Projektowanie Algorytmów i Metody Sztucznej Inteligencji	
Kierunek *Automatyka i Robotyka*	**Termin** *Wtorek 13.15*
Imię, nazwisko, numer albumu *Łukasz Walczak 259278*	**Ocena:**
Prowadzący projekt *Dr hab. inż. Andrzej Rusiecki*	**Data oddania sprawozdania** *17 marca 2023*

Projekt 2 - Zadanie na ocenę bdb

Spis treści

Link do repozytorium
`https://github.com/CrassusXY/Pamsi-2`

1 Zadanie do wykonania

Zadanie polegało na zaimplementowaniu oraz analizie trzech algorytmów sortowania. W celu wykonania tejże analizy, zaimplementowano strukturę zawierającą listę filmów. Lista miała zostać posortowana względem oceny danego filmu. W przypadku mojej implementacji założyłem sortowanie, od najwyższej oceny do najniższej. Spośród dostępnych algorytmów, zdecydowałem się na wybranie sortowania przez scalanie, quicksorta oraz sortowania kubełkowego. W celu porównania wyników, zmierzyłem zależność czasu sortowania względem długości listy.

2 Wybrane algorytmy sortowania

2.1 Sortowanie przez scalanie

Sortowana tablica dzielona jest rekurencyjnie na dwie podtablice aż do uzyskania tablic jednoelementowych. Następnie podtablice te są scalane w odpowiedni sposób, dający w rezultacie tablicę posortowaną. Wykorzystana jest tu metoda podziału problemu na mniejsze, łatwiejsze do rozwiązania zadania („dziel i rządź").
Złożoność obliczeniowa:

- Najlepszy przypadek $O(n * logn)$
- Średni przypadek $O(n * logn)$
- Najgorszy przypadek $O(n * logn)$

2.2 QuickSort

Na początku wybierany jest tzw. element osiowy. Następnie tablica dzielona jest na dwie podtablice. Pierwsza z nich zawiera elementy mniejsze od elementu osiowego, druga elementy większe lub równe, element osiowy znajdzie się między nimi. Proces dzielenia powtarzany jest aż do uzyskania tablic jednoelementowych, nie wymagających sortowania. Właściwe sortowanie jest tu jakby ukryte w procesie przygotowania do sortowania. Wybór elementu osiowego wpływa na równomierność podziału na podtablice (najprostszy wariant – wybór pierwszego elementu tablicy – nie sprawdza się w przypadku, gdy tablica jest już prawie uporządkowana).
Złożoność obliczeniowa:

- Najlepszy przypadek $O(n * logn)$
- Średni przypadek $O(n * logn)$
- Najgorszy przypadek $O(n^2 n)$

2.3 Sortowanie kubełkowe

Polega ono na podzieleniu danych na k mniejszych zbiorów, które dzięki temu są już częściowo posortowane. Następnie każdy z tych zbiorów jest sortowany indywidualnie. Ostatecznie w odpowiedniej kolejności, kolejne wiadra są opróżniane dając w ten sposób posortowane dane.
Złożoność obliczeniowa:

- Najlepszy przypadek $O(n + k)$
- Średni przypadek $O(n)$
- Najgorszy przypadek $O(n^2)$

3 Struktury danych

Pisząc program stworzyłem dwie struktury. Pierwsza to klasa movie, która w polu protected posiada dwa pola. Pole rating przechowujące ocenę filmu, oraz string rating, zawierający tytuł. Jest to struktura bazowa, posiadająca settery i gettery dla każdego z pól, oraz przeciążenia operatorów porównania i przypisania, które były mi potrzebne przy operacjach sortowania.
Druga klasa to MovieList, która składa się z listy klas Movie oraz z długości tej listy. Zdecydowałem się na jej implementację, ze względu na chęć zachowania podejścia obiektowego. Podczas testów zrozumiałem, że nie jest to optymalne rozwiązanie zadania, lecz ze względu na ograniczony czas do oddania, byłem zmuszony na pozostawienie tego w takiej formie. Mimo wszystko taka implementacja spełnia swoje założenia. Klasa ta składa się podobnie jak poprzednia z setterów i getterów, tutaj dodatkowo występują funkcje do liczenia średniej i mediany oraz rozbudowany konstruktor w którym następuje wczytanie bazy filmów. W jego wywołaniu musimy podać nazwę pliku i ewentualnie ograniczenie. Najpierw otwiera on strumień do pliku, a następnie sprawdza czy otwarcie się powiodło. Kolejnym krokiem jest przeliczenie ilości dobrych wpisów w bazie oraz błędnych. Zauważyłem, że niektóre wpisy mają brak oceny, więc je odrzucam. Następnie inicjuje tablicę o rozmiarze równym ilości poprawnych wpisów w bazie, którą wypełniam kolejnymi filmami. Jest też opcja wywołania konstruktora z limitem wczytywanych filmów.
Co do implementacji algorytmów sortowania, to starałem się je dość skrupulatnie komentować, aby ułatwić także sobie zrozumienie poszczególnych kroków. Jednak zrozumienie tychże funkcji nie było raczej trudne, zwłaszcza, że w internecie wszystkie z tych algorytmów są dokładnie omówione. W przypadku problemów z działaniem, po prostu można było sobie rozrysować poszczególne kroki w rekurencji, co kilkukrotnie zrobiłem. Największe problemy sprawił mi chyba algorytm quicksort, początkowo próbowałem go zaimplementować zgodnie ze schematem Lomuto, jednak dużo skuteczniejsze okazało się skorzystanie z algorytmu Hoare. Ostatecznie każdy z tych algorytmów działa zgodnie z oczekiwaniami.

4 Wyniki

4.1 Mediana

	mediana		
n	MergeSort	QuickSort	BucketSort
10000	5,0	5,0	5,0
50000	5,5	5,5	5,5
100000	7,0	7,0	7,0
500000	7,0	7,0	7,0
962903	7,0	7,0	7,0

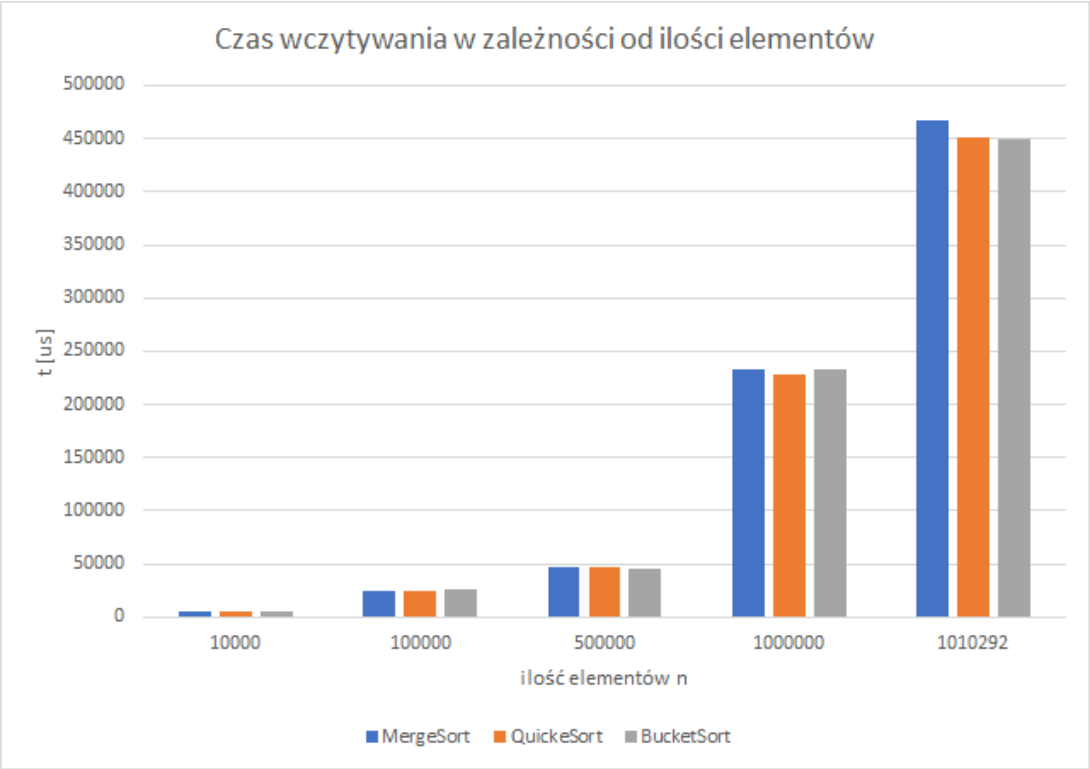
Tabela 1: Wyniki obliczeń miediany w zależności od długości pakietów

4.2 Średnia

	średnia		
n	MergeSort	QuickSort	BucketSort
10000	5,46030	5,46030	5,46030
50000	5,49092	5,49092	5,49092
100000	6,08993	6,08993	6,08993
500000	6,66572	6,66572	6,66572
962903	6,63661	6,63661	6,63661

Tabela 2: Wyniki obliczeń średniej w zależności od długości pakietów

4.3 Wczytywanie

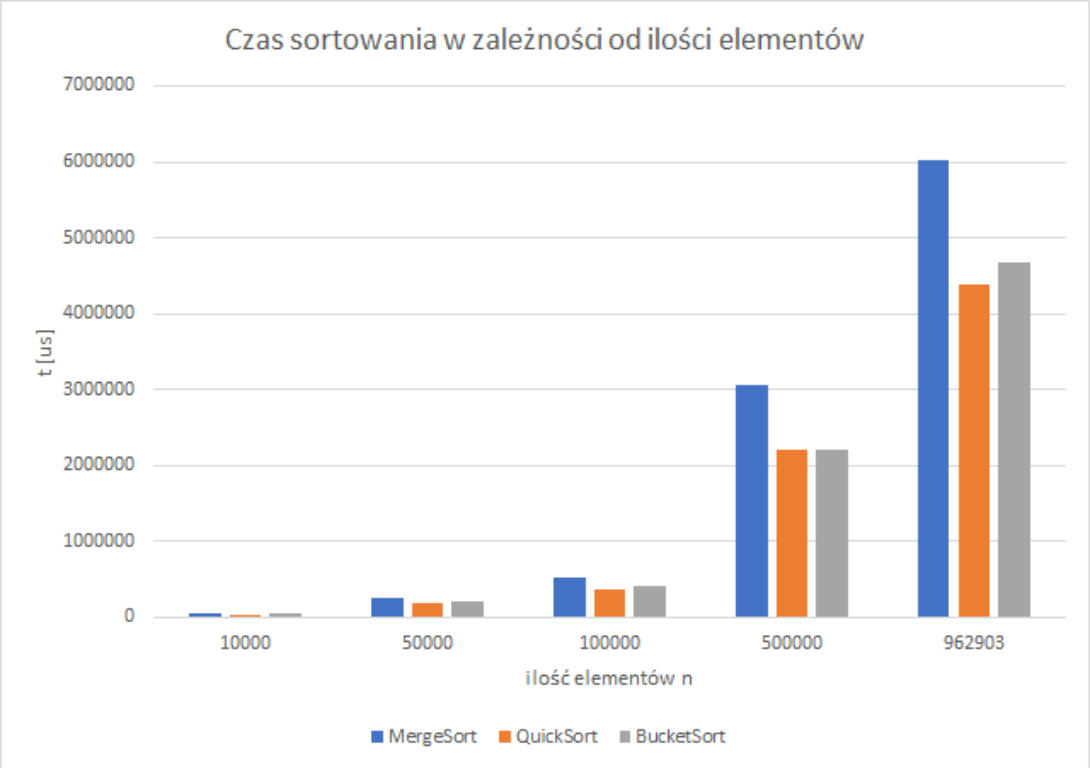


Rysunek 1: Wyniki wczytywania przy różnych rodzajach sortowań, jak widać ich czas nie jest w żaden sposób związany z rodzajem algorytmu sortowania.

	t wczytywania $[\mu s]$		
n	MergeSort	QuickSort	BucketSort
10000	5063	4765	5278
100000	25262	24248	25454
500000	46251	46371	45456
1000000	232737	228837	233728
1010292	467047	450480	449700

Tabela 3: Zestawienie wyników dla zapisu

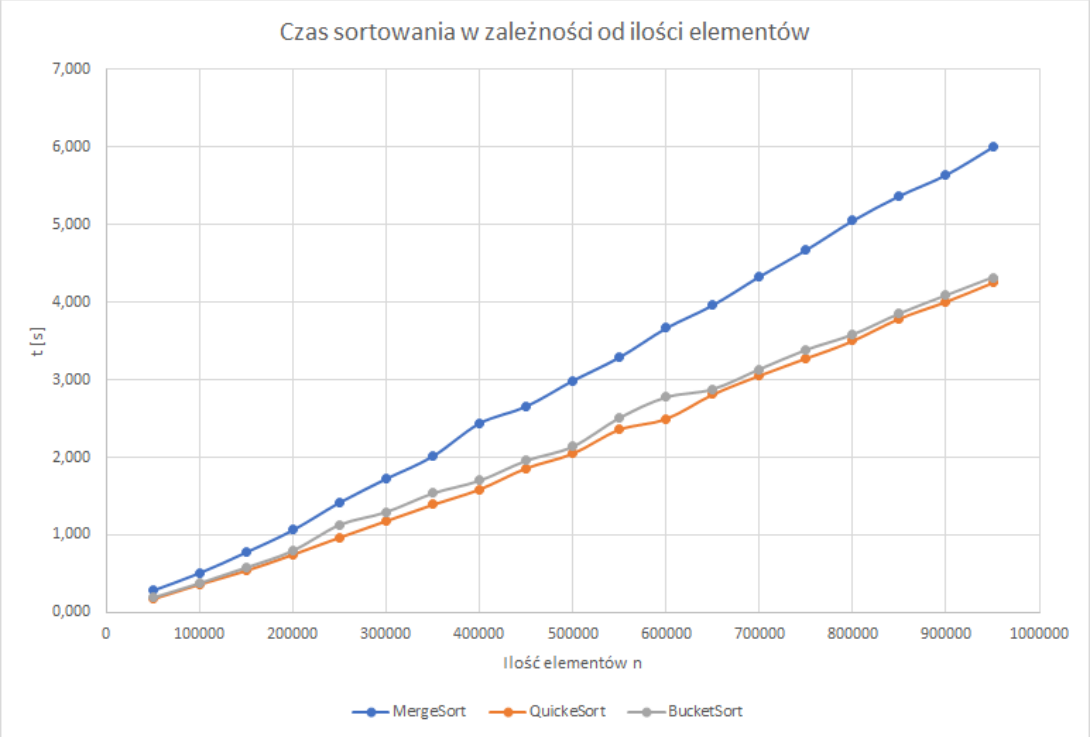
4.4 Sortowanie



Rysunek 2: Wyniki sortowania dla poszczególnych algorytmów.

	t sortowania $[\mu s]$		
n	MergeSort	QuickSort	BucketSort
10000	53854	29915	43043
50000	250216	176494	198219
100000	522568	375998	410397
500000	3058760	2208096	2200773
962903	6032766	4389918	4686689

Tabela 4: Zestawienie wyników dla wczytywania



Rysunek 3: Wykresy sortowania dla poszczególnych algorytmów.

	t sortowania [s]		
n	MergeSort	QuickSort	BucketSort
50000	0,287	0,176	0,191
100000	0,513	0,367	0,378
150000	0,781	0,542	0,580
200000	1,066	0,750	0,796
250000	1,420	0,969	1,130
300000	1,728	1,183	1,295
350000	2,018	1,393	1,536
400000	2,441	1,590	1,700
450000	2,660	1,862	1,956
500000	2,990	2,052	2,139
550000	3,292	2,361	2,508
600000	3,667	2,499	2,774
650000	3,963	2,813	2,878
700000	4,332	3,056	3,135
750000	4,674	3,277	3,386
800000	5,052	3,504	3,583
850000	5,371	3,791	3,854
900000	5,642	4,008	4,089
950000	5,998	4,252	4,320

Tabela 5: Zestawienie wyników dla sortowania

5 Wnioski

- Każdorazowo sortowanie sprawdzałem przy użyciu funkcji sorted, która sprawdzała, czy nie występuje sytuacja, że element o indeksie i+1, będzie większy od elementu o indeksie i. Dzięki tej funkcji mogę stwierdzić, że sortowanie odbywa się poprawnie.
- Dodatkową formą sprawdzenia, było porównanie mediany oraz średniej przy różnych algorytmach sortowaniach tej samej tablicy. Widać to w tabelach oraz . Wyniki nie zależą od użytego algorytmu, więc żaden element nie został "zgubiony" lub dodany podczas sortowania.
- Najszybciej działa algorytm QuickSort, najwolniej MergeSort, natomiast algorytm BucketSort, ma czas niewiele gorszy i porównywalny z QuickSort, co jest raczej oczekiwane, ponieważ to QuickSort jest wykonywany w środku BucketSorta.

6 Bibliografia

- Wikipedia Merge sort
  `https://en.wikipedia.org/wiki/Merge_sort`
- Wikipedia Quicksort
  `https://en.wikipedia.org/wiki/Quicksort`
- Wikipedia Bucket sort
  `https://en.wikipedia.org/wiki/Bucket_sort`
- Merge sort na GeeksforGeeks
  `https://www.geeksforgeeks.org/merge-sort/`
- Quicksort na GeeksforGeeks
  `https://www.geeksforgeeks.org/quick-sort/`
- Jasne wytłumaczenie Quicksorta
  `https://www.youtube.com/watch?v=MZaf_9IZCrc`
- Hoare vs Lomuto
  `https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/`
- Bucket sort na GeeksforGeeks
  `https://www.geeksforgeeks.org/bucket-sort-2/`