Politechnika Wrocławska

Projektowanie Algorytmów i Metody Sztucznej Inteligencji

Projekt nr 2

Zadanie na ocenę bdb (5.0)

Autor:	Aleksander Łyskawa 275462
Wydział i kierunek studiów:	W12N, Automatyka i Robotyka
Termin zajęć:	pon 15:15-16:55
Prowadzący:	dr inż. Witold Paluszyński
Data:	25.04.2024

Spis treści

1 Opis zadania **3**
1.1 Treść polecenia . . . 3
1.2 Wybrane algorytmy sortowania . . . 3

2 Opis wybranych algorytmów **3**
2.1 Sortowanie przez scalanie (mergesort) . . . 3
2.2 Sortowanie szybkie (quicksort) . . . 3
2.2.1 Quicksort - wariant standardowy . . . 3
2.2.2 Quicksort - wariant z funkcją threeWayPartition . . . 4
2.3 Sortowanie kubełkowe (bucketsort) . . . 5

3 Działanie programu **6**
3.1 Kompilacja programu . . . 6
3.2 Argumenty programu . . . 6
3.3 Zasada działania programu . . . 7

4 Analiza złożoności obliczeniowej **9**
4.1 Quick sort . . . 9
4.1.1 Opis: . . . 9
4.1.2 Tabele pomiarowe . . . 9
4.1.3 Wykresy . . . 10
4.1.4 Wniosek . . . 10
4.2 Merge sort . . . 11
4.2.1 Opis: . . . 11
4.2.2 Tabela pomiarowa . . . 11
4.2.3 Wykres . . . 11
4.2.4 Wniosek . . . 12
4.3 Bucket sort . . . 12
4.3.1 Opis: . . . 12
4.3.2 Tabela pomiarowa . . . 12
4.3.3 Wykres . . . 13
4.3.4 Wniosek . . . 13
4.4 Porównanie czasowej złożoności obliczeniowej zaimplementowanych sortowań 14
4.4.1 Wniosek . . . 14
4.5 Testy . . . 14

5 Źródła **15**

1 Opis zadania

1.1 Treść polecenia

Należy wybrać trzy algorytmy sortowania. Zaimplementować wybrane algorytmy oraz przeprowadzić analizę ich efektywności na podanym zbiorze danych.

1.2 Wybrane algorytmy sortowania

Z tabelii zawartej w instrukcji zadania wybrałem trzy następujące algorytmy sortowania:

- sortowanie przez scalanie (mergesort)
- sortowanie szybkie (quicksort)
- sortowanie kubełkowe (bucketsort)

2 Opis wybranych algorytmów

2.1 Sortowanie przez scalanie (mergesort)

Merge Sort jest rekurencyjnym algorytmem sortowania, który działa w oparciu o strategię "dziel i zwyciężaj". Polega na podziale listy na pół, rekurencyjnym posortowaniu obu połówek, a następnie scaleniu ich w jedną posortowaną listę. Proces ten jest powtarzany aż do momentu, gdy lista zostanie podzielona na pojedyncze elementy, które są posortowane.

Moja implementacja sortowania przez scalanie opiera się na funkcji pomocniczej merge() oraz funkcji mergeSort:

Funkcja merge() przyjmuje trzy argumenty: wektor vec, liczby całkowite begin, mid i end, które określają granice dwóch posortowanych podwektorów. Tworzy dwa pomocnicze wektory left i right, do których kopiowane są odpowiednie elementy z wektora vec. Następnie używa indeksów i, j oraz k do porównywania elementów z wektorów left i right, oraz scalania ich w odpowiedniej kolejności do wektora vec.

Funkcja mergeSort() przyjmuje wektor vec oraz liczby całkowite begin i end, określające zakres elementów do posortowania. Funkcja oblicza środek zakresu w celu uzyskania dwóch części, wywołuje dla nich rekurencyjnie funkcję mergeSort, a następnie wywołuje funkcję merge, aby je połączyć. Funkcja kontynuuje sortowanie tak długo, jak istnieje więcej niż jeden element do posortowania w podanym zakresie.

2.2 Sortowanie szybkie (quicksort)

2.2.1 Quicksort - wariant standardowy

Quicksort to algorytm sortowania, który również wykorzystuje strategię "dziel i zwyciężaj". Polega na wyborze elementu pivot, a następnie podziale listy na dwie części: elementy mniejsze od pivota i elementy większe od pivota. Następnie obie części są sortowane rekurencyjnie.

Moja implementacja sortowania przez szybkie sortowanie opiera się na funkcji pomocniczej threeWayPpartition() oraz funkcji głównej quickSort:

Funkcja partition() przyjmuje jako argumenty referencję do wektora vec typu Movie oraz liczby całkowite start i end. Celem funkcji jest podział wektora na dwie części względem wybranego elementu pivot: po lewej stronie pivot znajdować się będą elementy mniejsze od niego, a po prawej większe.
Na początku funkcji wybierany jest element środkowy wektora jako pivot. Zmienne pomocnicze left i right są ustawiane odpowiednio na początkowym i końcowym indeksie przeszukiwanego fragmentu wektora.
Następnie funkcja wykonuje pętlę, w której wskaźniki left i right przemieszczają się w kierunku siebie nawzajem – left jest zwiększane dopóki element vec[left].key jest mniejszy od pivot, a right jest zmniejszane, gdy element vec[right].key jest większy od pivot. Gdy oba wskaźniki wskazują na elementy spełniające warunek zamiany, elementy te są zamieniane miejscami. Po zamianie left jest inkrementowane, a right dekrementowane.
Pętla kontynuowana jest dopóki left nie przekroczy right. Po zakończeniu pętli funkcja zwraca indeks left, który jest punktem podziału wektora na dwie części dla kolejnych kroków algorytmu QuickSort.

Funkcja quickSort() jako argumenty przyjmuje wektor vec oraz liczby całkowite begin i end, określające zakres elementów do posortowania. Warunek stopu: Jeśli end jest większe niż begin, to oznacza to, że istnieje więcej niż jeden element do posortowania w podanym zakresie. W takim przypadku funkcja kontynuuje sortowanie.Wywoływana jest funkcja partition w celu uzyskania elementu pivot, a następnie rekurencyjnie wywoływana jest funkcja quickSort dla dwóch podwektorów: od begin do pivot - 1 oraz od pivot + 1 do end. Pętla kontynuowana jest tak długo, jak end jest większe niż begin, co oznacza to, że istnieje więcej niż jeden element do posortowania w podanym zakresie.

2.2.2 Quicksort - wariant z funkcją threeWayPartition

Pisząc algorytm sortowania szybkiego w testach napotkałem na znaczny spadek wydajności programu, gdzie przy większych danych sięgających miliona czas działania programu przekraczał kilkadziesiąt sekund, co mogło być to spowodowane dużą ilością dupikatów w pliku. Problem finalnie udało się zastosowując pomocnicze zminne left i right w opisanej wyrzej funkcji partition(), ale dla uzyskania jeszcze lepszej wydajności z pomocą ChatuGPT zaimplementowałem drugi wariant sorotwania szybkiego wykorzystując funkcję threeWayPartition() opisaną poniżej.

Moja implementacja sortowania przez szybkie sortowanie opiera się na funkcji pomocniczej threeWayPartition() oraz funkcji głównej quickSort:

Funkcja threeWayPartition() przyjmuje cztery argumenty: wektor vec przechowujący struktury Movie, indeksy begin i end określające zakres działania funkcji, oraz referencje „i” i „j”, które po zakończeniu funkcji wskazują odpowiednio na końcówkę sekcji elementów mniejszych od piwota oraz początek sekcji elementów większych od piwota. Funkcja iteruje przez elementy i przesuwa je do odpowiednich sekcji zgodnie z ich wartością względem piwota. Użycie referencji „i"i „j"pozwala na zwrócenie z funkcji dwóch wartości.

Funkcja quickSort() przyjmuje jako argumenty wektor vec oraz indeksy begin i end, które określają zakres sortowanych danych.threeWayPartition do efektywniejszego sortowania danych, szczególnie tych z dużą liczbą powtarzających się elementów.
Algorytm zaczyna od wywołania threeWayPartition służącej do efektywniejszego sortowania danych, szczególnie tych z dużą liczbą powtarzających się elementów,które partycjonuje wektor na trzy sekcje. Następnie funkcja rekurencyjnie stosuje samą siebie do sortowania sekcji zawierających elementy mniejsze od piwota oraz większe od piwota. Środkowa sekcja (elementy równe piwotowi) nie wymaga dalszego sortowania, co jest jedną z kluczowych zalet tego podejścia, gdy mamy do czynienia z dużą liczbą duplikatów — zmniejsza to ogólną liczbę potrzebnych operacji porównań i przestawień.

2.3 Sortowanie kubełkowe (bucketsort)

Bucketsort to algorytm sortowania, który wykorzystuje koncepcję kubełków do posortowania danych. Algorytm polega na podziale zbioru danych na odpowiednią liczbę kubełków, a następnie sortowaniu elementów wewnątrz każdego kubełka. Po posortowaniu kubełków, elementy są łączone w celu uzyskania posortowanej listy.

Funkcja bucketSort() jako argument pobiera referencje na wektor oraz ilość kubełków. Funkcja wyszukuje najmniejszą (minRating) i największą (maxRating) ocenę wśród filmów, aby określić zakres wartości, który zostanie podzielony na kubełki. Następnie funkcja tworzy wektor buckets, który zawiera numBuckets pustych wektorów (std::vector<Movie>). Dla każdego filmu obliczany jest indeks kubełka, do którego ma zostać przypisany, na podstawie jego oceny. Obliczenie to uwzględnia minimalną ocenę i zakres wartości podzielony na liczbę kubełków. Filmy są następnie dodawane do odpowiednich kubełków. Każdy kubełek jest sortowany osobno za pomocą standardowej funkcji std::sort, a posortowane kubełki są następnie scalane z powrotem w odpowiedniej kolejności do jednego wektora movies.

3 Działanie programu

3.1 Kompilacja programu

Program został skompilowany do pliku wykonywalnego następującymi komendami:
g++ sorting.cpp -c
g++ main.cpp sorting.o -o prog

3.2 Argumenty programu

Argumenty wywołania programu mają postać:

prog input_file_name sort_key_pos n_items algo_name shuffle_passes shuffle_seed

Oznaczją kolejno:

- input_file_name zadaje nazwę pliku z którego należy pobierać dane. Dodatkowo, jeżeli nazwa pliku wejściowego będzie zadana jako „-" to dane wejściowe odczytywane będą ze standardowego wejścia stdin.
- sort_key_pos zadaje numer pola w pliku wejściowym które stanowi klucz sortowania
- n_items zadaje liczbę elementów, które wczytane zostaną do pamięci i posortowane
- algo_name zadaje nazwę algorytmu sortowania
- shuffle_passes będącym parametrem opcjonalnym zadaje liczbę przebiegów mieszania, które wykonuje na danych wczytanych z pliku
- shuffle_seed będąc parametrem opcjonalnym zadaje ziarno generatora liczb losowych użyte do zainicjowania generatora liczb pseudolosowych przed mieszaniem pliku. Jeśli ten parametr nie zostanie zadany, to ziarno wygenerowane zostanie w sposób losowy.

3.3 Zasada działania programu

Funckja main() do zrealizowania zadanego polecenia ma postać:

```
int main(int argc, const char * argv[])
{
    if(argc < 5 )
    {
        std::cout << "Usage: ./prog input_file_name sort_key_pos
   n_items algo_name shuffle_passes shuffle_seed << std::endl";
        return 1;
    }
    std::string input_file_name = argv[1];
    int sort_key_pos = atoi(argv[2]);
    int n_items = atoi(argv[3]);
    std::string algo_name = argv[4];

    int shuffle_passes = 0;
    int shuffle_seed = 0;

    if (argc > 5)
        shuffle_passes = std::atoi(argv[5]);

    if (argc > 6)
        shuffle_seed = std::atoi(argv[6]);*/

    std::vector<Movie> movies;
    if(input_file_name == "-")
        movies = loadDataFromStdin(sort_key_pos);
    else
        movies = filterData(input_file_name, n_items,sort_key_pos);

    shuffleVector(movies, shuffle_passes, shuffle_seed);
    sortMovies(movies, algo_name);
    displayMovies(movies);
    std::cerr << "Average: " << average(movies) << std::endl;
    std::cerr << "Median: " << median(movies) << std::endl;
    return 0;
}
```

- **shuffleVector()**

  Funkcja ta przetasowuje elementy wektora obiektów typu Movie, korzystając z algorytmu Mersenne Twister do generowania liczb losowych. Przyjmuje argumenty: vec jako referencję do wektora, który ma zostać przetasowany, shuffle_passes jako liczbę operacji przetasowania wektora oraz wartość ziarna dla generatora liczb losowych. Jeśli ustawiona na 0, wybierane jest losowe ziarno na podstawie urządzenia losującego systemu w przeciwnym razie używane jest podane ziarno. Funkcja inicjalizuje silnik Mersenne Twister (std mt19937) i odpowiednio go inicjuje, co pozwala na większą losowość niż w wypadku użycia srand(time(NULL)).

- **average() i median()**

  Funkcje przyjmują const refernecję na wektor obiektów typy Movie i zwracają odpowiednio jego średnią oraz mediane rankingów.

- **displayMovies()**

  Funkcje przyjmują const refernecję na wektor obiektów typy Movie i wyswietla jego elementy na wyjściu stdout.

- **sortMovies()**

  Funckja przyjmuję refernecję na wektor obiektów typy Movie oraz nazwę sortowania jaką ma wykonać. Jeśli sortType jest "QUICK", używany jest algorytm sortowania szybkiego w zmodyfikownej wersj, a dla "STANDARDQUICK"stndardowa wersja sorotwania szybkiego. Jeśli sortType to "MERGE", zastosowany zostaje algorytm sortowania przez scalanie. Natomiast dla sortType "BUCKET", wykorzystywany jest algorytm sortowania kubełkowego. Dla każdego przypadku sortowania, funkcja mierzy czas wykonania sortowania i poprawnie wyświetla wynik w zależności od zmierzonego czasu z jednostką w mikrosekundach (us), milisekundach (ms) lub sekundach (s). Wartości czasu są wypisywane na standardowe wyjście std::cerr.

- **filterData())**

  Funkcja przyjmuje ona trzy parametry: nazwę pliku, maksymalną liczbę wierszy do przeczytania oraz pozycję klucza sortującego w wierszu. Funkcja otwiera plik, pomija pierwszy wiersz, a następnie czyta kolejne wiersze do określonej liczby, próbując w każdym z nich odczytać klucz sortujący z wskazanej pozycji. Jeżeli klucz jest poprawny, dany wiersz jest dodawany do wektora jako obiekt Movie.

- **loadDataFromStdin**

  Funkcja przyjmuje jeden parametr, który określa pozycję klucza sortującego w każdej linii danych, gdzie dane są rozdzielone znakiem "".$Napocz\k{a}tku funkcja wy\'swietlain formajceowprow$

4 Analiza złożoności obliczeniowej

4.1 Quick sort

4.1.1 Opis:

– **Najlepszy przypadek** $O(nlogn)$**:**

Najlepszy przypadek występuje, gdy przy każdym podziale elementy są dzielone na dwie równie liczne części.

– **Średni przypadek** $O(nlogn)$**:**

Quicksort osiąga swoją optymalną złożoność średnią, gdy elementy są dzielone na mniej więcej równych części przy każdym podziale.

– **Najgorszy przypadek** $O(n^2)$**:**

Najgorszy przypadek występuje, gdy wybrany pivot jest zawsze najmniejszym lub największym elementem z zestawu, co prowadzi do bardzo niezbalansowanych podziałów.

4.1.2 Tabele pomiarowe

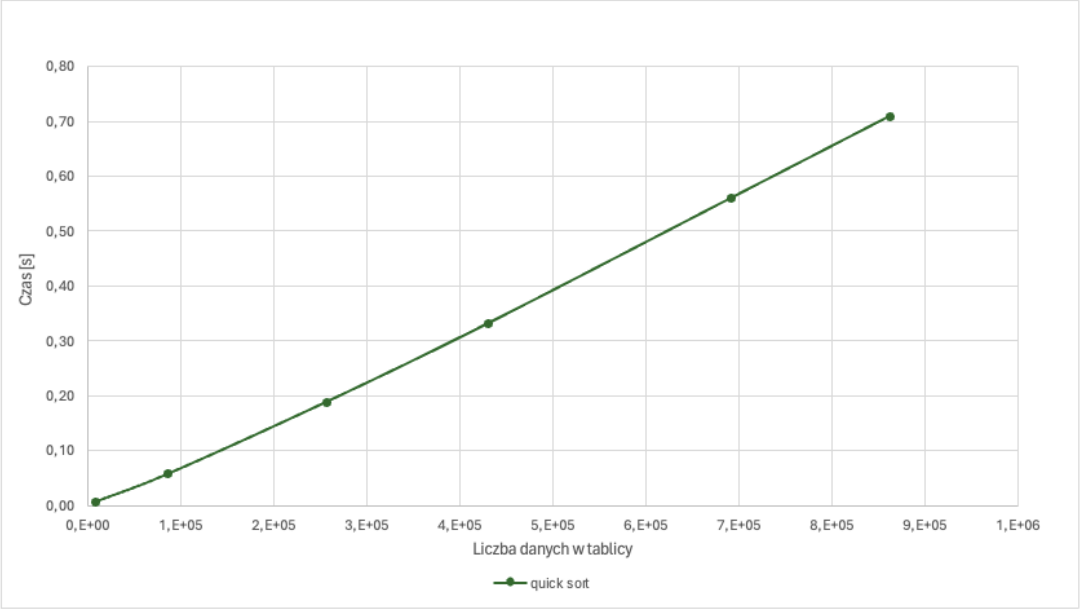
Quciksort - standard		
liczba zadanych danyh	liczba danych po odfiltrowaniu	czas [ms]
10000	8982	6,23
100000	86786	57,22
300000	256747	187,98
500000	430885	331,39
800000	691752	559,34
1000000	862708	708,74

Rysunek 1: Tabela pomiarowa dla standardowego sortowania szybkiego

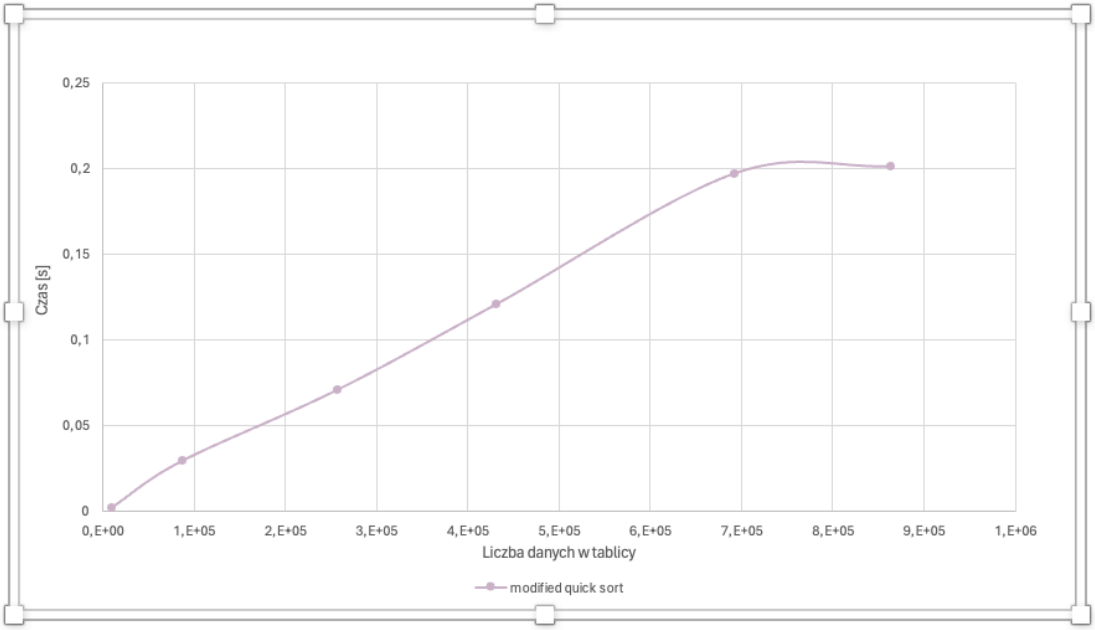
Quicksort - threeWayPartition		
liczba zadanych danyh	liczba danych po odfiltrowaniu	czas [ms]
10000	8982	2,18875
100000	86786	29,5629
300000	256747	70,8767
500000	430885	120,673
800000	691752	196,812
1000000	862708	201,169

Rysunek 2: Tabela pomiarowa dla modyfikowanego sortowania szybkiego

4.1.3 Wykresy



Rysunek 3: Wykres zależnośnosci czasu od danych



Rysunek 4: Wykres zależnośnosci czasu od danych

4.1.4 Wniosek

Z wykresów można oszacować, że złozoności obliczeniowa zapisana w postaci dużego O dla obydwu sortowań są zbliżone do wartości O(nlogn)

4.2 Merge sort

4.2.1 Opis:

– **Każdy przypadek** $O(nlogn)$**:**

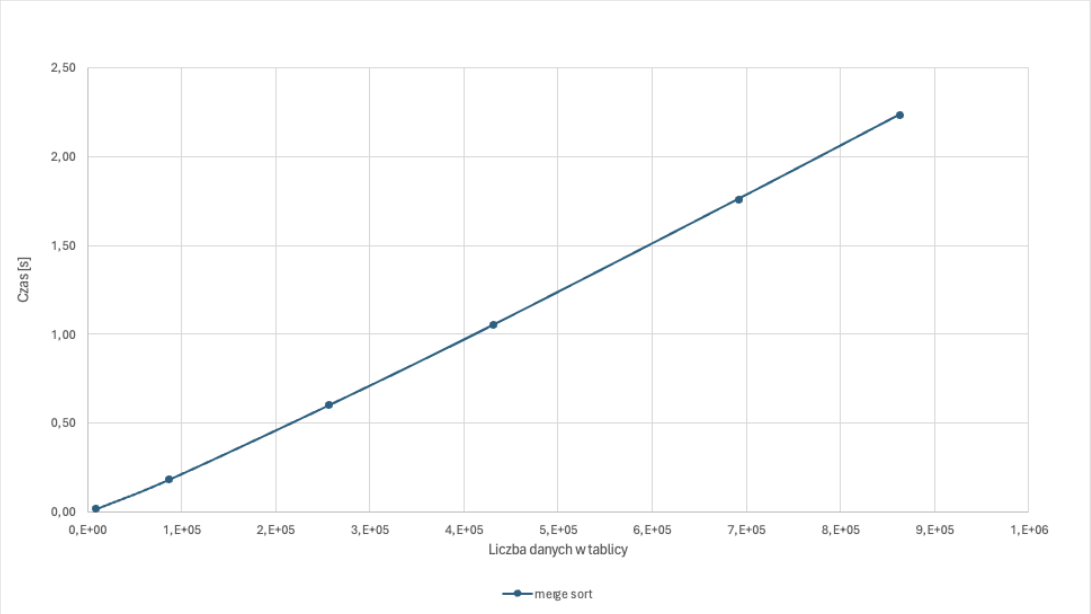
Mergesort zawsze dzieli dane na dwie równie liczne części i łączy je z powrotem w posortowaną całość, co skutkuje stałą złożonością

4.2.2 Tabela pomiarowa

Mergesort		
liczba zadanych danyh	liczba danych po odfiltrowaniu	czas [ms]
10000	8982	16,01
100000	86786	182,73
300000	256747	602,18
500000	430885	1053,86
800000	691752	1763,84
1000000	862708	2237,74

Rysunek 5: Tabela pomiarowa dla sortowania przez scalanie

4.2.3 Wykres



Rysunek 6: Wykres zależnośnosci czasu od danych

4.2.4 Wniosek

Z wykresu można oszacować, że złozóność obliczeniowa zapisana w postaci dużego O jest równa O(nlogn)

4.3 Bucket sort

4.3.1 Opis:

– **Najlepszy przypadek** $O(n)$**:**

W idealnym przypadku, gdy każdy element trafia do osobnego kubełka, a każdy kubełek zawiera tylko jeden element lub gdy elementy w kubełkach są już posortowane.

– **Średni przypadek** $O(n + k)$**:**

Gdzie n to liczba elementów do posortowania, a k to liczba kubełków. Działa efektywnie, gdy dane są równomiernie rozłożone i dobrze pasują do mechanizmu kubełkowania.

– **Najgorszy przypadek** $O(n^2)$**:**

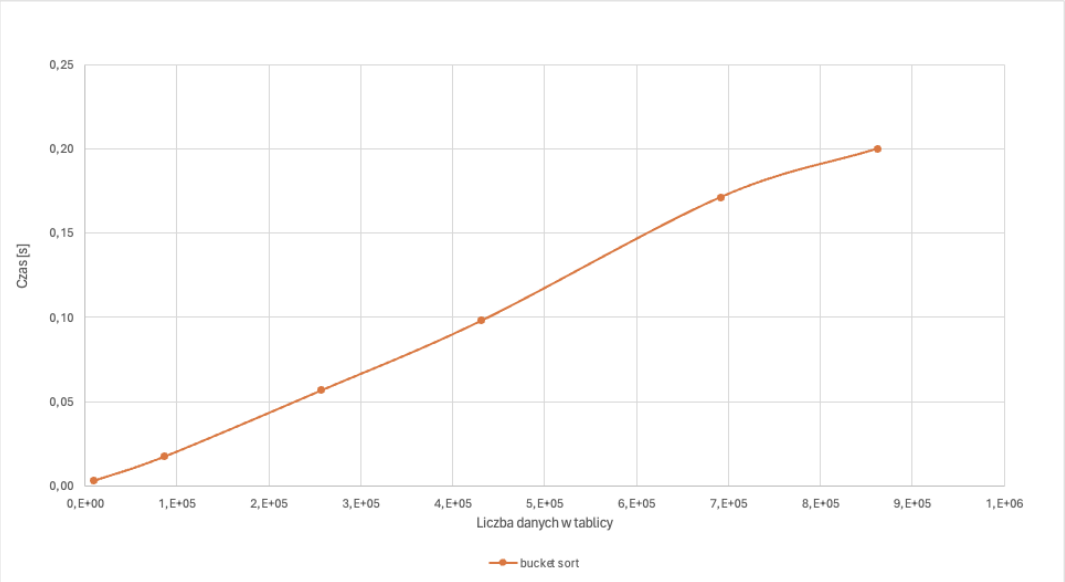
Gdy wszystkie elementy trafiają do jednego kubełka, wówczas sortowanie wewnątrz tego kubełka może znacznie wydłużyć czas działania.

4.3.2 Tabela pomiarowa

Bucketsort		
liczba zadanych danyh	liczba danych po odfiltrowaniu	czas [ms]
10000	8982	3,00
100000	86786	17,54
300000	256747	56,59
500000	430885	97,91
800000	691752	171,24
1000000	862708	199,99

Rysunek 7: Tabela pomiarowa dla sortowania kubełkowego

4.3.3 Wykres

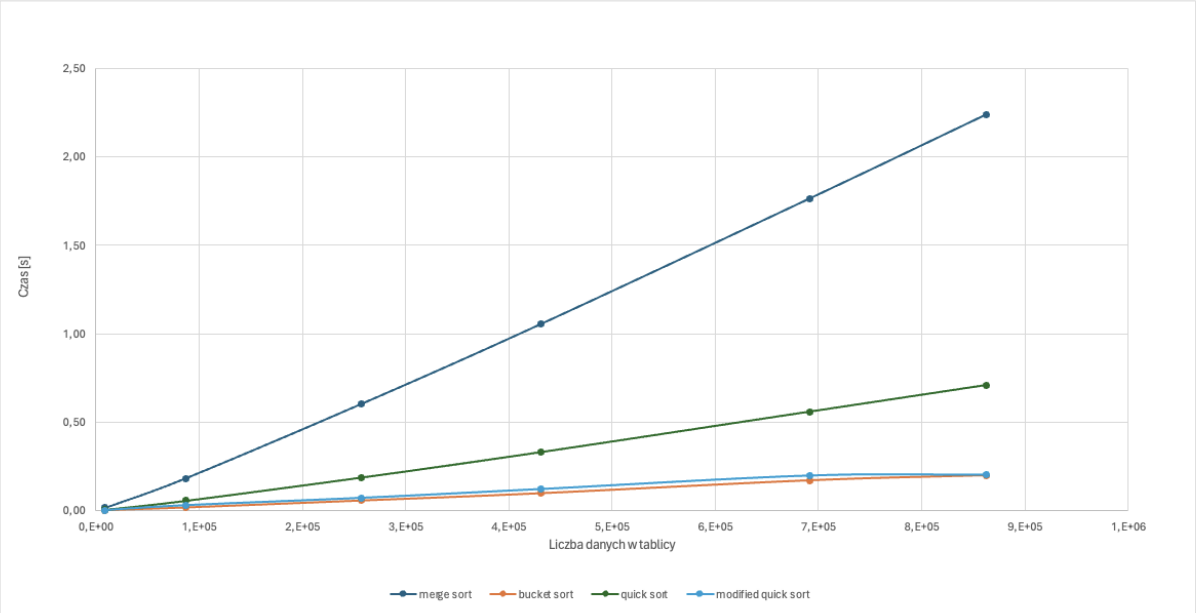


Rysunek 8: Wykres zależnośnosci czasu od danych

4.3.4 Wniosek

Z wykresu można oszacować, że złozóność obliczeniowa zapisana w postaci dużego O jest znajduje doposaowanie pomiędzy O(n) a O(nlogn).

4.4 Porównanie czasowej złożoności obliczeniowej zaimplementowanych sortowań



Rysunek 9: Wykres zależnośnosci czasu od danych

4.4.1 Wniosek

Najszybszym algorytmem dla zadanej bazy danych okazało się sortowanie kubełkowe wywoływany liczbą kubełków równą 1000, niewiele wolniejszy był zmodyfikowany wariant sortowania szybkiego, następnie standardowa wersja sortowania szybkiego, a na końcu sortowanie przez scalanie.

4.5 Testy

Wszystkie testy związane z badaniem złożoności obliczeniowej programu zostały przeprowadzone na laptopie MacBook Air M1. Laptop ten jest wyposażony w procesor Apple M1 z 8 rdzeniami CPU i 8 GB pamięci RAM. Testy zostały wykonane na systemie macOS przy użyciu domyślnego środowiska uruchomieniowego.

5 Źródła

Quicksort:

- https://www.youtube.com/watch?v=vhSLT3a-t-A
- https://www.youtube.com/watch?v=Vtckgz38QHs
- https://www.algorytm.edu.pl/algorytmy-maturalne/quick-sort.html
- https://www.geeksforgeeks.org/cpp-program-for-quicksort/

Funkcja threeWayPartition:

- ChatGPT
- Github Copilot
- https://www.cs.princeton.edu/courses/archive/spring13/cos226/demo/23DemoPartitioningD
- https://www.geeksforgeeks.org/3-way-quicksort-dutch-national-flag/

Mergesort:

- https://www.youtube.com/watch?v=3j0SWDX4AtU
- https://www.youtube.com/watch?v=4VqmGXwpLqc
- https://www.youtube.com/watch?v=9kvpCdHjqNk
- https://www.geeksforgeeks.org/merge-sort/?ref=shm

Bucketsort:

- ChatGPT
- Github Copilot
- https://www.geeksforgeeks.org/bucket-sort-2
- https://www.youtube.com/watch?v=vjahkW_xLBI

shuffleVector():

- https://www.geeksforgeeks.org/stdmt19937-class-in-cpp/
- ChatGPT
- Github Copilot

Powyższe źródła nie zostały skopiowane bezpośrednio do programu. Zamiast tego, zostały wykorzystane jedynie jako wsparcie merytoryczne do samodzielnego rozwiązania zadania.