# Analiza Algorytmów Sortowania

Ksawery Józefowski
Politechnika Wrocławska

November 5, 2024

## Contents

# 1 Wstep

W ramach Listy 1 zaimplementowano i przeanalizowano trzy klasyczne algorytmy sortowania: *Insertion Sort*, *Merge Sort* oraz *Heap Sort*. Każdy z algorytmów został zmodyfikowany: *Insertion Sort* wstawia jednocześnie dwa elementy, *Merge Sort* dzieli dane na trzy cześci zamiast dwóch, a *Heap Sort* korzysta z kopców ternarnych. Celem projektu jest analiza wydajności algorytmów pod wzgledem liczby porównań i przypisań dla różnych rozmiarów danych.

# 2 Fragmenty Kodów

Poniżej przedstawiono najciekawsze fragmenty kodu dla wybranych algorytmów.

## 2.1 Insertion Sort z Wstawianiem Dwóch Elementów

Klasyczny algorytm *Insertion Sort* działa poprzez iteracyjne wstawianie kolejnych elementów do uporzadkowanej cześci tablicy, przesuwajac wieksze elementy, aby zrobić miejsce na nowy element. Modyfikacja wprowadzona w tym algorytmie polega na jednoczesnym wstawianiu dwóch kolejnych elementów:

- Dwa kolejne elementy z nieuporzadkowanej cześci sa najpierw porównywane, aby ustalić ich kolejność.
- Nastepnie wiekszy z nich jest wstawiany na pozycje bardziej wysunieta w kierunku końca tablicy.
- Proces wstawiania odbywa sie iteracyjnie, gdzie każdy z dwóch elementów porównywany jest z kolejnymi elementami uporzadkowanej cześci tablicy i wstawiany na odpowiednie miejsce.

Dzieki temu podejściu liczba operacji wstawiania może zostać zmniejszona, gdyż jednocześnie przetwarzane sa dwa elementy.

```
void insertionSortDouble(float arr[], int n, int& comparisons
    , int& assignments) {
    for (int i = 2; i < n; i += 2) {
        float key1 = arr[i];
        float key2 = arr[i - 1];
        assignments += 2;

        if (++comparisons && key1 < key2) {
            std::swap(key1, key2);
            assignments += 3;
        }

        int j = i - 2;

        while (j >= 0 && ++comparisons && arr[j] > key1) {
            arr[j + 2] = arr[j];
            assignments++;
            j--;
        }

        arr[j + 2] = key1;
        assignments++;

        while (j >= 0 && ++comparisons && arr[j] > key2) {
            arr[j + 1] = arr[j];
            assignments++;
            j--;
        }

        arr[j + 1] = key2;
        assignments++;
    }
    if (n % 2 == 0) {
        int lastKey = arr[n - 1];
        assignments++;
        int j = n - 2;

        while (j >= 0 && ++comparisons && arr[j] > lastKey) {
            arr[j + 1] = arr[j];
            assignments++;
            j--;
        }

        arr[j + 1] = lastKey;
        assignments++;
    }
}
```

Listing 1: Insertion Sort z wstawianiem dwóch elementów

## 2.2 Merge Sort z Podziałem na Trzy Cześci

Algorytm *Merge Sort* w klasycznej wersji dzieli tablice na dwie cześci, które sa rekurencyjnie sortowane, a nastepnie łaczone w uporzadkowany sposób. Modyfikacja polega na dzieleniu tablicy na trzy cześci, co zmienia strukture rekurencyjnych wywołań:

- Tablica jest dzielona na trzy cześci o zbliżonej liczbie elementów.
- Każda z cześci jest sortowana rekurencyjnie za pomoca *Merge Sort*.
- Po zakończeniu sortowania każda z trzech cześci jest łaczona w jedna, uporzadkowana całość.

Zmiana ta powoduje, że podczas łaczenia musimy operować na trzech posortowanych fragmentach, co zwieksza złożoność procesu scalania, ponieważ liczba operacji porównawczych wzrasta. Z drugiej strony zmniejsza sie liczba poziomów rekurencji, co może przyspieszyć algorytm.

```
void mergeSort3way(float arr[], int left, int right, int&
    comparisons, int& assignments) {
    if (left < right) {
        comparisons++;
        int third = (right - left) / 3;
        int mid1 = left + third;
        int mid2 = right - third;

        mergeSort3way(arr, left, mid1, comparisons,
            assignments);
        mergeSort3way(arr, mid1 + 1, mid2, comparisons,
            assignments);
        mergeSort3way(arr, mid2 + 1, right, comparisons,
            assignments);

        merge3way(arr, left, mid1, mid2, right, comparisons,
            assignments);
    }
}
```

Listing 2: Merge Sort z podziałem na trzy cześci

## 2.3 Heap Sort z Kopcem Ternarnym

Algorytm *Heap Sort* opiera sie na koncepcji kopca binarnego – struktury danych, w której każdy wezeł ma co najwyżej dwóch potomków, a klucz rodzica jest wiekszy lub równy kluczom jego potomków (kopiec maksymalny). Klasyczny *Heap Sort* składa sie z dwóch głównych etapów:

- **Budowanie kopca**: Tablica wejściowa jest przekształcana w kopiec maksymalny.
- **Sortowanie**: Najwiekszy element (szczyt kopca) jest zamieniany z ostatnim elementem tablicy, a kopiec jest ponownie porzadkowany, aby zachować właściwość kopca.

W modyfikacji *Heap Sort* zastosowano kopiec ternarny, w którym każdy wezeł może mieć do trzech potomków. Dzieki tej zmianie zmniejsza sie głebokość kopca, co wpływa na zmniejszenie liczby operacji porównawczych przy wyprowadzaniu najwiekszego elementu na szczyt kopca.

Poniżej przedstawiono fragment kodu implementujacego *Heap Sort* z kopcem ternarnym:

```
void ternaryHeapify(float arr[], int n, int i, int&
    comparisons, int& assignments) {
    int largest = i;
    int left = 3 * i + 1;
    int middle = 3 * i + 2;
    int right = 3 * i + 3;

    if (left < n && ++comparisons && arr[left] > arr[largest
        ]) largest = left;
    if (middle < n && ++comparisons && arr[middle] > arr[
        largest]) largest = middle;
    if (right < n && ++comparisons && arr[right] > arr[
        largest]) largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        assignments += 3;
        ternaryHeapify(arr, n, largest, comparisons,
            assignments);
    }
}

void ternaryHeapSort(float arr[], int n, int& comparisons,
    int& assignments) {
    for (int i = n / 3; i >= 0; --i) ternaryHeapify(arr, n, i
        , comparisons, assignments);

    for (int i = n - 1; i >= 0; --i) {
        std::swap(arr[0], arr[i]);
        assignments += 3;
        ternaryHeapify(arr, i, 0, comparisons, assignments);
    }
}
```

Listing 3: Heap Sort z kopcem ternarnym

# 3 Analiza i Wyniki

Testy przeprowadzono na losowych tablicach o rozmiarach 1000, 5000, 10000 elementów. Dla każdego algorytmu zliczono liczbe porównań oraz przypisań.

## 3.1 Tabela Wyników

| Rozmiar danych | Algorytm | Porównania | Przypisania |
|---|---|---|---|
| 10 | Insertion Sort | 18 | 27 |
| 10 | Insertion Sort (Double) | 16 | 27 |
| 10 | Merge Sort | 22 | 68 |
| 10 | Merge Sort (Three-Way) | 37 | 55 |
| 10 | Heap Sort | 37 | 78 |
| 10 | Heap Sort (Ternary) | 41 | 46 |
| 10000 | Insertion Sort | 24886600 | 24896608 |
| 10000 | Insertion Sort (Double) | 12312894 | 12327768 |
| 10000 | Merge Sort | 120429 | 267232 |
| 10000 | Merge Sort (Three-Way) | 226002 | 166360 |
| 10000 | Heap Sort | 235363 | 372447 |
| 10000 | Heap Sort (Ternary) | 226431 | 166800 |
| 100000 | Insertion Sort | 1450825191 | 1450946597 |
| 100000 | Insertion Sort (Double) | 1250835994 | 1250985962 |
| 100000 | Merge Sort | 1536358 | 3337856 |
| 100000 | Merge Sort (Three-Way) | 2886413 | 2094331 |
| 100000 | Heap Sort | 3019604 | 4724586 |
| 100000 | Heap Sort (Ternary) | 2897214 | 2090924 |

Table 1: Porównanie liczby operacji dla różnych rozmiarów danych

## 3.2 Wykresy Wyników

Poniżej przedstawiono wykresy liczby porównań i przypisań dla poszczególnych algorytmów w zależności od rozmiaru danych.

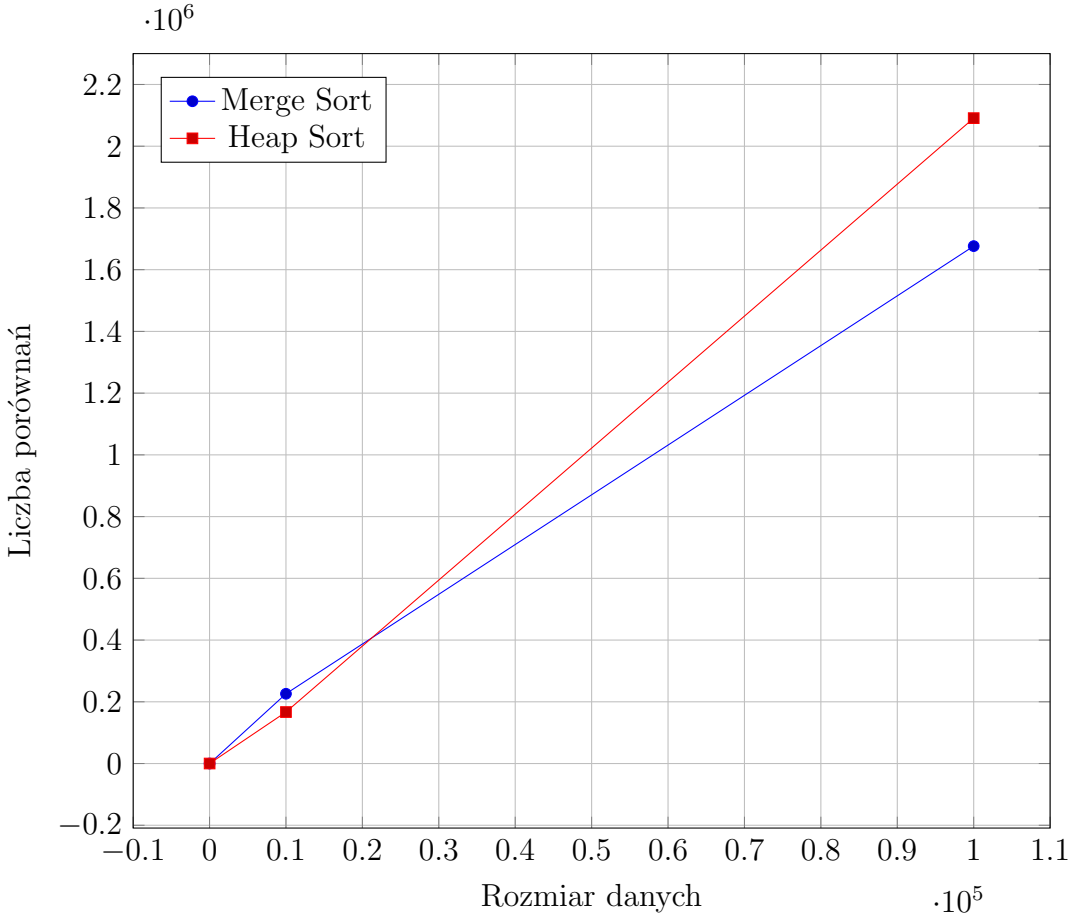


Figure 1: Liczba porównań w zależności od rozmiaru danych

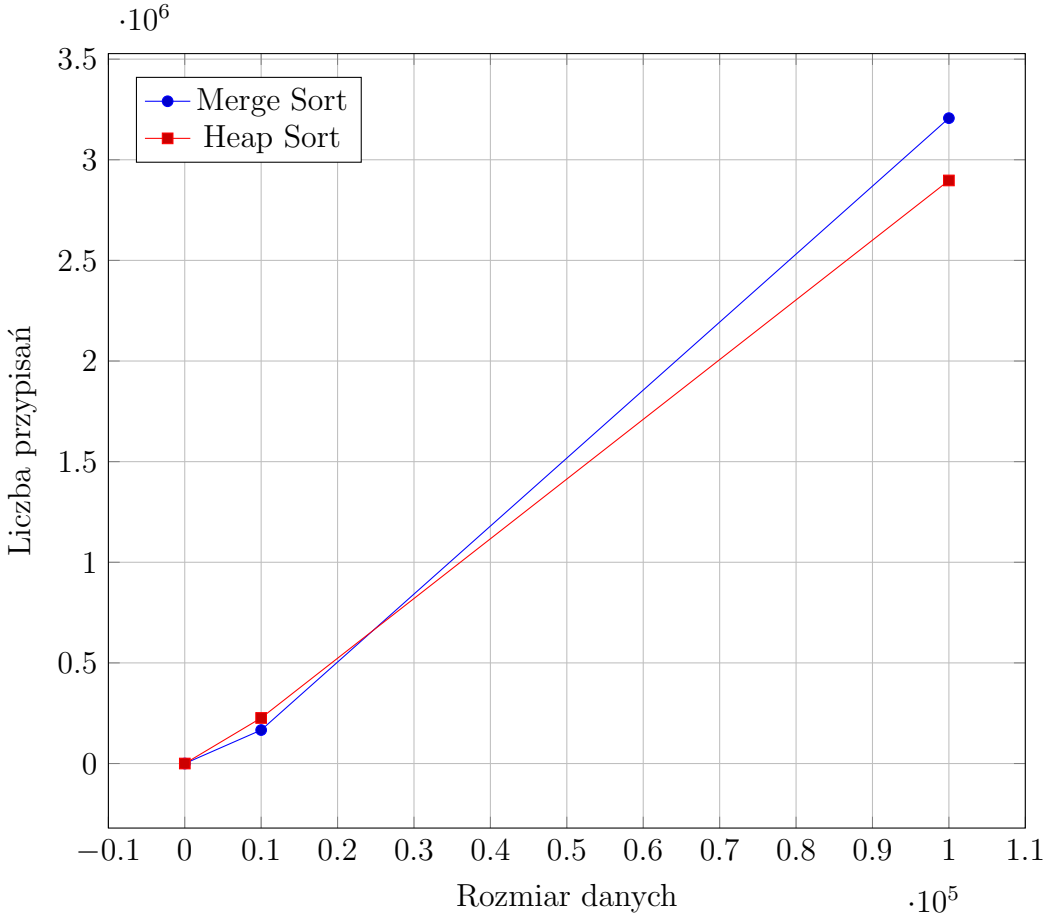


Figure 2: Liczba przypisań w zależności od rozmiaru danych

# 4 Wnioski

Przeprowadzona analiza wykazała, że:

- *Insertion Sort* z wstawianiem dwóch elementów zmniejsza liczbe operacji dla małych zestawów danych, lecz staje sie mniej efektywny przy dużych rozmiarach tablicy.
- *Merge Sort* z podziałem na trzy cześci zwieksza złożoność algorytmu, co skutkuje wieksza liczba porównań, ale zmniejsza liczbe przypisań w porównaniu do tradycyjnego *Merge Sort*.
- *Heap Sort* z kopcem ternarnym zmniejsza głebokość kopca, co skraca liczbe operacji porównawczych przy dużych zbiorach danych, jednak zwieksza liczbe przypisań.

Z tego wzgledu, klasyczna wersja *Merge Sort* okazała sie najbardziej efektywna przy dużych danych, zachowujac korzystna równowage miedzy liczba porównań a liczba przypisań.