Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

PAMSI

Sprawozdanie nr 2

Sortowanie

Prowadzący:
dr hab. inż. Andrzej Rusiecki

Wykonał:
Jakub Kusz

Wrocław, 4 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. **bucketsort**,
4. **introsort**.

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

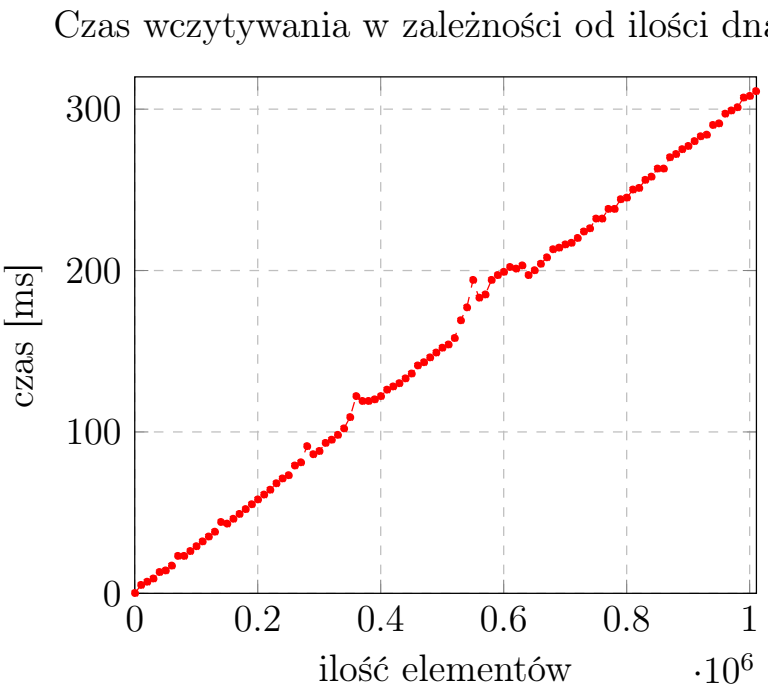
również, wiec czas przefiltrowania danych zależny jest tylko od ich ilości, więc powinien być liniowy.

W notacji dużego O:

$$O(n) = n$$

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.



Rys. 1

Tak jak widać na Rys. 1 złożoność jest liniowa, wiec jednocześnie zgodna z przewidywaną.

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:

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

4.1.2. Przypadek przeciętny

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

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

4.1.3. Przypadek pesymistyczny

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

$$O(n) = \frac{n^2}{2}$$

4.2. Implementacja

Poniższy listing przedstawia rekurencyjną implementację algorytmu Quicksort.

```
1  void quick_sort(double *tab, int left, int right){
2      if(right <= left)
3              return;
4
5      int i = left - 1;
6      int j = right + 1;
7      int pivot = tab[(left+right)/2];
8
9      while(1){
10
11         while(pivot>tab[++i]);
12         while(pivot<tab[--j]);
13
14
15         if( i <= j)
```

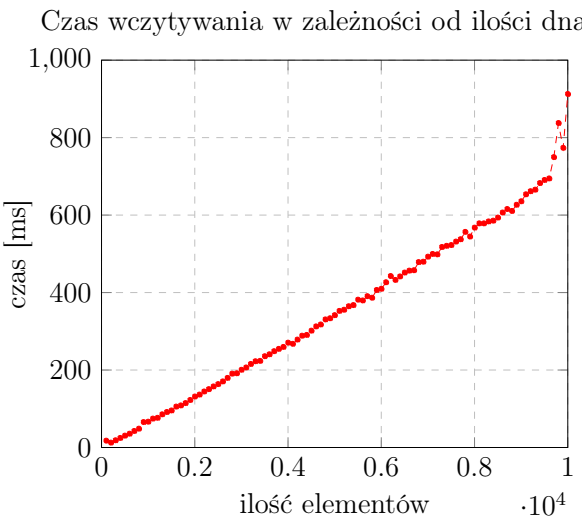
```
16                  std :: swap(tab[i],tab[j]);
17              else
18                  break;
19      }
20
21      if(j > left)
22          quick_sort(tab, left, j);
23      if(i < right)
24          quick_sort(tab, i, right);
25  }
```

4.3. Testy złożoności obliczeniowej

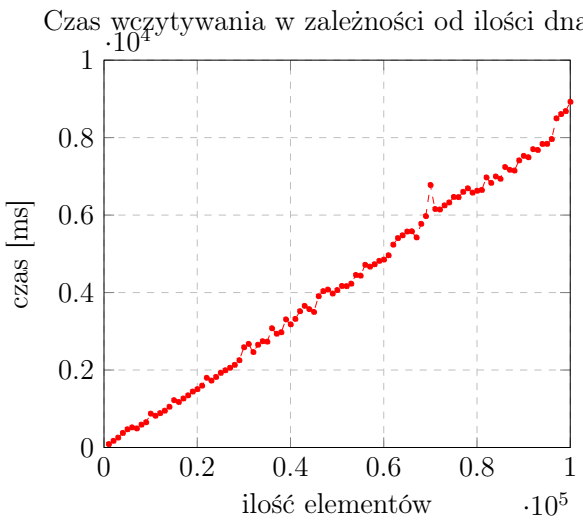
W celu eksperymentalnego wyznaczenia złożoności obliczeniowej algorytmu Quicksort zostały przeprowadzone testy szybkości dla czterech różniących się ilością danych pakietów:

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

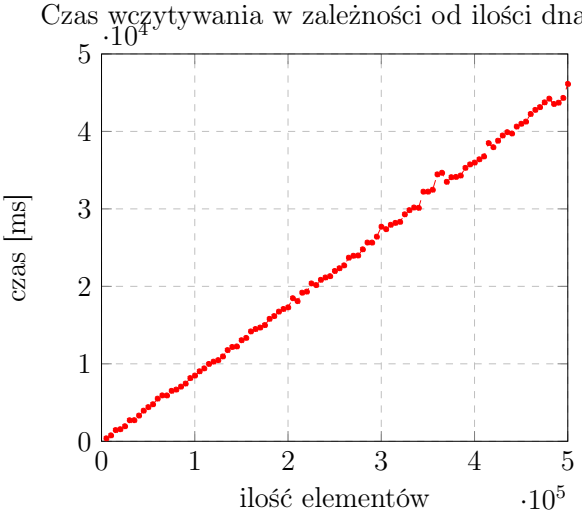
Dla każdego zestawu danych przeprowadzono 100 pomiarów czasu. Najmniejszą paczką dla poszczególnej ilości danych jest $\frac{x}{100}$ gdzie x to ilość danych. Każdy kolejny pomiar pomiar wykonywano dla $n \cdot \frac{x}{100}$ ilości w paczce, gdzie $n \in 1, 2, 3...100$. Na przykład dla **10000** poszczególne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



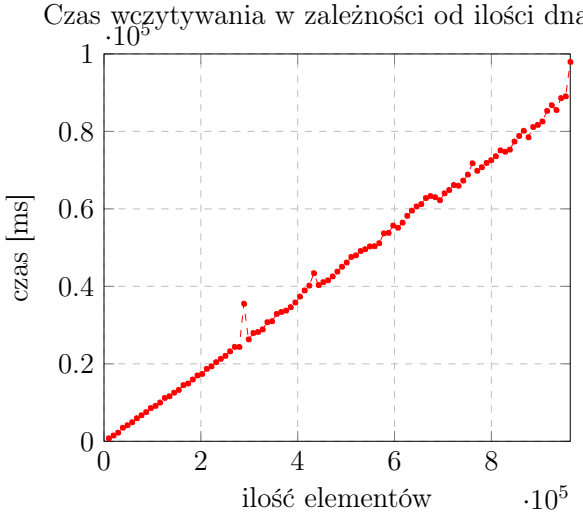
(a) dsda



(b) dsda



(c) dsda



(d) dsda