

SPRAWOZDANIE

zadanie projektowe II

Temat zadania:

Zaimplementuj sortowanie przez scalanie oraz sortowanie metodą quicksort.

Politechnika Rzeszowska im. Ignacego Łukasiewicza

Wydział Matematyki i Fizyki Stosowanej

Algorytmy i Struktury Danych

-zajęcia projektowe grupa L8

Weronika Wojtuła

Wstęp

W sprawozdaniu będę starać się opisać algorytm sortowania szybkiego (QuickSort) oraz algorytm sortowania przez scalanie (MergeSort). Oba algorytmy oparte są na strategii “dziel i zwyciężaj” (ang. *“divide and conquer”*). Możemy ją opisać w trzech krokach:

1. Dziel :

problem dzieli się rekurencyjnie na dwa lub więcej mniejszych podproblemów tego samego (lub podobnego) typu, tak długo, aż fragmenty staną się wystarczająco proste do bezpośredniego rozwiązania

2. Zwyciężaj :

rozwiązujemy dane podproblemy i znajdujemy ich rozwiązanie

3. Połącz:

scalamy rozwiązania podproblemów, uzyskując rozwiązanie problemu

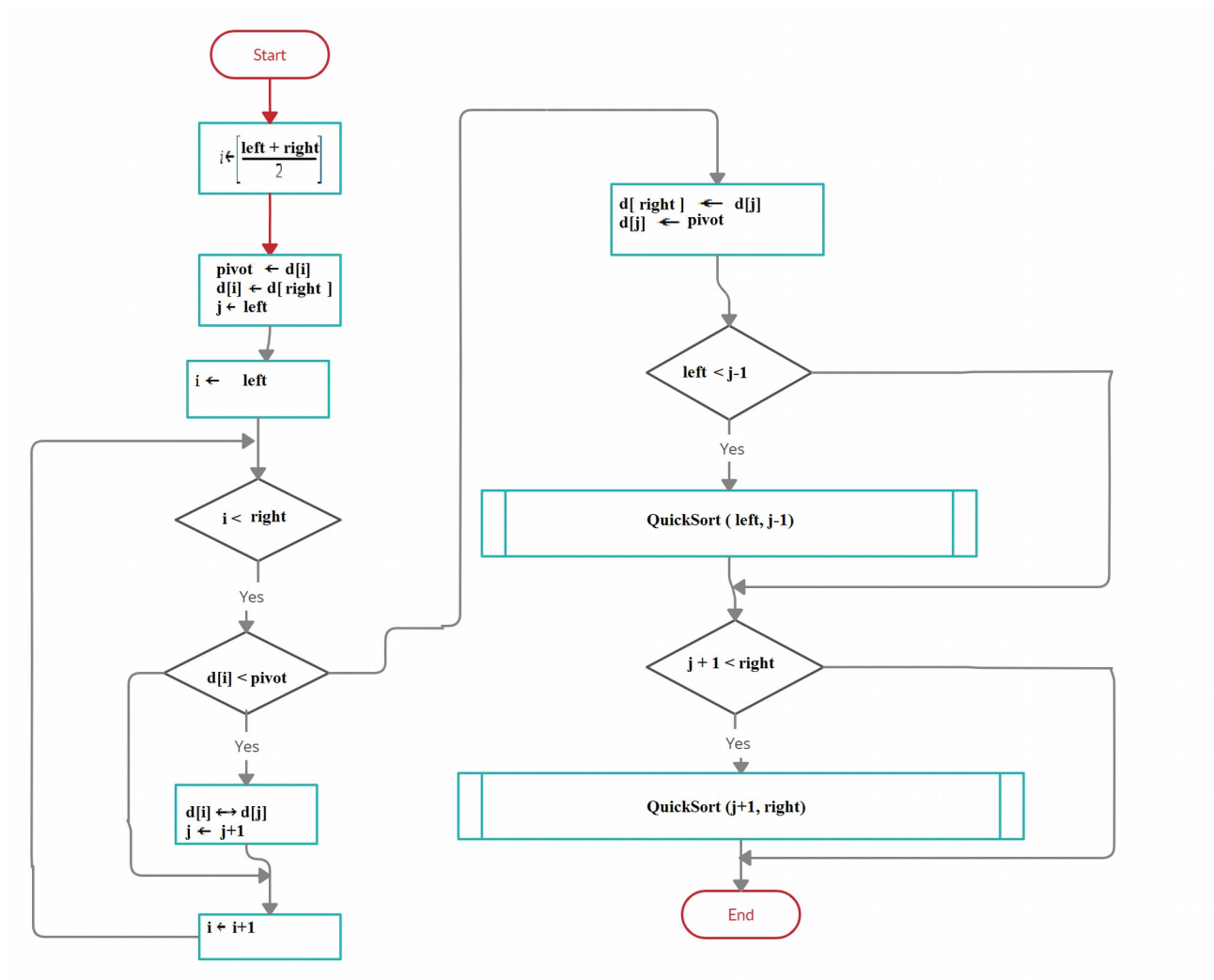
Sortowanie szybkie

Sortowanie szybkie (inaczej QuickSort) zostało wynalezione przez angielskiego informatyka, profesora Tony'ego Hoare'a w latach 60-tych ubiegłego wieku.

1. Zasada działania:

Algorytm sortowania szybkiego jak w wstępie zostało podane opiera się na strategii "Dziel i zwyciężaj". Jego działanie opiera się na dzieleniu tablicy na dwie części sposobem tzw. **tworzenie partycji**. Do utworzenia partycji ze zbioru trzeba wybrać element, który nazwiemy *pivot'em* (może on znajdować się w środku, na początku, na końcu lub w losowym miejscu). Wszystkie elementy leżące po pierwszej stronie (zwanej **lewą partycją**) są nie większe od *pivot'u*, a elementy leżące w drugiej części (zwanej **prawą partycją**) są większe od *pivot'u*. Dzielenie na partycje polega na umieszczeniu dwóch wskaźników na początku zbioru -*i* oraz *j*. Wskaźnik *i* przebiega przez zbiór poszukując wartości mniejszych od *pivot'u*. Po znalezieniu takiej wartości jest ona wymieniana z elementem na pozycji *j*. Po tej operacji wskaźnik *j* jest przesuwany na następną pozycję. Wskaźnik *j* zapamiętuje pozycję, na którą trafi następny element oraz na końcu wskazuje miejsce, gdzie znajdzie się *pivot*. W trakcie podziału *pivot* jest bezpiecznie przechowywany na ostatniej pozycji w zbiorze. Po zakończeniu podziału na partycje wskaźnik *j* wyznacza pozycję piwotu. Lewa partycja zawiera elementy mniejsze od *pivot'u* i rozciąga się od początku zbioru do pozycji $(j-1)$. Prawa partycja zawiera elementy większe lub równe *pivot'owi* i rozciąga się od pozycji $(j+1)$ do końca zbioru. Na koniec połączenie tych partycji daje jeden zbiór z wynikiem posortowanych danych wejściowych.

2. Schemat blokowy algorytmu:



3. Pseudokod algorytmu :

Dane wejściowe :

$d[]$ - zbiór zawierający dane do posortowania;

left - indeks pierwszego elementu w zbiorze ;

right – indeks ostatniego elementu w zbiorze;

Dane wyjściowe :

$d[]$ -zbiór zawierający elementy posortowane rosnąco;

Zmienne pomocnicze:

pivot – element podziałowy

i, j -indeksy i, j

Lista kroków:

- K01:
 $i \leftarrow \lfloor (left + right) / 2 \rfloor$
- K02:
 pivot $\leftarrow d[i]$;
 $d[i] \leftarrow d[right]$;
 $j \leftarrow left$;;
- K03:
 for $i = left + 1, \dots, right - 1$;
 do K04.... K05
- K04:
 if $d[i] \geq pivot$
 do again K03
- K05:
 $d[i] \leftrightarrow d[j]$;
 $j \leftarrow j + 1$;
- K06:
 $d[right] \leftarrow d[j]$;
 $d[j] \leftarrow pivot$;
- K07:
 if $left < j - 1$
 then QuickSort (left, j-1)
- K08:
 if $j + 1 < right$
 then QuickSort (j+1 , right)
- K09:
 END

4. Złożoność obliczeniowa :

Czas działania algorytmu sortowania jak zarówno zapotrzebowanie na pamięć jest uzależnione od postaci tablicy wejściowej. Od tego zależy również czy podziały dokonywane w algorytmie są zrównoważone, czy też nie. A to z kolei od wybranego klucza podziału. Algorytm jest szybki w przypadku zrównoważonego podziału. Natomiast gdy podziały nie są zrównoważone, sortowanie może przebiegać wolno. Złożoność algorytmu sortowania szybkiego możemy rozpatrywać analizując trzy możliwości:

- przypadek przecięty (dla zbiorów o losowym rozkładzie elementów)

Średni czas działania algorytmu jest bliski najlepszemu przypadkowi QuickSort. Podziały przeciętnie wypadają w połowie w większości sytuacji. Dla tak równomiernego rozkładu, algorytm sortowania wymaga czasu działania :

$$T(n) = O(2n \cdot \log n);$$

- przypadek optymistyczny (dla zbiorów uporządkowanych)

Za każdym razem jako *pivot* wybierana zostaje mediana z sortowanego fragmentu tablicy, czyli gdy każdy podział daje równe podzbiory danych. Wówczas liczba porównań potrzebnych do posortowania tablicy jest rzędu:

$$T(n) = O(n \cdot \log n);$$

- przypadek pesymistyczny (dla zbiorów posortowanych odwrotnie)

W przypadku pesymistycznym, polegającym na każdorazowym wyborem elementu najmniejszego, bądź największego w sortowaniu tablicy, złożoność obliczeniowa ma postać:

$$T(n) = O(n^2);$$

W przypadku mojego algorytmu, sortowanie tablicy posortowanej odwrotnie nie jest przypadkiem pesymistycznym. Ponieważ czas sortowania tablicy uporządkowanej i tablicy uporządkowanej odwrotnie jest taki sam.

Sortowanie przez scalanie :

Odkrycie algorytmu przypisuje się Johnowi von Neumannowi.

1. Zasada działania

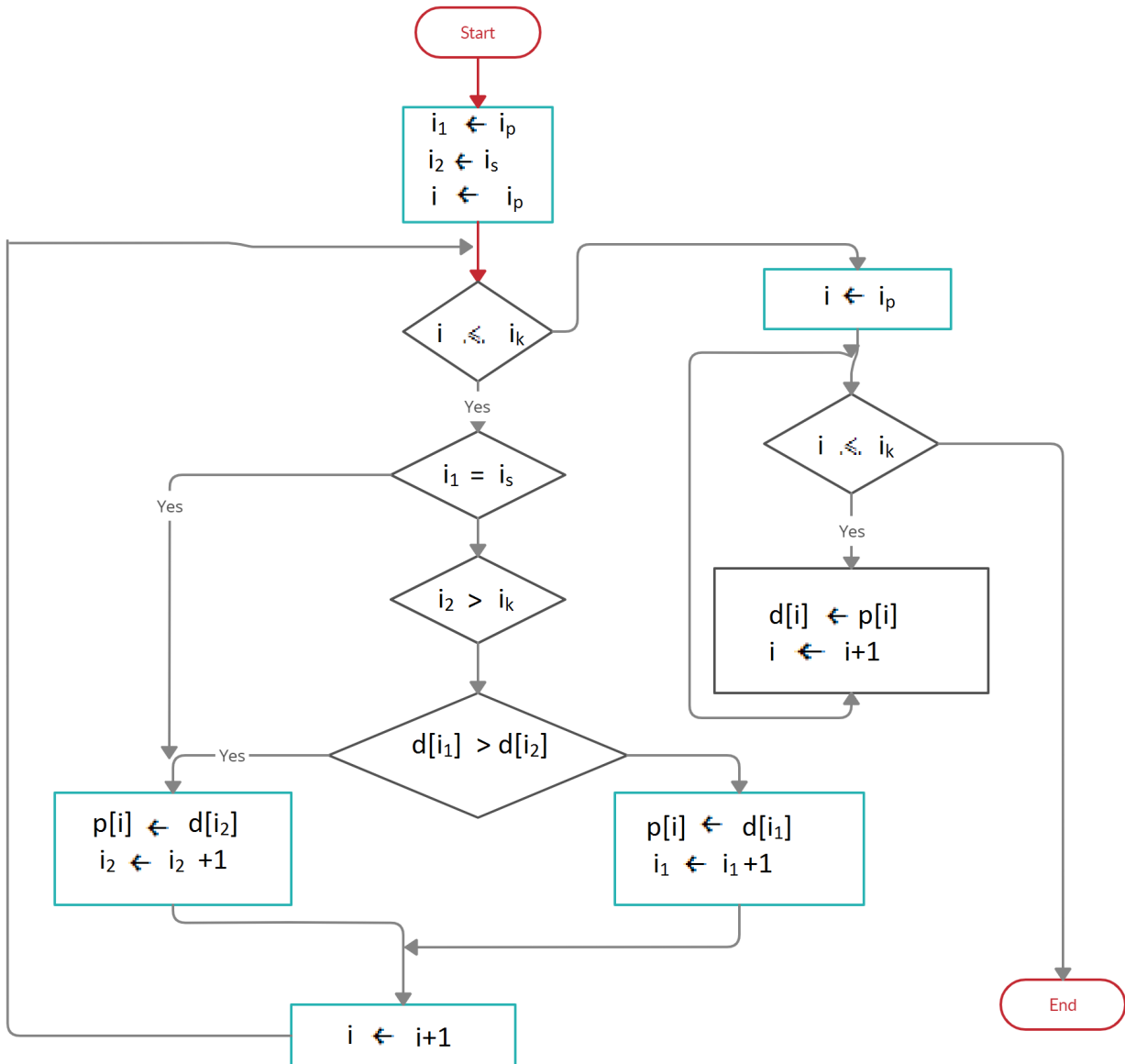
Algorytm wykorzystuje zasadę „*dziel i zwyciężaj*”, która polega na podziale zadania głównego na zadania mniejsze dotąd, aż rozwiązanie stanie się oczywiste. Algorytm sortujący dzieli porządkowany zbiór na kolejne połowy dopóki taki podział jest możliwy (tzn. podzbiór zawiera co najmniej dwa elementy). Następnie uzyskane w ten sposób części zbioru rekurencyjnie sortuje tym samym algorytmem. Posortowane części łączy ze sobą za pomocą scalania tak, aby wynikowy zbiór był posortowany.

Podstawową operacją algorytmu jest scalanie dwóch zbiorów uporządkowanych w jeden zbiór również uporządkowany. Operację scalania realizujemy wykorzystując pomocniczy zbiór, w którym będziemy tymczasowo odkładać scalane elementy zbiorów.

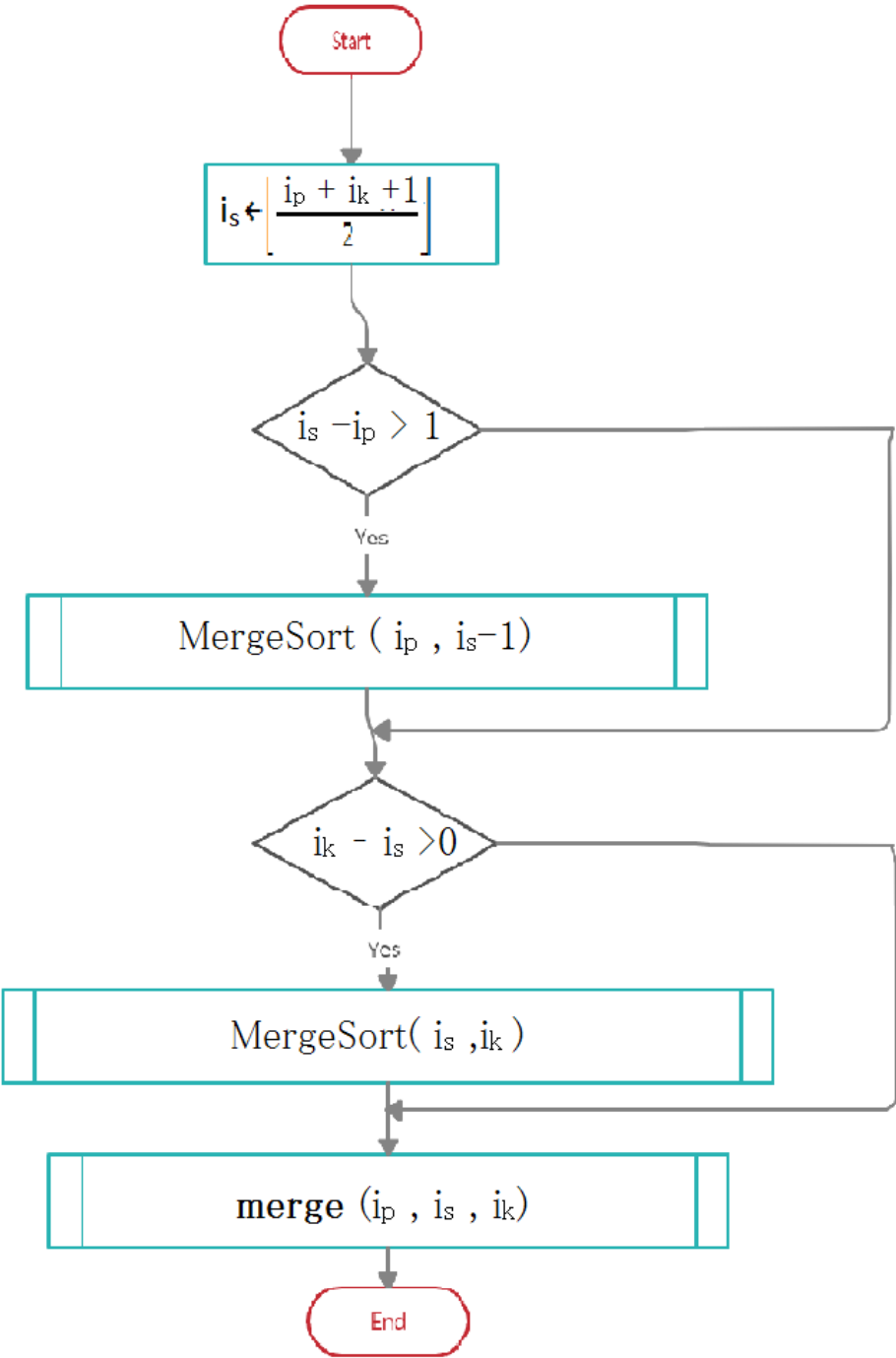
Zaczynamy od przygotowania pustego zbioru tymczasowego. Dopóki żaden z scalanych zbiorów nie jest pusty, algorytm porównuje ze sobą pierwsze elementy z dwóch zbiorów i w zbiorze tymczasowym umieszcza mniejszy z elementów usuwając go z scalonego zbioru. Algorytm w zbiorze tymczasowym umieszcza elementy z scalanego zbioru dopóki zawiera jeszcze elementy. Na koniec przepisuje zbiór tymczasowy do zbioru wynikowego.

2. Schemat blokowy:

2.1 Schemat blokowy algorytmu scalania:



2.2 Schemat blokowy algorytmu sortowania:



3. Pseudokod algorytmu:

Dane wejściowe:

$d[]$ - sortowany zbiór;

i_p - indeks pierwszego elementu w młodym podzbiorze;

i_k - indeks ostatniego elementu w starym podzbiorze;

Dane wyjściowe:

$d[]$ - posortowany zbiór;

Zmienne pomocnicze:

i_s - indeks pierwszego elementu w starym podzbiorze;

3.1 Pseudokod algorytmu scalania:

Lista kroków algorytmu scalania :

- K01:

$i_1 \leftarrow i_p; i_2 \leftarrow i_s; i \leftarrow i_p;$

- K02:

for $i = i_p, i_p + 1, \dots, i_k;$

if $(i_1 = i_s) \vee (i_2 \leq i_k$ **and** $d[i_1] > d[i_2];$

then $p[i] \leftarrow d[i_2]; i_2 \leftarrow i_2 + 1$

else $p[i] \leftarrow d[i_1]; i_1 \leftarrow i_1 + 1$

- K03:

for $i = i_p; i_p + 1, \dots, i_k;$

$d[i] \leftarrow p[i]$

- K04:

END

3.2 Pseudokod algorytmu sortowania:

Lista kroków algorytmu sortowania:

- K01:
 $i_s \leftarrow (i_p + i_k + 1) \text{ div } 2$
- K02:
if $i_s - i_p > 1$ **then** MergeSort (i_p , $i_s - 1$)
- K03:
if $i_k - i_s > 0$ **then** MergeSort(i_s , i_k)
- K04:
merge (i_p , i_s , i_k)
- K05:
END

4. Złożoność obliczeniowa :

Podczas sortowania n obiektów sortowanie przez scalanie ma średnią i najgorszą wydajność równą $O(n \log n)$. W najgorszym przypadku scalania sortowania korzysta około 39% mniej porównań niż Quicksort robi w jego przeciętnego przypadku i pod względem ruchów. Sortowanie przez scalanie w najgorszym przypadku złożoność jest $O(n \log n)$.

- przypadek przeciętny : (dla zbiorów o losowym rozkładzie)

$$T(n) = O(n \log n);$$

- przypadek optymistyczny: (dla zbiorów uporządkowanych (z niewielką liczbą elementów nie na swoim miejscu)

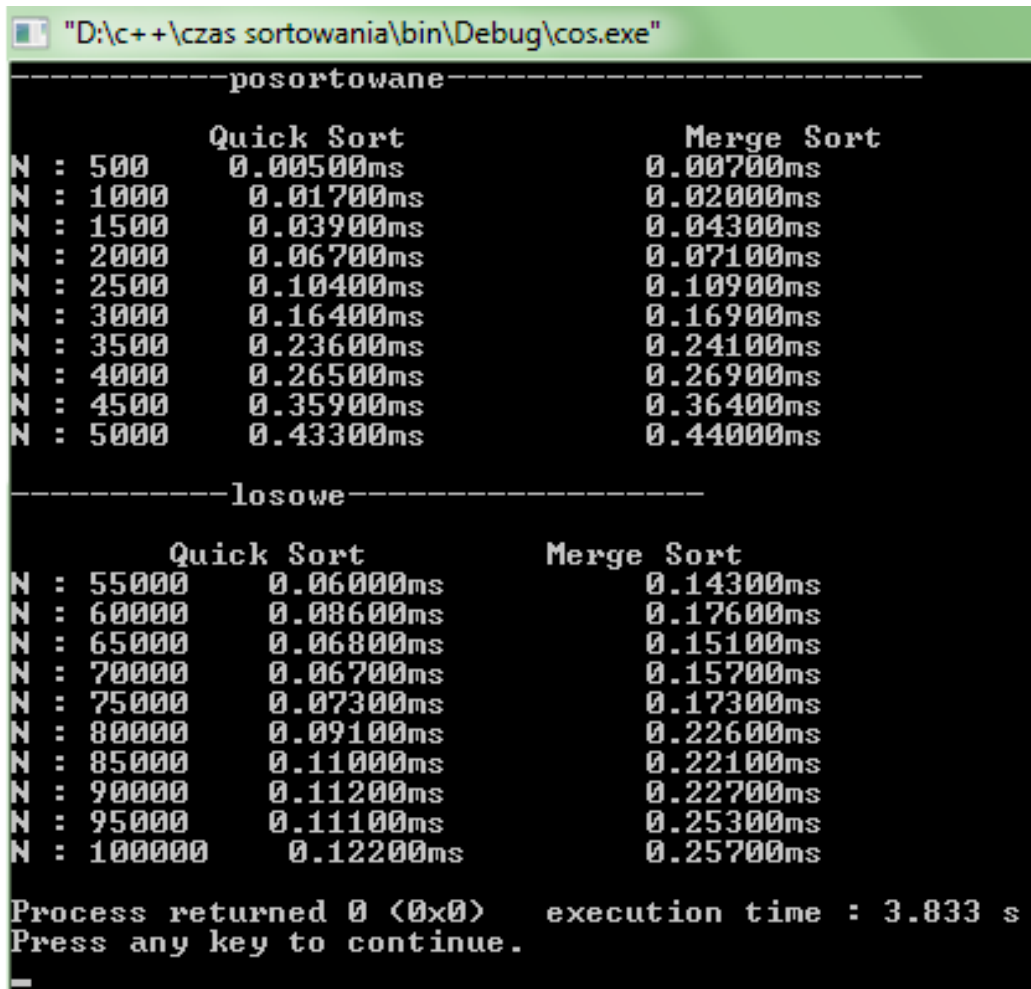
$$T(n) = O(n \log n);$$

- przypadek pesymistyczny: (dla zbiorów posortowanych odwrotnie)

$$T(n) = O(n \log n);$$

5. Złożoność obliczeniowa algorytmów

5.1 Czasy sortowania:



```
"D:\c++\czas sortowania\bin\Debug\cos.exe"

-----posortowane-----

                Quick Sort                Merge Sort
N : 500          0.00500ms                0.00700ms
N : 1000         0.01700ms                0.02000ms
N : 1500         0.03900ms                0.04300ms
N : 2000         0.06700ms                0.07100ms
N : 2500         0.10400ms                0.10900ms
N : 3000         0.16400ms                0.16900ms
N : 3500         0.23600ms                0.24100ms
N : 4000         0.26500ms                0.26900ms
N : 4500         0.35900ms                0.36400ms
N : 5000         0.43300ms                0.44000ms

-----losowe-----

                Quick Sort                Merge Sort
N : 55000        0.06000ms                0.14300ms
N : 60000        0.08600ms                0.17600ms
N : 65000        0.06800ms                0.15100ms
N : 70000        0.06700ms                0.15700ms
N : 75000        0.07300ms                0.17300ms
N : 80000        0.09100ms                0.22600ms
N : 85000        0.11000ms                0.22100ms
N : 90000        0.11200ms                0.22700ms
N : 95000        0.11100ms                0.25300ms
N : 100000       0.12200ms                0.25700ms

Process returned 0 (0x0)    execution time : 3.833 s
Press any key to continue.
```

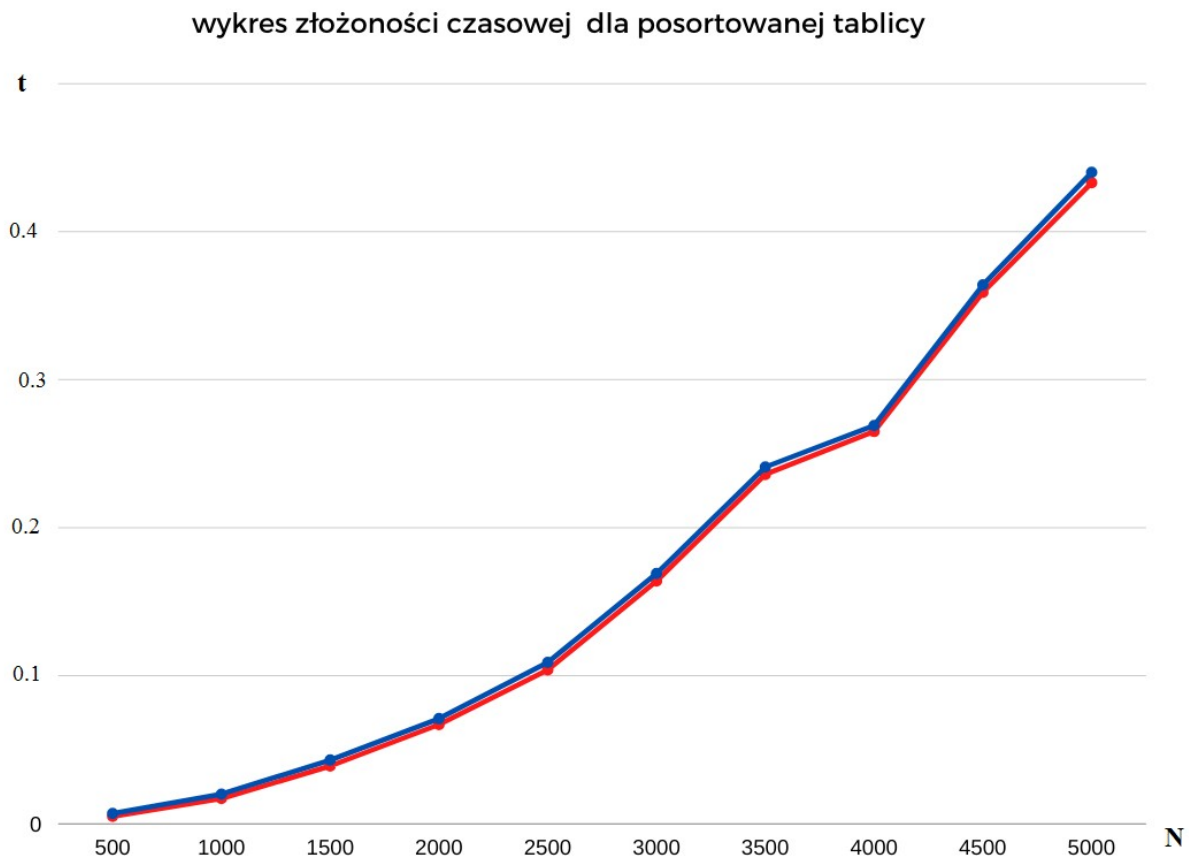
Na zdjęciu zostały pokazane dwa przypadki algorytmów :

- przypadek optymistyczny (posortowane dane wejściowe),
- przypadek przeciętny (losowe dane wejściowe).

Algorytm mojego sortowania nie posiada przypadku pesymistycznego, ponieważ czas sortowania QuickSort jest równy czasowi sortowania uporządkowanej tablicy.

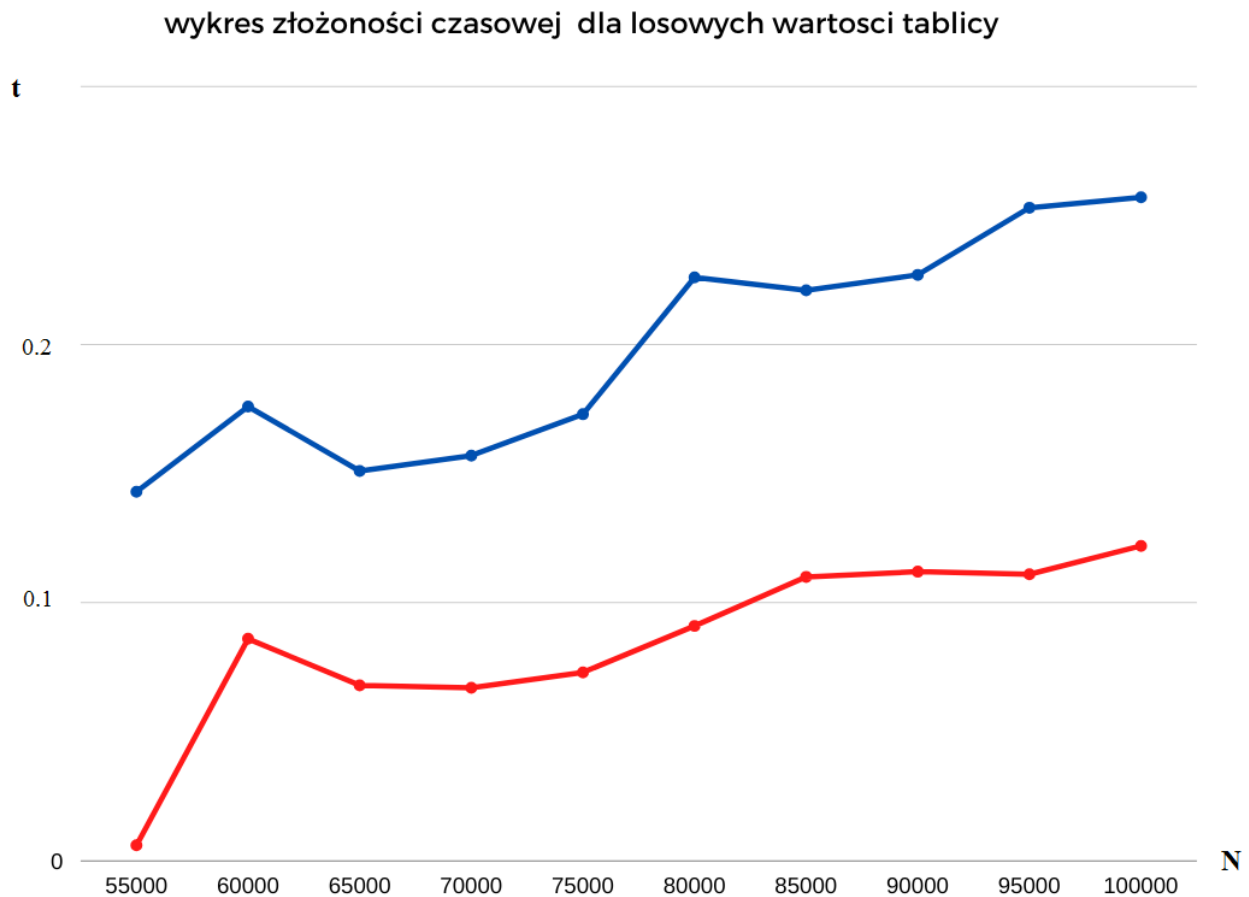
6. Wykresy

6.1 Wykres dla wartości posortowanych oraz N (500, 5000)



Otrzymane wyniki potwierdzają, iż algorytm sortowania szybkiego jest najszybszym algorytmem sortującym w porównaniu z algorytmem sortującym przez scalanie. Jednakże w przypadku ogólnym (posortowane dane wejściowe) notujemy jedynie bardzo nieznaczny wzrost prędkości sortowania w stosunku do algorytmu sortowania przez scalanie.

6.2 Wykres dla wartości losowych oraz N (55000, 10000):



Najdłużej trwa sortowanie zbioru nieuporządkowanego. Jednakże badane czasy(dane z wykresu 1 i 2) nie różnią się wiele pomiędzy sobą, co sugeruje, iż algorytm nie jest specjalnie czuły na rozkład danych wejściowych.