Politechnika Wrocławska

Projektowanie i Analiza Algorytmów

Projekt 1

Mateusz Marko ISA nr 273168

2.04.2024

1. Wprowadzenie

• Cel zadania

Zadanie polegało na wybraniu trzech algorytmów sortowania i zaimplementowaniu ich w celu analizy ich efektywności na specyficznym zbiorze danych. Wybrane przeze mnie algorytmy to sortowanie przez scalanie (merge sort), quicksort i sortowanie kubełkowe (bucket sort).

• Dane

Dane pochodziły z uproszczonej bazy filmów IMDb, zawierającej tytuły filmów i ich oceny. Strukturę którą wybrałem do przechowywania danych była lista krotek z oczekiwaną złożonością obliczeniową O(n). Wydawał się to dobry wybór gdy mamy do czynienia ze strukturą zestawów wartości, gdzie każdy element listy (krotka) może przechowywać wiele powiązanych ze sobą danych (np. indeks, tytuł filmu, ocena). Jest ona wydajna dla operacji odczytu i umożliwia łatwe grupowanie, co jest szczególnie ważne podczas przeszukiwania i sortowania danych.

❖ Złożoność obliczeniowa

Złożoność obliczeniowa to termin używany do opisania ilości zasobów, takich jak czas wykonywania (czasowa złożoność) lub pamięć (złożoność pamięciowa), które są potrzebne do rozwiązania problemu algorytmicznie w zależności od rozmiaru danych wejściowych.

• Filtracja danych

Przed sprawdzaniem efektywności należało przefiltrować dane i usunąć wpisy bez oceny. Czas przeszukiwania wierszy bez ratingu, mierzony podczas procesu filtrowania, wyniósł około 1.5431 sekund i był zgodny z oczekiwaną złożonością obliczeniową O(n) dla operacji na listach. Złożoność ta wynikała także z konieczności przeglądania każdego elementu listy w celu sprawdzenia, czy posiada ocenę. Na tak już przygotowanych danych mogliśmy wykonać dalsze części zadania i sprawdzić efektywność algorytmów dla ówcześnie przygotowanych struktur danych kolejno 10.000, 100.000, 500.000, 1.000.000 oraz na wszystkich danych (jednak jak się później okaże po filtracji danych zamiast 1.010.293 elementów będziemy mieli jedynie 962.903 więc nie otrzymamy struktury milionowej i tyle będzie wynosił co nasz max).

• Funkcja sprawdzająca

Jednak zanim przystąpiłem do zadania Aby zweryfikować poprawność sortowania, zaimplementowałem także prostą funkcję pomocniczą, która sprawdzała, czy lista jest posortowana rosnąco według ocen. Dla małych zbiorów danych weryfikacja mogła być przeprowadzona wizualnie, jednak dla większych zbiorów funkcji była potrzebna.

2. Sortowanie przez Scalanie

- Wstęp teoretyczny

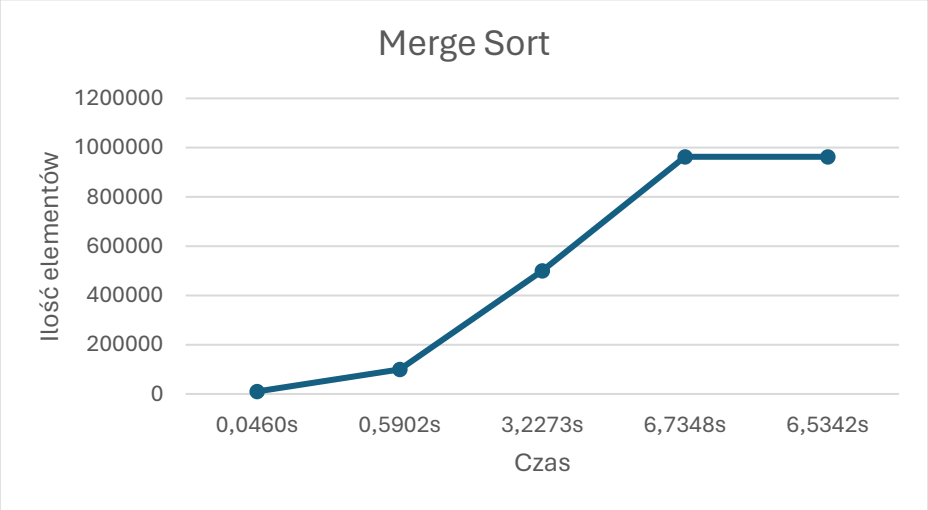
Sortowanie przez scalanie to algorytm typu "dziel i zwyciężaj", który dzieli listę na coraz mniejsze fragmenty, aż do uzyskania list jednoelementowych, a następnie scala te fragmenty w sposób uporządkowany. Proces scalania zakłada, że mniejsze listy są już posortowane, co umożliwia ich efektywne łączenie w większe posortowane listy.

```
def merge_sort(lista):
    if len(lista) <= 1:
        return lista

    srodek = len(lista) // 2
    lewa_polowa = merge_sort(lista[:srodek])
    prawa_polowa = merge_sort(lista[srodek:])

    return merge(lewa_polowa, prawa_polowa)
```

Rysunek 1 fragment kodu sortowania przez Scalanie

- Wyniki

Złożoność obliczeniowa sortowania przez scalanie jest taka sama dla przypadków średniego i najgorszego (O(n log n)), ponieważ algorytm zawsze dzieli listę na połowy, a następnie wykonuje operacje scalania, które są proporcjonalne do logarytmu liczby podziałów i liniowo zależne od rozmiaru listy.



Rysunek 2 Wykres zależności czasu od ilości elementów w zbiorze dla sortowania przez Scalanie

Widać wyraźny wzrost czasu sortowania wraz ze wzrostem liczby elementów. Algorytm jest stabilny, ale stosunkowo wolniejszy w porównaniu do quicksorta i sortowania kubełkowego, co może wynikać z dodatkowej pamięci wymaganej do przechowywania podziaelonych danych oraz z potrzeby ciągłego dzielenia i łączenia list.

Tabela 1 Statystyki dla sortowania przez Scalanie

Ilość danych	10.000	100.000	500.000	1.000.000	max
Czas sortowania	0.0460s	0.5902s	3.2273s	6.7348s	6.5342s
Średnia wartość	5.46	6.09	6.67	6.64	6.64
Mediana rankingu	5.0	7.0	7.0	7.0	7.0

3. Sortowanie Quicksort

- Wstęp teoretyczny

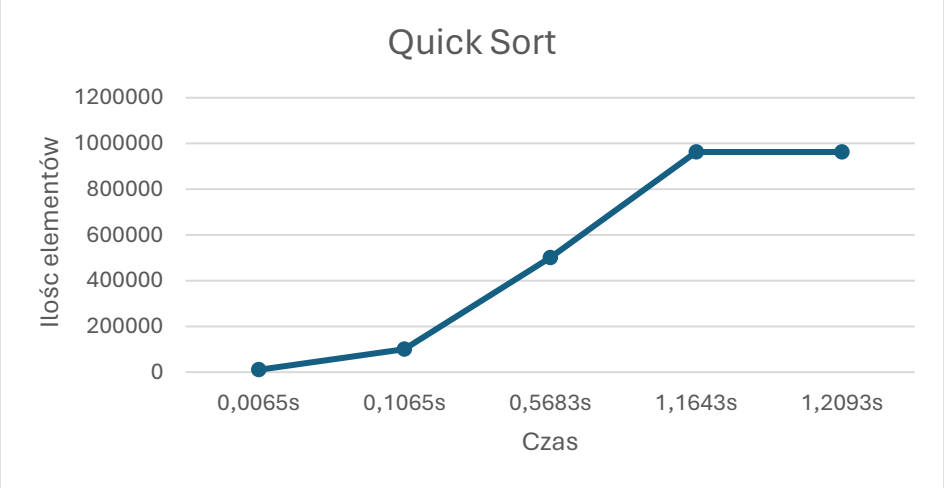
Quicksort to również algorytm typu "dziel i zwyciężaj", który wybiera element zwanym pivotem, a następnie porządkuje listę tak, aby wszystkie elementy mniejsze od pivota znalazły się przed nim, a większe po. Proces jest powtarzany rekursywnie dla list utworzonych z elementów mniejszych i większych niż pivot.

```python
def quicksort(lista):
    if len(lista) <= 1:
        return lista

    pivot = lista[len(lista) // 2][2]  #środkowy element jako pivot
    mniejsze = [x for x in lista if x[2] < pivot]
    rowne = [x for x in lista if x[2] == pivot]
    wieksze = [x for x in lista if x[2] > pivot]

    return quicksort(mniejsze) + rowne + quicksort(wieksze)
```

Rysunek 3 fragment kodu sortowania Quicksort

- Wyniki

Średnia złożoność obliczeniowa quicksort (O(n log n)) wynika z założenia, że pivoty są dobrze wybierane, dzieląc listę na mniej więcej równe części, co prowadzi do zbalansowanego drzewa rekursji. W najgorszym przypadku ($O(n^2)$), gdyby lista była już posortowana lub pivot zawsze dzielił listę na bardzo nierównomierne części, głębokość rekursji może wzrosnąć do n, co prowadzi do kwadratowej złożoności.



Rysunek 4 Wykres zależności czasu od ilości elementów w zbiorze dla sortowania Quicksort

Wyniki eksperymentów wykazują znacznie krótsze czasy sortowania w porównaniu do sortowania przez Scalanie, co wskazuje na to, że dane były na tyle losowe, że algorytm działał blisko swojego optymalnego przypadku średniego. Należy uważać, gdyż jego wydajność może być zależna od wyboru pivota. Skuteczność może wynikać z efektywnego podziału danych oraz mniejszej ilości operacji na danych w pamięci.

Tabela 2 Statystyki dla sortowania Quicksort

Ilość danych	10.000	100.000	500.000	1.000.000	max
Czas sortowania	0.0065s	0.1065s	0.5683s	1.1643s	1.2093
Średnia wartość	5.46	6.09	6.67	6.64	6.64
Mediana rankingu	5.0	7.0	7.0	7.0	7.0

4. Sortowanie Kubełkowe

- Wstęp teoretyczny

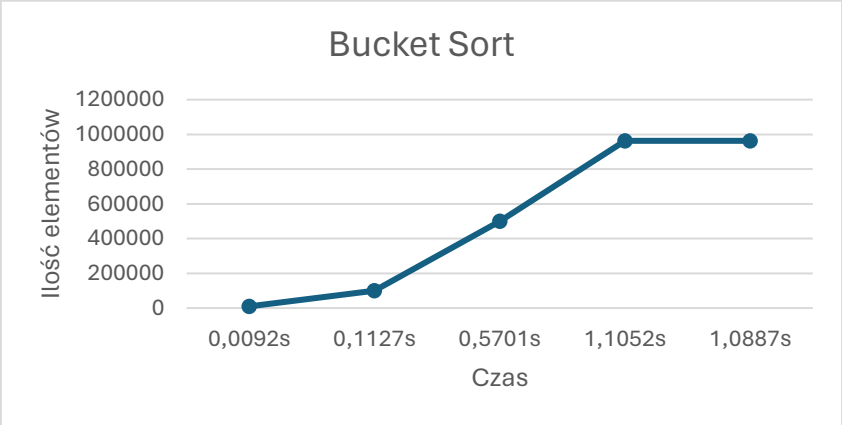
Sortowanie kubełkowe polega na rozdzieleniu danych na ograniczoną liczbę segmentów, zwanych kubełkami, na podstawie przedziałów wartości. Każdy kubełek jest następnie sortowany niezależnie, można użyć np. innego algorytmu sortowania w moim przypadku było to sortowanie przez wstawianie gdyż wydawało się optymalne dla sortowania małych kubełków, a na koniec dane z posortowanych kubełków są łączone w jedną posortowaną listę.

```python
def bucket_sort(lista, liczba_kubelkow=10):
    if len(lista) == 0:
        return lista

    min_value, max_value = min(lista, key=lambda x: x[2])[2], max(lista, key=lambda x: x[2])[2]
    bucket_range = (max_value - min_value) / liczba_kubelkow
    buckets = [[] for _ in range(liczba_kubelkow)]

    for film in lista:
        bucket_index = int((film[2] - min_value) // bucket_range)
        if bucket_index == liczba_kubelkow:
            bucket_index -= 1
        buckets[bucket_index].append(film)

    #sortowanie elementów w każdym kubelku i ich scalanie
    return [film for bucket in buckets for film in insertion_sort(bucket, key=lambda x: x[2])]
```

Rysunek 5 fragment kodu sortowania Kubełkowego

- Wyniki

Średnia złożoność obliczeniowa sortowania kubełkowego (O(n + k), gdzie k to liczba kubełków) zakłada, że dane są równomiernie rozłożone po kubełkach i że liczba kubełków jest odpowiednio dobrana do rozmiaru danych. W najgorszym przypadku ($O(n^2)$), jeśli wszystkie dane trafią do jednego kubełka, złożoność algorytmu sortowania użytego do sortowania danych w kubełku (np. jeśli używamy sortowania przez wstawianie tak jak ja) może prowadzić do kwadratowej złożoności.



Rysunek 6 Wykres zależności czasu od ilości elementów w zbiorze dla sortowania Kubełkowego

Sortowanie to jest idealne do danych równomiernie rozłożonych gdy znamy zakres wartości, co pozwala na wydajne podzielenie danych na kubełki. Wydaje się mieć złożoność bliską O(n), ale wymaga jednak odpowiedniego doboru liczby i rozmiarów kubełków.

Tabela 3 Statystyki dla sortowania Kubełkowego

Ilość danych	10.000	100.000	500.000	1.000.000	max
Czas sortowania	0.0092s	0.1127s	0.5701s	1.1052s	1.0887s
Średnia wartość	5.46	6.09	6.67	6.64	6.64
Mediana rankingu	5.0	7.0	7.0	7.0	7.0

5. Wnioski

- Wybór algorytmu sortowania zależy od charakterystyki danych wejściowych i wymagań dotyczących wydajności. Sortowanie przez Scalanie i Quicksort są często wybierane ze względu na ich ogólną efektywność w większości przypadków, podczas gdy sortowanie Kubełkowe może być preferowane dla specyficznych rozkładów danych, które umożliwiają jego efektywne zastosowanie i rozkład na odpowiednie kubełki.

- Zrozumienie złożoności obliczeniowej algorytmów jest fundamentalne dla projektowania efektywnych, szybkich i skalowalnych rozwiązań informatycznych. Pomaga to w ocenie, jak algorytm będzie skalował się z rozmiarem danych i jakie może mieć to konsekwencje dla ogólnej wydajności i kosztów systemu.

- Wykonanie badań na dobrym komputerze, ma znaczenie dla oceny wydajności algorytmów sortowania, szczególnie przy dużych zbiorach danych. Wyższa moc obliczeniowa i większa ilość dostępnej pamięci RAM pozwalają na szybsze przetwarzanie danych i zmniejszają ryzyko wpływu ograniczeń sprzętowych na czas wykonania algorytmów. Dzięki temu możliwe jest uzyskanie bardziej wiarygodnych wyników dotyczących efektywności samych algorytmów, a nie ograniczeń sprzętowych.

- Ilość danych w zbiorze także ma kluczowe znaczenie dla czasu sortowania. Dla wszystkich algorytmów, w miarę zwiększania się rozmiaru danych, obserwujemy znaczme zwiększenie czasu sortowania. Szczególnie pokazują to wyraźnie rosnące czasy sortowania dla sortowania przez Scalanie. Jest to zgodne z założeniem, że złożoność obliczeniowa ma wpływ na czas potrzebny do przetworzenia większej ilości danych.

- Podczas badań napotkałem także małą różnicę czasu sortowania dla zestawu danych 962.903 elementów. Pomimo tej samej wielkości, gdy mieliśmy do czynienia ze zbiorem milionowym czas był inny niż w przypadku szeregowania zbioru maksymalnego. Przy sortowaniu przez Scalanie czas sortowania największego ze zbiorów był paradoksalnie nieco krótszy niż w teorii mniejszego milionowego zbioru, który w rzeczywistości był tego samego rozmiaru (6,7348s / 6,5342s), z kolei dla Quicksorta był on nieco większy (1,1643s / 1,2093s) tak samo dla sortowania Kubełkowego (1,1052s / 1,0887s). Może to wynikać z wielu czynników, w tym z niestabilności środowiska wykonawczego, procesów w tle lub niejednorodności danych. Niewielka różnica może być również w granicach błędu pomiarowego.

6. Bibliografia

- https://informatyka.2ap.pl/ftp/3d/algorytmy/podr%C4%99cznik_algorytmy.pdf
- http://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie
- http://pl.wikipedia.org/wiki/Sortowanie_szybkie
- https://en.wikipedia.org/wiki/Bucket_sort
- https://pl.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort
- https://pl.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms