PAMSI Projekt #2
Algorytmy sortowania

Marcin Poźniak 263441

May 2023

1 Wprowadzenie

Zgodnie z treścią zadania mieliśmy przygotować 3 wybrane algorytmy sortowania , przetestować je i zrecenzować.(Tutaj link do GitHuba) Wybrałem następujące algorytmy:

- Sortowanie przez scalanie (*merge sort*)
- Sortowanie szybkie (*quick sort*)
- Sortowanie przez kopcowanie (*heap sort*)

2 Omówienie poszczególnych algorytmów

2.1 Sortowanie przez scalanie

Jest to rekurencyjny algorytm sortowania danych, stosujący metodę *dziel i zwyciężaj*. Składa się z dwóch fukcji:

- **mergeSort**

  Funkcja ta przyjmuje następujące argumenty:

  - vector obiektów typu *film*
  - lewy graniczny index (skąd zacząć sortowanie)
  - prawy graniczny index (gdzie skończyć sortowanie)

  działa w następujący sposób:

  1. Jeśli lewy index minie się z prawym, wychodzi z rekurencji, bo jest przypadek bazowy
  2. Inaczej oblicza środek tablicy, żeby móc ją następnie podzielić.
  3. Potem następuje rekurencyjne wywołanie funkcji *mergeSort* w celu posortowania dwóch podtablic ; pierwsza to indexy od L do środka, a druga to od środek+1 do P

- **merge**

  Funkcja ta przyjmuje następujące argumenty:

  - vector obiektów typu *film*
  - lewy graniczny index
  - prawy graniczny index
  - środkowy index

  i działa w następujący sposób:

  1. Oblicza rozmiary obu podtablic i alokuje dynamicznie pamięc na nie
  2. Następnie wypełnia podtablice danymi
  3. Teraz w pętli, dopóki indexy pomocnicze nie przekroczą rozmiarów obu podtablic, porównuje pierwszy element z lewej i pierwszy z prawej , większy dodaje do wynikowej tablicy i zwiększa index tablicy z większą wartością i wynikowej.
     Kiedy z którejś z tablic wyczerpią się elementy wykonuję dwię pętle dodające pozstałe elementy do wynikowej tablicy.

2.2 Sortowanie przez kopcowanie

Sortowanie przez kopcowanie (*quicksort*) to także rekurencyjna metoda sortowania, które oparta jest na drzewach binarnych. Jego złożonośc onbliczeniowa w notacji dużego O wynosi **O(nlog(n))**.
Cały algorytm składa się z dwóch głównych kroków:

- Utworzenie drzewa , w którym każde dziecko jest mniejsze od rodzica (max heap) lub mniejsze (min heap) za pomocą operacji *heapify*. Zaczynając na ostatnim bezdzietnym elemencie, porównując najpierw z rodzeństwem, potem większy z nich z rodzicem, rekurencyjnie dochodzimy aż do korzenia (*roota*.
- Zbieranie z drzewa największego elementu , który znajduje się w roocie. Wyłuskanie uzyskujemy przez zamianę roota z ostanim liściem i usunięciem go . Następnie porządkujemy drzewo aż do uzyskania chcianych właściwości.

2.3 Sortowanie szybkie

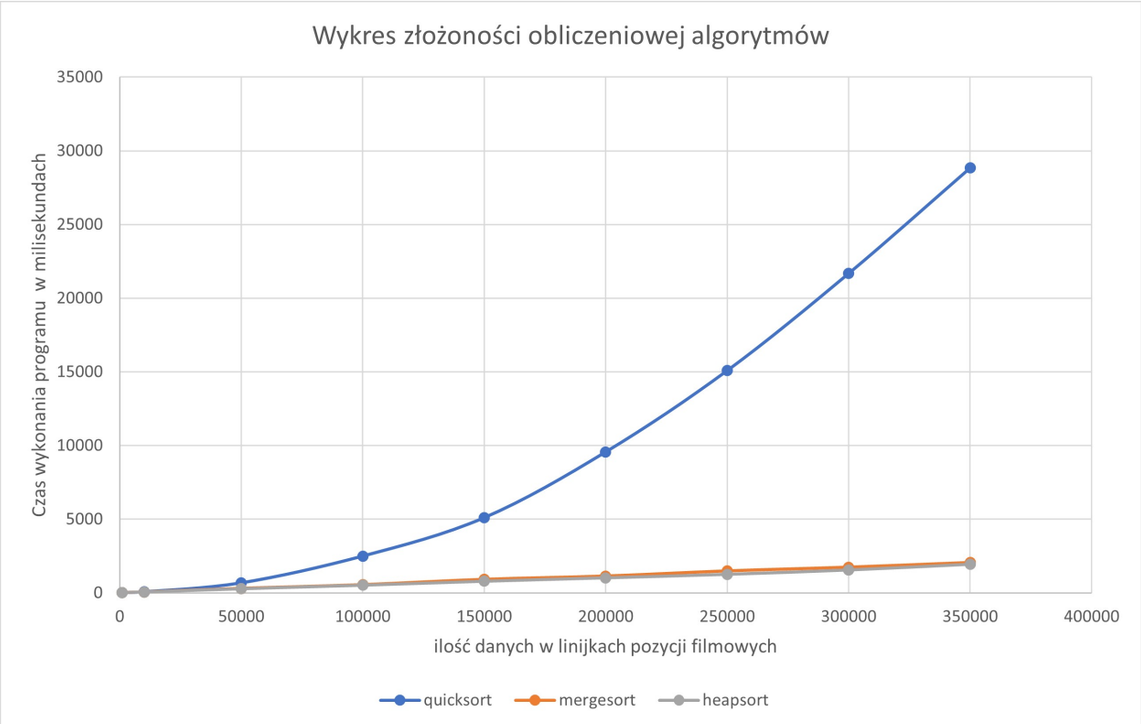
Sortowanie szybkie (*quicksort*) ma złożoność czasową **(O(nlog(n))** ,
a najgorszą $O(n^2)$. Jego idea jest dość prosta:

1. Wybieramy element zwany **pivot** (jest wiele podejść do wyboru piwotu np. pierwszy element, ostatni albo losowy)
2. Następnie dzielimy tak tablicę, aby po lewej stronie od piwota były elementy mniejsze (lub równe) , a po prawej większe.
3. I tak dalej rekursyjnie dzielimy kolejne tablice na dwie podtablice aż pozostaną pojedyncze elementy, następnie po kolei łączymy wszystko w posortowaną całość.

3 Porównanie

Poniżej przedstawiona została zależność czasu wykonywania programu (w milisekundach) od ilości danych wejściowych (w tysiącach pozycji rankingowych w formacie **"nr,tytuł filmu, ocena"**)

Liczba danych (tys)	1	10	50	100	150	200	250	300	350
QuickSort (ms)	11	73	676	2493	5102	9568	15092	21689	28854
MergeSort (ms)	6	61	299	549	914	1135	1486	1738	2056
HeapSort (ms)	16	62	280	524	793	1027	1264	1564	1952



Rysunek 1: Wykres złożoności obliczeniowej algorytmów

4 Wnioski

Jak widać na wykresie coś jest nie tak z quicksortem. Heapsort i mergesort działają nieźle (praktycznie liniowo), ale na pewno możnaby jeszcze je sporo zoptymalizować. Podczas pisania programu wystąpił problemy z GitHubem i musiałem zaczynać od początku , stąd duże opóźnienie. Zaznajomiłem się bardziej z algorytmiką i poznałem nowy sposób myślenia o problemach i ich rozwiązaniach.