Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

PAMSI

Sprawozdanie nr 2

Sortowanie

Prowadzący:
dr hab. inż. Andrzej Rusiecki

Wykonał:
Jakub Kusz

Wrocław, 24 maja 2022r.

Spis treści

1. Cel projektu

Celem projektu jest zapoznanie się z różnymi algorytmami sortującymi i ich złożonością obliczeniową zależną od różnych zestawów danych.

2. Zadanie do wykonania

Dla danych w pliku projekt2_dane.csv należy wykonać eksperymenty z sortowaniem danych względem rankingu filmów. Załączony plik jest okrojoną bazą filmów „IMDb Largest Review Dataset" z kaggle.com. Plik zawiera tylko tytuł oraz ranking. Proszę o wykonanie następujących zadań:

1. Przefiltrowanie danych i usunięcie pustych wpisów w polu ranking (jeśli występują). Proszę zmierzyć i podać w sprawozdaniu czas przeszukiwania. Czy był on zgodny z oczekiwaną złożonością przeszukiwania dla wybranej struktury danych?
2. Przygotować strukturę danych zawierającą odpowiednio: 10 000, 100 000, 500 000, 1 000 000, maksymalną ilość danych z pliku.
3. Przeprowadzić analizę efektywności sortowania na danych z §2 z wykorzystaniem zaimplementowanych algorytmów.
4. Dodatkowo dla każdego zestawu danych proszę podać w tabeli czas sortowania, średnią wartość oraz medianę rankingu.

Zostały przeze mnie wybrane 4 algorytmy sortujące, które poddałem testom:

1. **Quicksort**,
2. **Mergesort**,
3. **Introsort**.

Kod źródłowy programu znajduje się **Pod tym adresem**

3. Filtrowanie danych

Pierwszym krokiem było przefiltrowanie danych z pliku .csv zawierającego tytułu i rankingi. Należało usunąć wpisy, które nie zawierały rankingu. Filtrowanie odbywało się jednocześnie z pobieraniem poszczególnych linii z pliku .csv - jeśli dana linia nie zawierała pola z rankingiem to nie zostawała zapisana do struktury danych.

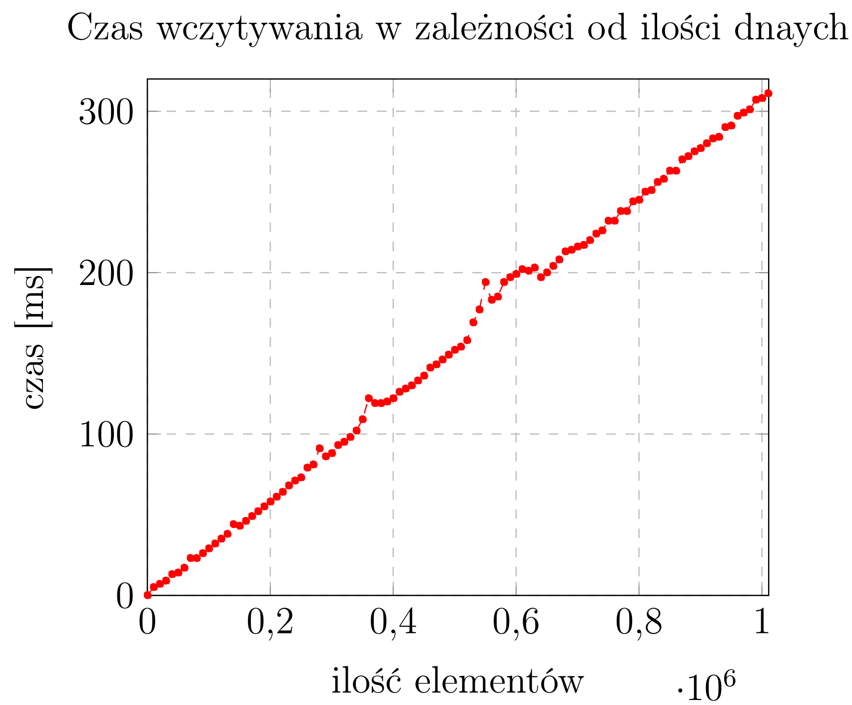
3.1. Przewidywana złożoność obliczeniowa

Przewiduje sie złożoność obliczeniową liniową. Złożoność dodania na koniec wykorzystanego w zadaniu kontenera std :: vector jest stała w czasie, sprawdzenie czy dana linia zawiera pole z rankingiem również, wiec czas przefiltrowania danych zależny jest tylko od ich ilości, więc powinien być liniowy. W notacji dużego O:

$$O(n)$$

3.2. Wyznaczona złożoność obliczeniowa

W celu wyznaczenia złożoności obliczeniowej został przeprowadzony test, polegający na mierzeniu czasu wczytywania liczby kolejnych krotności 1000 linii z pliku .csv.



Wykres. 1: Przedstawiający czas filtrowania kolejnych ilości danych

4. Quicksort

Algorytm wykorzystuje technikę "dziel i zwyciężaj". Według ustalonego schematu wybierany jest jeden element w sortowanej tablicy, który będziemy nazywać pivot. Pivot może być elementem środkowym, pierwszym, ostatnim, losowym lub wybranym według jakiegoś innego schematu dostosowanego do zbioru danych. Następnie ustawiamy elementy nie większe na lewo tej wartości, natomiast nie mniejsze na prawo. W ten sposób powstaną nam dwie części tablicy (niekoniecznie równe), gdzie w pierwszej części znajdują się elementy nie większe od drugiej. Następnie każdą z tych podtablic sortujemy osobno według tego samego schematu.

4.1. Złożoność obliczenia

W zależności od rozkładu danych i elementu pivot określa się następujące złożoności obliczeniowe:

4.1.1. Przypadek optymistyczny

W przypadku optymistycznym, jeśli mamy szczęście za każdym razem wybrać medianę z sortowanego fragmentu tablicy, to liczba porównań niezbędnych do uporządkowania n-elementowej tablicy opisana jest rekurencyjnym wzorem:

$$O(n \cdot log_2 n)$$

4.1.2. Przypadek przeciętny

W przypadku przeciętnym, to jest dla równomiernego rozkładu prawdopodobieństwa wyboru elementu z tablicy:

$$O(1,39 \cdot n \cdot log_2 n)$$

4.1.3. Przypadek pesymistyczny

W przypadku pesymistycznym, jeśli zawsze wybierzemy element najmniejszy (albo największy) w sortowanym fragmencie tablicy, to:

$$O\left(n^2\right)$$

4.2. Implementacja

Poniższy listing przedstawia rekurencyjną implementację algorytmu Quicksort.

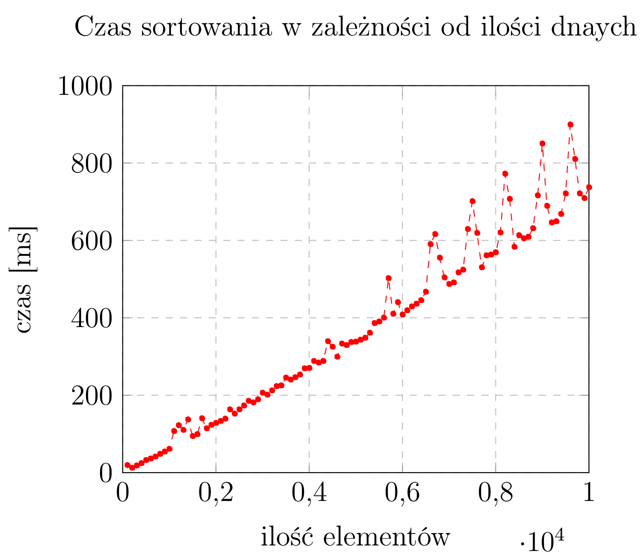
```
void quick_sort(double *tab, int left, int right){
    if(right <= left)
            return;

    int i = left - 1;
    int j = right + 1;
    int pivot = tab[(left+right)/2];

    while(1){

        while(pivot>tab[++i]);
        while(pivot<tab[--j]);


        if( i <= j)
            std::swap(tab[i],tab[j]);
        else
            break;
    }

    if(j > left)
        quick_sort(tab, left, j);
    if(i < right)
        quick_sort(tab, i, right);
}
```

4.3. Testy złożoności obliczeniowej

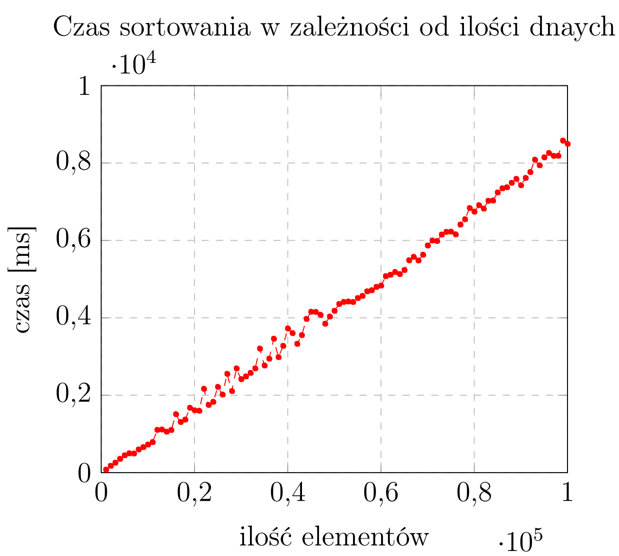
W celu eksperymentalnego wyznaczenia złożoności obliczeniowej algorytmu Quicksort zostały przeprowadzone testy szybkości dla czterech różniących się ilością danych pakietów:

- **10000**,
- **100000**,
- **500000**,
- **962903**.

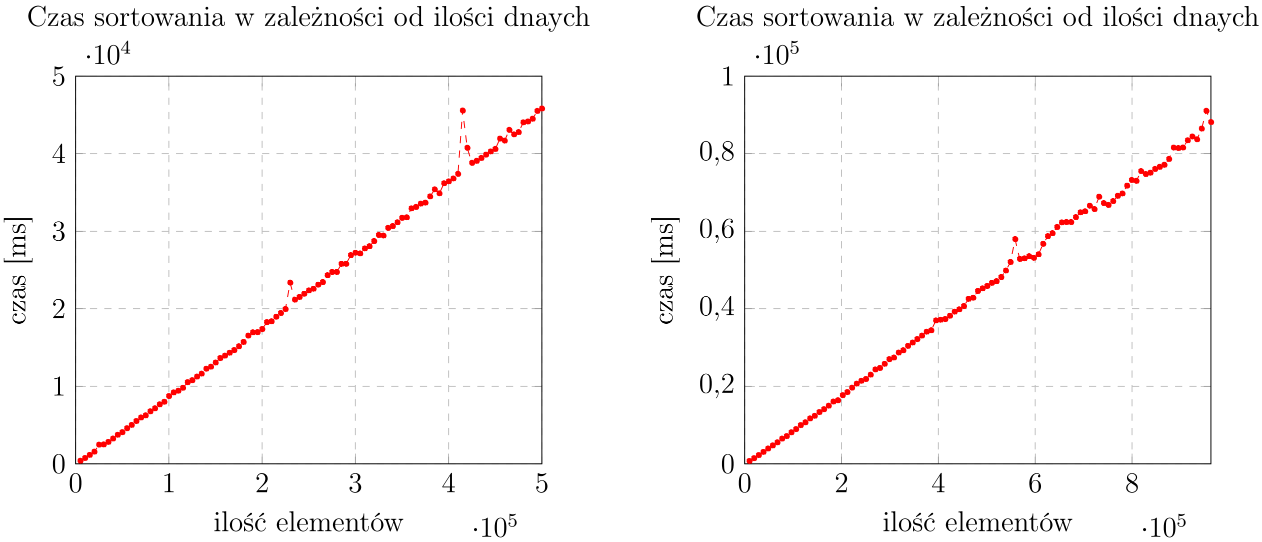
Dla każdego zestawu danych przeprowadzono 100 pomiarów czasu. Najmniejszą paczką dla poszczególnej ilości danych jest $\frac{x}{100}$ gdzie x to ilość danych. Każdy kolejny pomiar pomiar wykonywano dla $n \cdot \frac{x}{100}$ ilości w paczce, gdzie $n \in 1, 2, 3...100$. Na przykład dla **10000** poszczególne paczki to 100, 200, 300... a dla **500000** to 5000, 10000, 15000... .



(a) ilość danych: 10000



(b) ilość danych: 100000



(c) ilość danych: 500000

(d) ilość danych: 962903

Wykres. 2: Wykresy przedstawiające czas sortowania określonej ilości danych

Na Rys. 2 można zauważyć, iż złożoność obliczeniowa prezentuje się jako f. liniowa, co nie jest dużym zaskoczeniem, ponieważ złożoność zarówno optymistyczna(4.1.1) jak i przeciętna(4.1.2) w dużej dziedzinie kształtem również przypominają f. linowe, aczkolwiek nimi nie są.

5. Mergesort

Ideą działania algorytmu jest dzielenie zbioru danych na mniejsze zbiory, aż do uzyskania n zbiorów jednoelementowych, które same z siebie są posortowane, następnie zbiory te są łączone w coraz większe zbiory posortowane, aż do uzyskania jednego, posortowanego zbioru n-elementowego. Etap dzielenia nie jest skomplikowany, dzielenie następuje bez sprawdzania jakichkolwiek warunków. Z kolei łączenie zbiorów posortowanych wymaga odpowiedniego wybierania poszczególnych elementów z łączonych zbiorów z uwzględnieniem faktu, że wielkość zbioru nie musi być równa (parzysta i nieparzysta ilość elementów), oraz tego, iż wybieranie elementów z poszczególnych zbiorów nie musi następować naprzemiennie, przez co jeden zbiór może osiągąć swój koniec wcześniej niż drugi. Robi sie to w następujący sposób. Kopiujemy zawartość zbioru głównego do struktury pomocniczej. Następnie, operując wyłącznie na kopii, ustawiamy wskaźniki na początki kolejnych zbiorów i porównujemy wskazywane wartości. Mniejszą wartość wpisujemy do zbioru głównego i przesuwamy odpowiedni wskaźnik o 1 i czynności powtarzamy, aż do momentu, gdy jeden ze wskaźników osiągnie koniec zbioru. Wówczam mamy do rozpatrzenia dwa przypadki, gdy zbiór 1 osiągnął koniec i gdy zbiór 2 osiągnął koniec. W przypadku pierwszym nie będzie problemu, elementy w zbiorze głównym są już posortowane i ułożone na właściwych miejscach. W przypadku drugim trzeba skopiować pozostałe elementy zbioru pierwszego pokolei na koniec. Po zakończeniu wszystkich operacji otrzmujemy posortowany zbiór główny.

5.1. Złożoność obliczeniowa

Złożoność obliczeniowa jest niezależna od zestawu danych:

$$O(n \cdot log_2 n)$$

5.2. Implementacja

Poniższe listingi przedstawiają funkcje funkcję scalającą i funkcje sortującą:

5.2.1. Scalanie

```
void merge(double array [], int const left , int const mid, int const right ){
    int const sub_array_one = mid - left + 1;
    int const sub_array_two = right - mid;
    double *left_array = new double[sub_array_one];
    double *right_array = new double[sub_array_two];
    for ( int i = 0; i < sub_array_one; i++)
        left_array [ i ] = array[ left + i ];
    for ( int j = 0; j < sub_array_two; j++)
         right_array[ j ] = array[mid + 1 + j];
    int index_of_sub_array_one = 0;
    int index_of_sub_array_two = 0;
    int index_of_merged_array = left;
    while (index_of_sub_array_one < sub_array_one && index_of_sub_array_two < sub_array_two) {
        if (left_array [index_of_sub_array_one] <= right_array[index_of_sub_array_two]) {
            array[index_of_merged_array] = left_array[index_of_sub_array_one];
            index_of_sub_array_one++;
        }
        else {
            array[index_of_merged_array] = right_array[index_of_sub_array_two];
            index_of_sub_array_two++;
        }
        index_of_merged_array++;
    }
     while (index_of_sub_array_one < sub_array_one) {
        array[index_of_merged_array] = left_array[index_of_sub_array_one];
        index_of_sub_array_one++;
        index_of_merged_array++;
    }
    while (index_of_sub_array_two < sub_array_two) {
        array[index_of_merged_array] = right_array[index_of_sub_array_two];
        index_of_sub_array_two++;
        index_of_merged_array++;
    }
}
```

5.2.2. Sortowanie

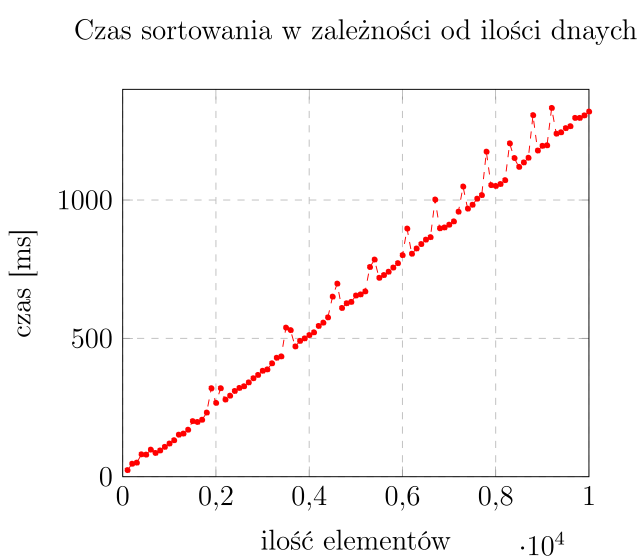
```
void merge_sort(double *arr, int left , int right ){
    if ( left >= right)
        return;

    auto mid = left + (right - left ) / 2;
    merge_sort(arr, left , mid);
    merge_sort(arr, mid + 1, right);
    merge(arr, left , mid, right );
}
```

5.3. Testy złożoności obliczeniowej

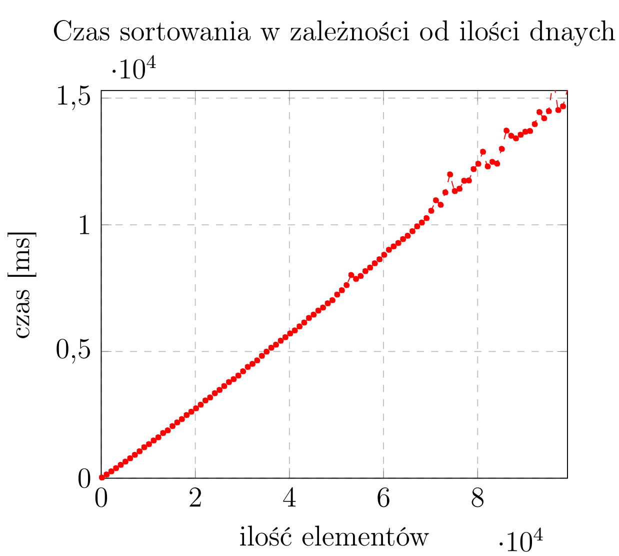
W celu eksperymentalnego wyznaczenia złożoności obliczeniowej algorytmu Mergesort zostały przeprowadzone testy szybkości dla czterech różniących się ilością danych pakietów:

- **10000**,
- **100000**,
- **500000**,
- **962903**.

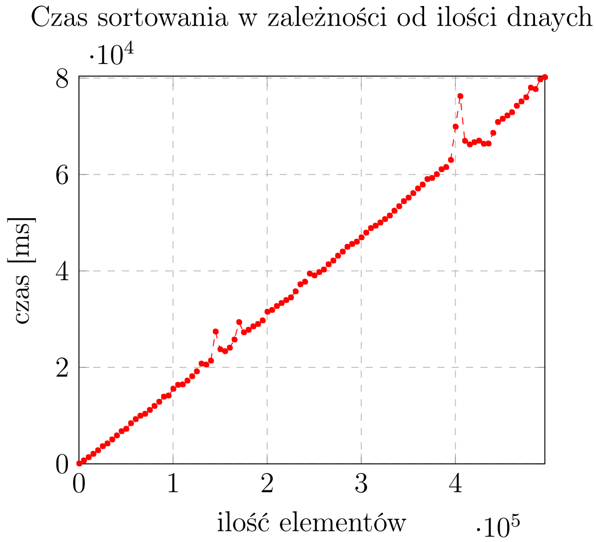
Dla każdego zestawu danych przeprowadzono 100 pomiarów czasu. Najmniejszą paczką dla poszczególnej ilości danych jest $\frac{x}{100}$ gdzie x to ilość danych. Każdy kolejny pomiar pomiar wykonywano dla $n \cdot \frac{x}{100}$ ilości w paczce, gdzie $n \in 1, 2, 3...100$. Na przykład dla **10000** poszczególne paczki to 100, 200, 300... a dla **500000** to 5000, 10000, 15000... .



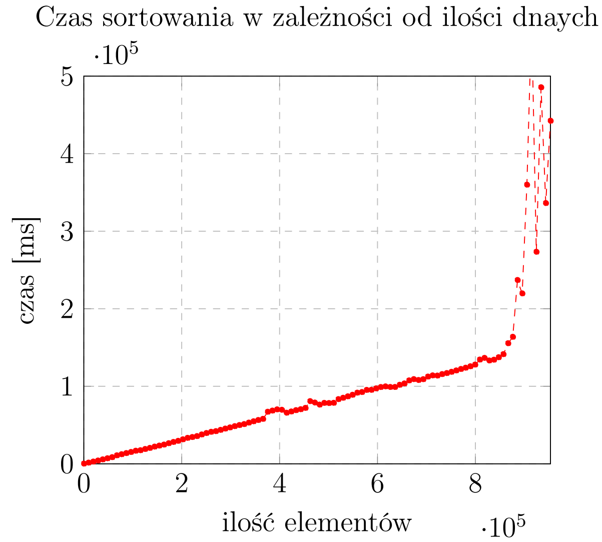
(a) ilość danych: 10000



(b) ilość danych: 100000



(c) ilość danych: 500000



(d) ilość danych: 962903

Wykres. 3: Wykresy przedstawiające czas sortowania określonej ilości danych

6. Introsort

Sortowanie introspektywne to odmiana sortowania hybrydowego, w której wyeliminowany został problem złożoności $O(n^2)$ występującej w najgorszym przypadku algorytmu sortowania szybkiego. Algorytm łączy w sobie 3 rodzaje sortowań, które wywoływane są dla różnych przypadków. Używa on sortowania przez wstawianie (insertion sort), sortowania przez kopcowanie (heap sort) i sortowania szybkiego (quick sort).

6.1. Złożoność obliczeniowa

6.1.1. Przypadek optymistyczny

$$O(n \cdot \log_2 n)$$

6.1.2. Przypadek przeciętny

$$O(1.39 \cdot n \cdot log_2 n)$$

6.1.3. Przypadek pesymistyczny

$$O(2 \cdot n \cdot log_2 n)$$

6.2. Implementacja

Poniższe listingi przedstawiają poszczególne funkcje konieczne do wykonania Introsort

6.2.1. Podział

```
int partition (double *data, int left , int right ) {
        int pivot = data[right ];
        int temp;
        int i = left ;
        for ( int j = left ; j < right; ++j){
            if (data[j ] <= pivot){
                temp = data[j];
                data[j ] = data[i ];
                data[i ] = temp;
                i++;
            }
        }

         data[right ] = data[i ];
        data[i ] = pivot;

        return i;
    }

```

6.2.2. Sortowanie

```
void intro_sort(double *arr , int size ) {
        int partitionSize = partition(arr , 0, size - 1);
            if ( partitionSize < 16){
                insertion_sort(arr , size );
            }
            else if ( partitionSize >(2 * std :: log( size ))){
                heap_sort(arr, size );
            }
            else{
                quick_sort(arr, 0, size -1);
            }
    }

```

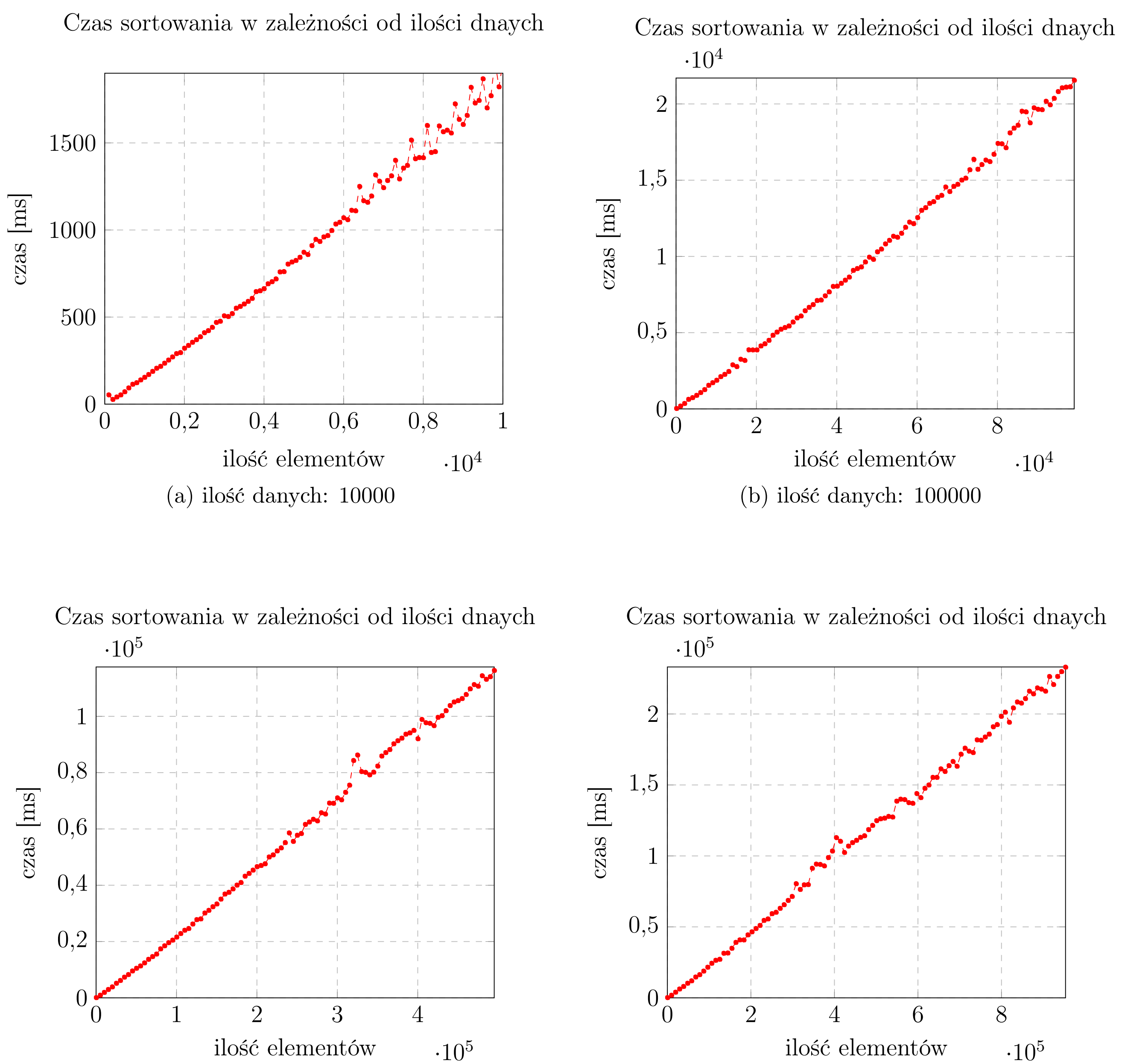
6.3. Testy złożoności obliczeniowej

W celu eksperymentalnego wyznaczenia złożoności obliczeniowej algorytmu Introsort zostały przeprowadzone testy szybkości dla czterech różniących się ilością danych pakietów:

- **10000**,

- **100000**,
- **500000**,
- **962903**.

Dla każdego zestawu danych przeprowadzono 100 pomiarów czasu. Najmniejszą paczką dla poszczególnej ilości danych jest $\frac{x}{100}$ gdzie x to ilość danych. Każdy kolejny pomiar pomiar wykonywano dla $n \cdot \frac{x}{100}$ ilości w paczce, gdzie $n \in 1, 2, 3...100$. Na przykład dla **10000** poszczególne paczki to 100, 200, 300... a dla **500000** to 5000, 10000, 15000... .



(a) ilość danych: 10000

(b) ilość danych: 100000

(c) ilość danych: 500000

(d) ilość danych: 962903

Wykres. 4: Wykresy przedstawiające czas sortowania określonej ilości danych

7. Dyskusja

7.1. Czas trwania filtracji

Wyznaczona doświadczalnie złożoność obliczeniowa filtrowania danych (Wykres 1) jest zgodna ze złożonością prognozowaną (pkt. 3).

Filtrowanie danych trwało ∼: **311 ms**

7.2. Sortowanie

7.2.1. Quicksort

Analizując Wykres 2 można dojść do wniosku, iż złożoność obliczeniowa algorytmu Quicksort jest liniowa, co nie zgadza się z optymistyczną przeciętną jak i pesymistyczną złożonością. (pkt. 4). Można jednak taki rezultat uznać za właściwy, ponieważ przebieg funkcji $n \cdot log_2 n$ jest kształtem zbliżony do funkcji liniowej.

Sortowanie największego zestawu danych (962903 elementów) trwało ∼: **88,101 ms**

7.2.2. Mergesort

Analizując Wykres 3 można dojść do wniosku, iż złożoność obliczeniowa algorytmu Mergesort jest liniowa, co nie zgadza się z teoretyczną złożonością. (pkt. 5). Można jednak taki rezultat uznać za właściwy, ponieważ przebieg funkcji $n \cdot log_2 n$ jest kształtem zbliżony do funkcji liniowej. Dla największego zestawu powyżej $9 \cdot 10^5$ ilości danych czas wykonywania sortowania gwałtownie wzrasta, przez co finalny czas sortowania znacznie się wydłużył.

Sortowanie największego zestawu danych (962903 elementów) trwało ∼: **442,203 ms**

7.2.3. Introsort

Analizując Wykres 4 można dojść do wniosku, iż złożoność obliczeniowa algorytmu Introsort jest liniowa, co nie zgadza się z optymistyczną, przeciętną jak i pesymistyczną złożonością. (pkt. 6). Można jednak taki rezultat uznać za właściwy, ponieważ przebieg funkcji $n \cdot log_2 n$ jest kształtem zbliżony do funkcji liniowej.

Sortowanie największego zestawu danych (962903 elementów) trwało ∼: **232,851 ms**

8. Podsumowanie

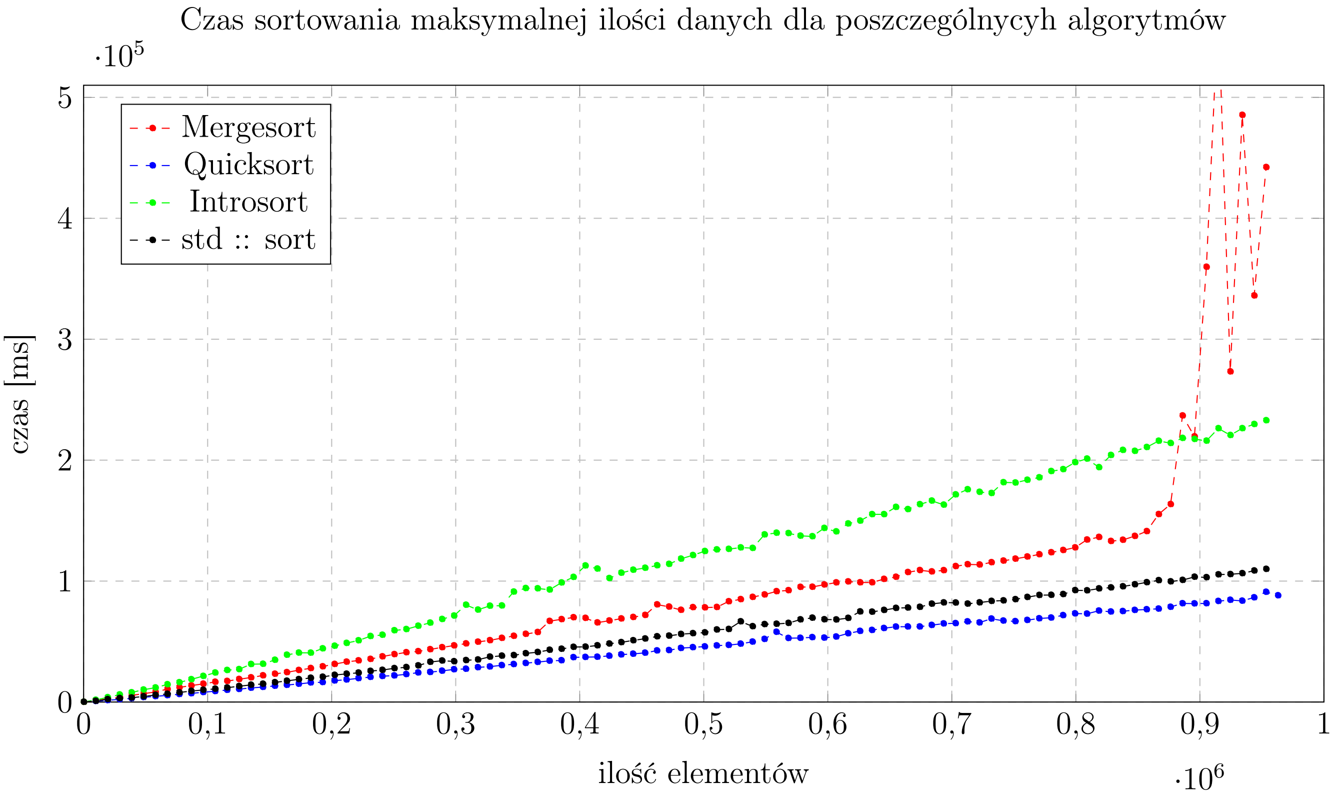
8.1. Czas trwania

Poniższa tabela przedstawia czasy trwania sortowania poszczególnych algorytmów:

Tablica 1: Zawierająca czasy trwania sortowania poszczególnych algorytmów

algorytm	czas trwania [ms]
Quciksort	88,101
Mergesort	442,203
Introsort	232,851
filtrowanie	311

Na poniższym wykresie zostały zaprezentowane charakterystyki złożoności czasowej wszystkich omawianych algorytmów, dodatkowo została zaprezentowana implementacja algorytmu sortującego z biblioteki STL C++ std :: sort(), w celu porównania szybkości gotowego rozwiązania dostępnego dla programisty, z jego własną implementacją algorytmu sortującego.



Wykres. 6: Przedstawiający czas sortowania maksymalnej ilości danych dla poszczególnycyh algorytmów

Uwaga. Algorytm std :: sort() został przedstawiony tylko w celu porównania i zaprezentowania go na wykresie z z napisanymi w ramach zadania algorytmami, nie został on uwzględniony w podsumowaniu i wnioskach, pojawił się jako rodzaj ciekawostki.

8.2. Średnia arytmetyczna i mediana

Dla każdego zestawu danych została obliczona średnia arytmetyczna i mediana rankingu. Zostały one zaprezentowane w poniższej tabeli.

Tablica 2: Zawierająca wartości średniej arytmetycznej i mediany dla poszczególnych zestawów danych

ilość elementów	średnia arytmetyczna	mediana
10000	5.4603	5
100000	6.08993	7
500000	6.66572	7
962903	6.63661	7

8.3. Wnioski

- najszybciej działającym algorytmem sortowania okazał się Quicksort (tab. 1),
- najwolniej działającym algorytmem sortowania okazał się Introsort (tab. 1),
- Mergesort dla danych w ilości większej od ok. $9 \cdot 10^5$ gwałtowanie zwalnia przez co dla dużej ilości danych okazuje się najwolniejszy, nie udało mi się ustalić tego przyczyny (Wykres 6),
- w przypadku algorytmów sortujących złożoności czasowe wyznaczone doświadczalnie nie są zgodne z teoretycznymi, aczkolwiek na tyle podobne, iż wyniki doświadczeń można uznać za satysfakcjonujące.
- mimo, iż najszybciej działającym algorytmem jest Quicksort, może on dla niekorzystnego wyznaczenia pivota okazać się algorytmem o złożoności kwadratowej (pkt. 4), w przeciwieństwie do algorytmów Introsort i Mergesort(pkt. 6,5), które gwarantują taką samą złożoność nie zależnie od zestawu danych, lecz działają wolniej.