0;
   k = lewy; // przypisanie wartosci zmiennym iteracyjnym potrzebnym do petli "for"
   while(i < lewyRozmiar && j<prawyRozmiar)
   {
      if(lewyTab[i] <= prawyTab[j])
      {
         Tab[k] = lewyTab[i];    // scalenie podtablic z tablica glowna
         i++;                    // inkrementacja zmiennej i
      }
      else
      {
```

```
            Tab[k] = prawyTab[j];
            j++; // inkrementacja zmiennej j
        }
        k++; // inkrementacja zmiennej k
    }
    while(i<lewyRozmiar)
    {
        Tab[k] = lewyTab[i];    // dodatkowy element z lewej podtablicy
        i++;
        k++;   // inkrementacja zmiennej k i zmiennej i
    }
    while(j<prawyRozmiar)
    {
        Tab[k] = prawyTab[j];   //dodatkowy element z prawej podtablicy
        j++;
        k++; // inkrementacja zmiennej k i zmiennej j
    }
}

void mergesort(int *Tab, int lewy, int prawy) // funkcja realizujaca sortowanie przez scalanie
{
    int srodek; // zmienna przechowujaca srodek tablicy
    if(lewy < prawy)
    {
        srodek = lewy+(prawy-lewy)/2; // wyznaczenie srodka tablicy, potrzebnego do podzialu
na 2 tablice,
        mergesort(Tab, lewy, srodek); // sortowanie rekurencyjnie lewej czesci tablicy
        mergesort(Tab, srodek+1, prawy);    // sortowanie rekurencyjnie prawej czesci tablicy
        scalanie(Tab, lewy, srodek, prawy); // scalenie lewej i prawej czesci tablicy
    }
}

//sortowanie quicksort
void quicksort(int *tablica, int lewy, int prawy)  //funkcja realizująca sortowanie metoda
quicksort
{
    int v=tablica[(lewy+prawy)/2];  //zmienna v to obrany piwot
    int i,j,x;  //zmienne iteracyjne dla petli while
    i=lewy;
    j=prawy;
    do
    {
        while(tablica[i]<v)  //przeniesienie do lewej tablicy elementow mniejszych od piwota
            i++;
        while(tablica[j]>v)  //przeniesienie do lewej tablicy elementow wiekszych od piwota
            j--;
        if(i<=j)
```

```
        {
            x=tablica[i];
            tablica[i]=tablica[j];
            tablica[j]=x;
            i++;
            j--;
        }
    }
    while(i<=j);
    if(j>lewy)
        quicksort(tablica,lewy, j);  // sortowanie rekurencyjnie lewej czesci tablicy
    if(i<prawy)
        quicksort(tablica, i, prawy);  // sortowanie rekurencyjnie prawej czesci tablicy
}

int main()
{
    fstream plik, wyniki;
    // plik.open("tablica.txt", ios::in);  //funkcja do wczytywania tablic do testow
    wyniki.open("wyniki.txt", ios::out);
    cout << "Porownanie czasow sortowania" << endl;

    cout<<"Ile losowych liczb w tablicy: ";
    cin>>ile;

    //dynamiczna alokacja tablicy

    int *tablica;
    tablica=new int [ile];

    int *tablica2;
    tablica2=new int [ile];

    //inicjowanie generatora
    srand(time(NULL));

     //wczytywanie losowych liczb do tablicy
     for(int i=0; i<ile; i++)
    {
        tablica[i] = rand()%100000+1;
    }

/*
    int ile=1000;  //liczba elementow w pliku  //funkcja do wczytywania tablic do testow
    int tablica[ile];  //zadeklarowanie dwoch tablic
    int tablica2[ile];
```

```
   for(int i=0; i<ile; i++) //petla wpisujaca elementy z pliku do tablicy
   {
      plik>>tablica[i];
   }
*/

   //przepisanie tablicy do tablicy 2
   for(int i=0; i<ile; i++)
   {
      tablica2[i]=tablica[i];
   }


   //testowe wypisanie losowo wygenerowanej tablicy
   cout<<"Przed posortowaniem: "<<endl;
   wyniki<<"Przed posortowaniem: "<<endl;
      for(int i=0; i<ile; i++)
      {
         cout<<tablica[i]<<" ";      //mozliwosc zmiany na tablica2[i] w celu sprawdzenia poprawnoœci sortowania metoda quicksort
         wyniki<<tablica[i]<<" ";
      }


   cout<<endl<<"Sortowanie  algorytmem przez scalanie."<<endl;
   start = clock();  //zanotowanie liczby cykli procesora w momencie rozpoczecia algorytmu
   mergesort(tablica, 0, ile-1);  //wywolanie funkcji
   stop = clock();  //zatrzymanie pobierania liczby cykli
   czas = (double)(stop-start) / CLOCKS_PER_SEC;  //podzielenie liczby cykli przez stala oraz zrzutowanie na typ double
   cout<<endl<<"Czas sortowania przez scalanie: "<<czas<<" s"<<endl;  //wypisanie czasu sortowania
   wyniki<<"\n"<<endl<<"Czas sortowania przez scalanie: "<<czas<<" s"<<endl;

   cout<<endl<<"Sortowanie algorytmem quicksort."<<endl;
   start = clock();
   quicksort(tablica2, 0, ile-1);
   stop = clock();
   czas = (double)(stop-start) / CLOCKS_PER_SEC;
   cout<<endl<<"Czas sortowania quicksort: "<<czas<<" s"<<endl;
   wyniki<<endl<<"Czas sortowania quicksort: "<<czas<<" s"<<endl;


   //testowe wypisanie posortowanej, losowo wygenerowanej tablicy
   cout<<"Po posortowaniu: "<<endl;
   wyniki<<"\n"<<"Po posortowaniu: "<<endl;
      for(int i=0; i<ile; i++)
```

```
        {
            cout<<tablica[i]<<" ";      //mozliwosc zmiany na tablica2[i] w celu sprawdzenia poprawnoœci sortowania metoda quicksort
            wyniki<<tablica[i]<<" ";
        }

    plik.close();
    wyniki.close();

    delete [] tablica;
    delete [] tablica2;

    return 0;
}
```