Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji
Kierunek: Informatyczne Systemy Automatyki

Projektowanie i analiza algorytmów

Michał Wróblewski
272488

Termin zajęć:
wtorek 13:15 - 15:00

7 marca 2024

Spis treści

Spis tabel

Spis rysunków

Streszczenie

Niniejsze sprawozdanie przedstawia analizę i porównanie wybranych algorytmów sortowania pod kątem ich wydajności. Badaniu poddano sortowanie kubełkowe, sortowanie przez scalanie i sortowanie szybkie.

1 Wprowadzenie

Sortowanie jest fundamentalną operacją w informatyce, wykorzystywaną w wielu dziedzinach. Istnieje wiele algorytmów sortowania, różniących się złożonością obliczeniową, stabilnością i sposobem działania. Celem tego sprawozdania jest analiza i porównanie wybranych algorytmów sortowania pod kątem ich złożoności obliczeniowej.

2 Opis badanych algorytmów

2.1 Sortowanie kubełkowe

- Złożoność obliczeniowa:
  - Średni przypadek: $O(n)$
  - Najgorszy przypadek: $O(n^2)$
- Opis działania: Algorytm dzieli dane w zależności od zakresu, który jest ustalony na podstawie wartości minimalnej i maksymalnej z tablicy danych. Dzieli dane na podzakresy (kubły), wpisując wartości same w sobie jako indeks kubła, aby następnie, gdy taka wartość pojawi się w tablicy, zwiększyć licznik o jeden. Ostatecznie należy przypisać do tablicy indeksy, w zależności od tego jak duży jest licznik danego indeksu.
- Łatwość implementacji: Wysoka

2.2 Sortowanie przez scalanie

- Złożoność obliczeniowa:
  - Średni przypadek: $O(n \log n)$
  - Najgorszy przypadek: $O(n \log n)$
- Opis działania: Algorytm dzieli dane na mniejsze podtablice. Wykonuje to dla prawej i lewej strony w zależności od elementu środkowego (m). Następnie dane są scalane w jedną tablicę. Scalanie polega na przyrównaniu do siebie lewej wartości z prawą wartością w następującej po sobie kolejności. Algorytm wykorzystuje rekurencje.
- Łatwość implementacji: Średnia; problem z implementacją scalania.

2.3 Sortowanie szybkie

- Złożoność czasowa:
  - Średni przypadek: $O(n \log n)$
  - Najgorszy przypadek: $O(n^2)$

- Opis działania: Algorytm wybiera element bazowy (pivot), a następnie wyszukuje z lewej strony elementu większego od elementu bazowego, a z prawej strony szuka elementu większego niż nasz pivot, następnie zamienia je ze sobą. Dzieje się tak do momentu, gdy lewa strona przekroczy prawą, co jest uwarunkowane tym że chcemy, aby nasze dane były posortowane od najmniejszego do największego. Wszystko dzieje się rekurencyjnie.
- Łatwość implementacji: Średnia

3 Przebieg eksperymentów

Eksperymenty zostały przeprowadzone na tablicach o różnych rozmiarach (od 100 000 do ponad 1 000 000 elementów) wypełnionych losowymi liczbami. Do pomiaru czasu wykonania algorytmów użyto funkcji ‘chrono()‘.

4 Przefiltrowanie danych

W celu poprawy dokładności analizy algorytmów sortowania przefiltrowałem puste wpisy oraz nałożyłem ograniczenie oceny mniejszej bądź równej 10. Puste wpisy mogą wprowadzać zakłócenia w czasie wykonania algorytmów sortowania oraz mogą prowadzić do błędnych wyników analizy.

4.1 Metoda

Do przefiltrowania danych użyłem warunku if, który sprawdził mi czy pobrana wartość istnieje i czy jest mniejsza badz równa 10.

4.2 Czas przeszukiwania

Czas przeszukiwania został zmierzony dla każdego z algorytmów sortowania po przefiltrowaniu danych. Uzyskane wyniki zostały przedstawione w tabeli poniżej.

5 Złożoność obliczeniowa

1. **Sortowanie kubełkowe**:
   - Dostarczone dane sugerują, że złożoność obliczeniowa sortowania kubełkowego jest bliska liniowej ($O(n)$).
   - Czas wykonania rośnie bardzo wolno wraz z rozmiarem danych wejściowych, co jest charakterystyczne dla algorytmów o liniowej złożoności.

```
Funkcja bucketSort(tab, tabSize):
    min = 0.0                                   // O(1)
    max = 0.0                                   // O(1)

    Dla każdego i od 0 do tabSize wykonaj:      // O(n)
        Jeżeli i == 0:                          // O(1)
            min = tab[i].rating                 // O(1)
```

```
            max = tab[i].rating                    // O(1)
        W przeciwnym razie:                        // O(1)
            Jeżeli tab[i].rating > max:            // O(1)
                max = tab[i].rating                // O(1)
            W przeciwnym razie jeżeli tab[i].rating < min:  // O(1)
                min = tab[i].rating                // O(1)

    numberOfBuckets = max - min + 1                // O(1)

    Stwórz tablicę buckets o rozmiarze numberOfBuckets + 1 i
    ↪ zainicjuj ją zerami  // O(numberOfBuckets)

    Dla każdego i od 0 do tabSize wykonaj:         // O(n)
        Jeżeli tab[i].rating jest pomiędzy 0 a numberOfBuckets:  //
        ↪ O(1)
            buckets[tab[i].rating]++               // O(1)

    k = 0                                          // O(1)
    Dla każdego i od 0 do numberOfBuckets wykonaj:  //
    ↪ O(numberOfBuckets)
        Dla każdego j od buckets[i] do 1 wykonaj:
            tab[k++].rating = i                    // O(n)

    Zwolnij pamięć zaalokowaną dla tablicy buckets  // O(1)
```

2. **Sortowanie przez scalanie**:

   - Dane wskazują, że złożoność obliczeniowa sortowania przez scalanie jest zbliżona do $O(n \log n)$.
   - Czas wykonania rośnie wolniej niż liniowo, ale szybciej niż w przypadku sortowania kubełkowego, co jest zgodne z oczekiwaniami dla algorytmów o złożoności $O(n \log n)$.

```
Funkcja merge(tab, l, m, r, temp):
    tempSize = r - l + 1                       // O(1)
    Jeżeli tempSize > 1:                        // O(1)
        a = l, b = m + 1, i = 0                 // O(1)
        Dopóki i < tempSize wykonuj:            // O(n)
            Jeżeli a <= m oraz (b > r lub tab[a].rating <
            ↪ tab[b].rating):  // O(1)
                temp[i++] = tab[a++]           // O(1)
            W przeciwnym razie:
                temp[i++] = tab[b++]           // O(1)

        Dla każdego j od 0 do tempSize wykonuj:  // O(n)
            tab[l + j] = temp[j]               // O(1)

Funkcja mergeSort(tab, l, r):
    Jeżeli l < r:                               // O(1)
        temp = nowa tablica o rozmiarze r - l + 1  // O(1)
```

```
m = floor((l + r) / 2)                    // O(1)
mergeSort(tab, l, m)                      // O(log n) w
↪ najlepszym przypadku, a O(n) w najgorszym przypadku
mergeSort(tab, m + 1, r)                  // O(log n) w
↪ najlepszym przypadku, a O(n) w najgorszym przypadku
merge(tab, l, m, r, temp)                 // O(n)
Usuń tablicę temp                         // O(1)
```

3. **Sortowanie szybkie**:

   - Dane sugerują, że złożoność obliczeniowa sortowania szybkiego również jest zbliżona do $O(n \log n)$.
   - Czas wykonania rośnie szybciej niż liniowo, ale wolniej niż kwadratowo, co jest charakterystyczne dla algorytmów o złożoności $O(n \log n)$.

```
Funkcja quicksort(tab, a, b):
    Jeżeli a < b:                                    // O(1)
        middle = (a + b) / 2                         // O(1)
        pivotIndex = losowa liczba z zakresu od a do b  // O(1)
        pivot = tab[pivotIndex]                      // O(1)
        l = a, r = b                                  // O(1)

        Jeżeli pivot.rating != tab[middle].rating: // O(1)
            zamień miejscami tab[pivotIndex] i tab[middle]  // O(1)

        Dopóki l <= r wykonuj:                       // O(n)
            Dopóki tab[l].rating < pivot.rating wykonuj:  // O(n)
                l++                                  // O(1)
            Dopóki tab[r].rating > pivot.rating wykonuj:  // O(n)
                r--                                  // O(1)

            Jeżeli l <= r:                           // O(1)
                zamień miejscami tab[l] i tab[r]     // O(1)
                l++                                  // O(1)
                r--                                  // O(1)

        quicksort(tab, a, r)                         // O(log n) w
        ↪ najlepszym przypadku, a O(n) w najgorszym przypadku
        quicksort(tab, l, b)                         // O(log n) w
        ↪ najlepszym przypadku, a O(n) w najgorszym przypadku
```

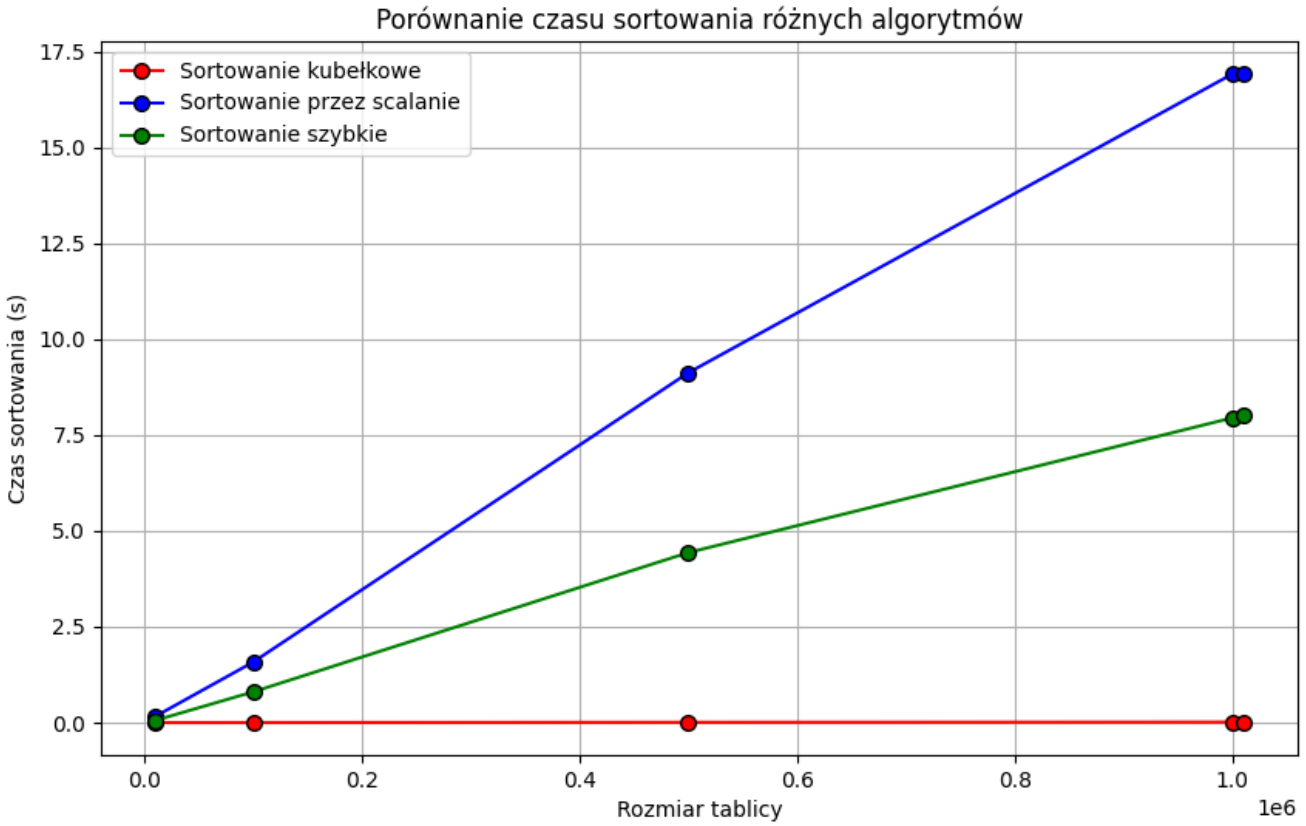
6 Wyniki

Uzyskane wyniki zostały przedstawione w tabeli 1 i na wykresie 1.

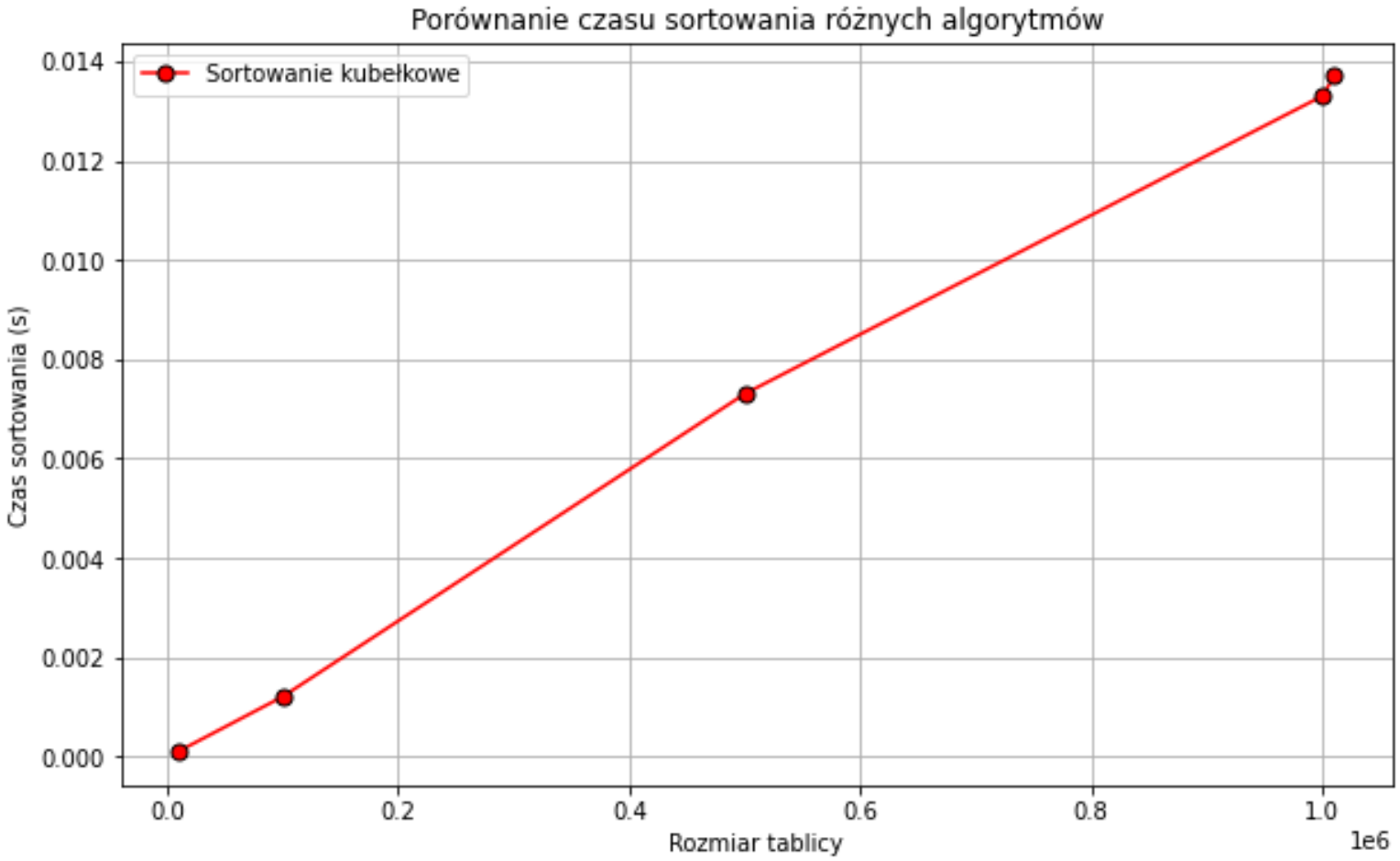
Uzyskane wyniki zostały przedstawione na poniższych wykresach:

Rozmiar tablicy	Sortowanie kubełkowe	Sortowanie przez scalanie	Sortowanie szybkie
10000	0.0001	0.1627	0.0398
100000	0.0011	1.5792	0.5401
500000	0.0073	9.1213	3.4320
1000000	0.0133	16.9078	6.1904
Max	0.0137	16.9207	6.2201

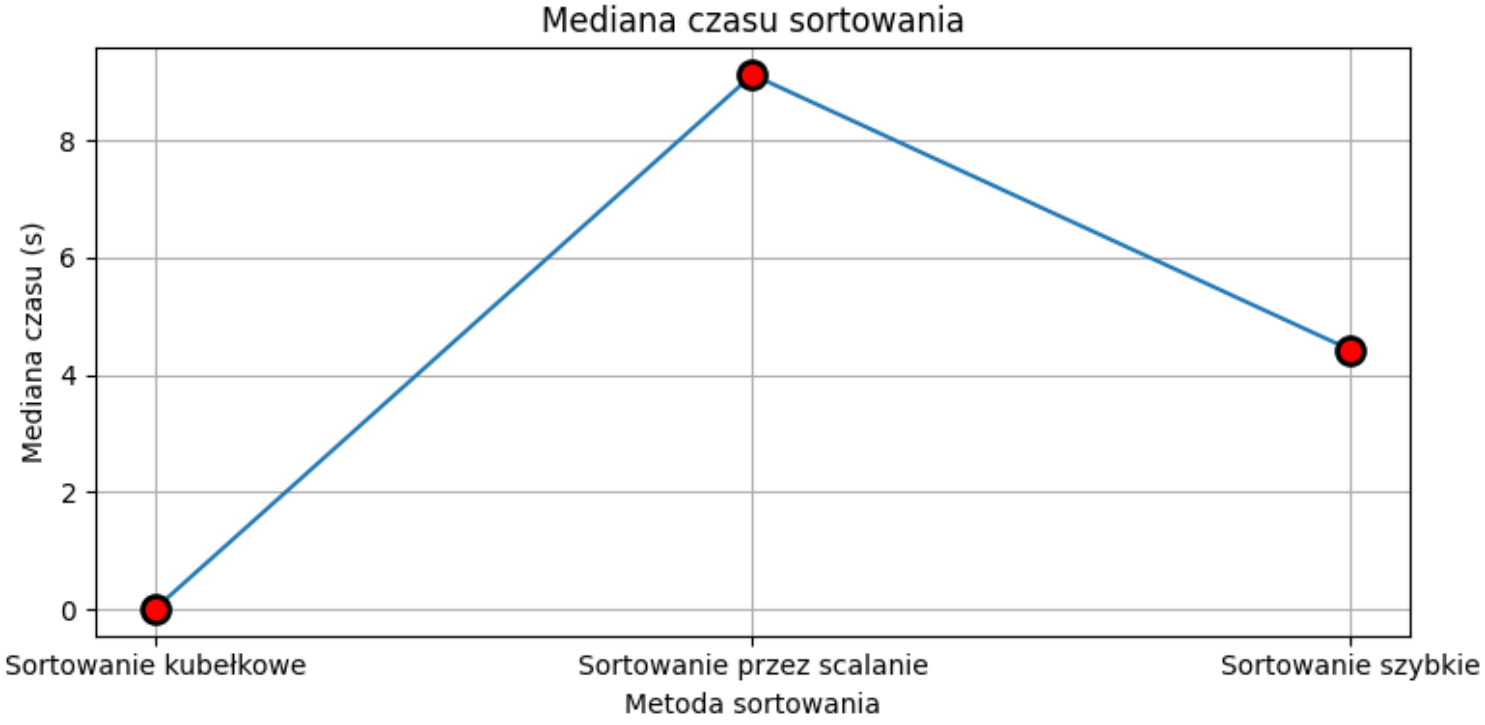
Tabela 1: Czas wykonania algorytmów sortowania (w sekundach)



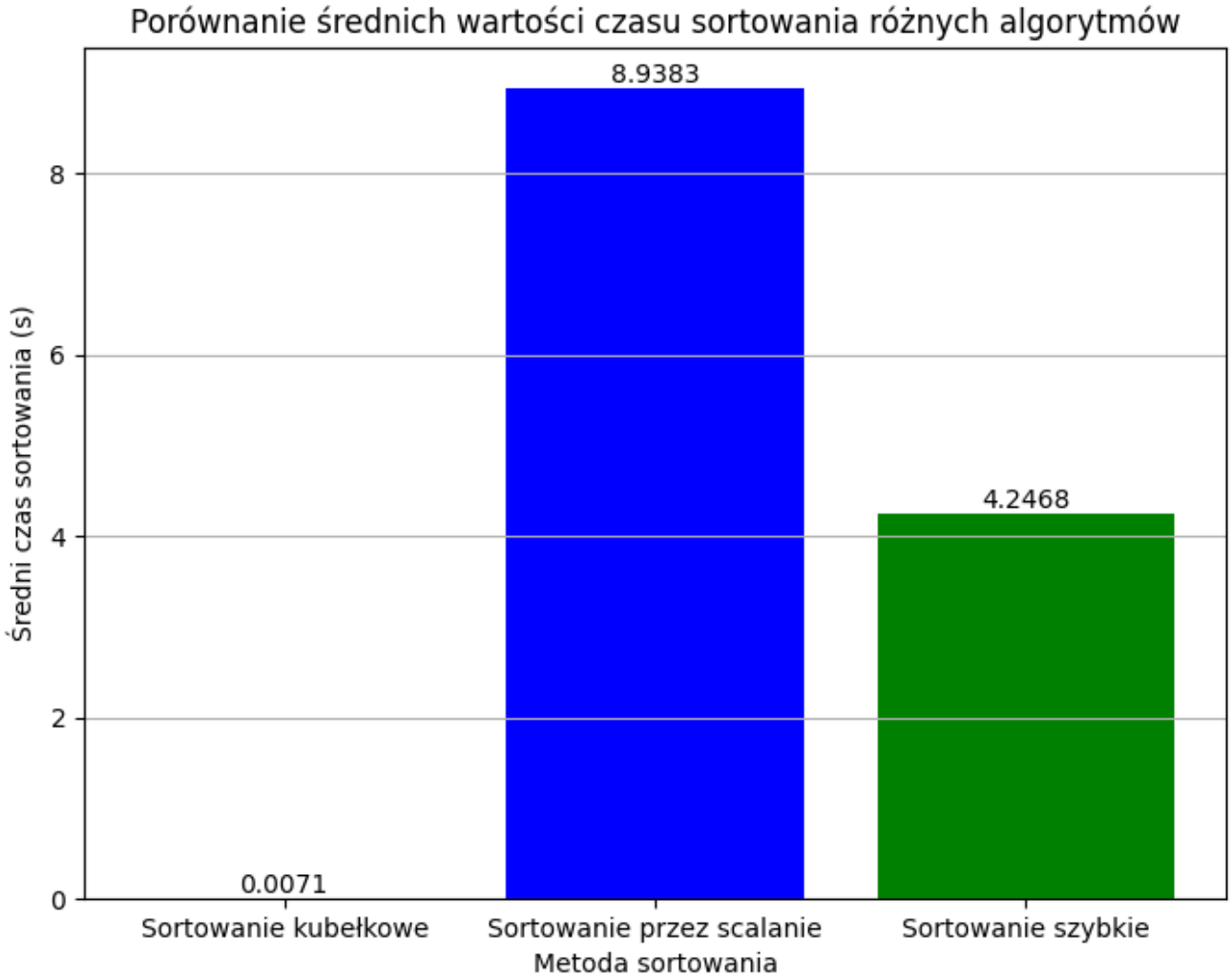
Rysunek 1: Porównanie czasu wykonania algorytmów sortowania



Rysunek 2: Wykres sortowania kubełkowego



Rysunek 3: Mediana czasu sortowania



Rysunek 4: Średnie wartości czasu

7 Podsumowanie i wnioski

Przeprowadzone eksperymenty pokazują, że sortowanie kubełkowe jest najszybszym algorytmem dla podanych danych, ponieważ sortuje on dane w naszym przypadku na jedynie 11 kubełków, dlatego też jest tak wydajny i nieporównywalny względem mergesort czy quicksort, gdzie dane są sortowane w pełnym zakresie.

Dla dużych tablic mergesort i quicksort stają się porównywalnie wydajne (comparably efficient), lecz jedynie w ich zakresie.

Dostarczone dane wydają się zgadzać się z oczekiwanymi złożonościami obliczeniowymi dla poszczególnych algorytmów sortowania.

8 Bibliografia

1. `https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie`

2. https://pl.wikipedia.org/wiki/Asymptotyczne_tempo_wzrostu

3. https://pl.wikipedia.org/wiki/Sortowanie_szybkie

4. https://pl.wikipedia.org/wiki/Sortowanie_kubeÅĆkowe

5. https://binarnie.pl/sortowanie-kube%C5%82kowe/