

# Analiza Algorytmów Sortowania

Ksawery Józefowski  
Politechnika Wrocławska

November 8, 2024

## Contents

<b>1</b>	<b>Wstęp</b>	<b>2</b>
<b>2</b>	<b>Fragmenty Kodów</b>	<b>2</b>
2.1	Insertion Sort z Wstawianiem Dwóch Elementów . . . . .	2
2.2	Merge Sort z Podziałem na Trzy Części . . . . .	4
2.3	Heap Sort z Kopcem Ternarym . . . . .	6
<b>3</b>	<b>Analiza i Wyniki</b>	<b>8</b>
3.1	Tabela Wyników . . . . .	8
3.2	Wykresy Wyników . . . . .	8
<b>4</b>	<b>Wnioski</b>	<b>12</b>

# 1 Wstęp

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 części zamiast dwóch, a *Heap Sort* korzysta z kopców ternarnych. Celem projektu jest analiza wydajności algorytmów pod względem 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 uporządkowanej części tablicy, przesuwając większe elementy, aby zrobić miejsce na nowy element. Modyfikacja wprowadzona w tym algorytmie polega na jednoczesnym wstawianiu dwóch kolejnych elementów:

- Dwa kolejne elementy z nieuporządkowanej części są najpierw porównywane, aby ustalić ich kolejność.
- Następnie większy z nich jest wstawiany na pozycję bardziej wysuniętą w kierunku końca tablicy.
- Proces wstawiania odbywa się iteracyjnie, gdzie każdy z dwóch elementów porównywany jest z kolejnymi elementami uporządkowanej części tablicy i wstawiany na odpowiednie miejsce.

Dzięki temu podejściu liczba operacji wstawiania może zostać zmniejszona, gdyż jednocześnie przetwarzane są dwa elementy.

```

1 void insertionSortDouble(float arr[], int n, unsigned long
  long& comparisons, unsigned long long& assignments) {
2     for (int i = 2; i < n; i += 2) {
3         float key1 = arr[i];
4         float key2 = arr[i - 1];
5         assignments += 2;
6
7         if (++comparisons && key1 < key2) {
8             std::swap(key1, key2);
9             assignments += 3;
10        }
11
12        int j = i - 2;
13
14        while (j >= 0 && ++comparisons && arr[j] > key1) {
15            arr[j + 2] = arr[j];
16            assignments++;
17            j--;
18        }
19
20        arr[j + 2] = key1;
21        assignments++;
22
23        while (j >= 0 && ++comparisons && arr[j] > key2) {
24            arr[j + 1] = arr[j];
25            assignments++;
26            j--;
27        }
28
29        arr[j + 1] = key2;
30        assignments++;
31    }
32    if (n % 2 == 0) {
33        int lastKey = arr[n - 1];
34        assignments++;
35        int j = n - 2;
36
37        while (j >= 0 && ++comparisons && arr[j] > lastKey) {
38            arr[j + 1] = arr[j];
39            assignments++;
40            j--;
41        }
42
43        arr[j + 1] = lastKey;
44        assignments++;
45    }
46 }

```

Listing 1: Insertion Sort z wstawianiem dwóch elementów

## 2.2 Merge Sort z Podziałem na Trzy Części

Algorytm *Merge Sort* w klasycznej wersji dzieli tablice na dwie części, które są rekurencyjnie sortowane, a następnie łączone w uporządkowany sposób. Modyfikacja polega na dzieleniu tablicy na trzy części, co zmienia strukturę rekurencyjnych wywołań:

- Tablica jest dzielona na trzy części o zbliżonej liczbie elementów.
- Każda z części jest sortowana rekurencyjnie za pomocą *Merge Sort*.
- Po zakończeniu sortowania każda z trzech części jest łączona w jedną, uporządkowaną całość.

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

```

1 void merge3way(float arr[], int left, int mid1, int mid2, int
   right, unsigned long long& comparisons, unsigned long
   long& assignments) {
2     float temp[right - left + 1];
3     int i = left, j = mid1 + 1, k = mid2 + 1, idx = 0;
4
5     while (i <= mid1 && j <= mid2 && k <= right) {
6         if (arr[i] <= arr[j] && arr[i] <= arr[k]) temp[idx++]
           = arr[i++];
7         else if (arr[j] <= arr[i] && arr[j] <= arr[k]) temp[
           idx++] = arr[j++];
8         else temp[idx++] = arr[k++];
9         comparisons += 2;
10        assignments++;
11    }
12
13    while (i <= mid1 && j <= mid2) {
14        temp[idx++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
15        comparisons++;
16        assignments++;
17    }
18    while (j <= mid2 && k <= right) {
19        temp[idx++] = arr[j] <= arr[k] ? arr[j++] : arr[k++];
20        comparisons++;
21        assignments++;
22    }
23    while (i <= mid1 && k <= right) {
24        temp[idx++] = arr[i] <= arr[k] ? arr[i++] : arr[k++];
25        comparisons++;
26        assignments++;
27    }
28
29    while (i <= mid1) {
30        temp[idx++] = arr[i++];
31        assignments++;
32    }
33    while (j <= mid2) {
34        temp[idx++] = arr[j++];
35        assignments++;
36    }
37    while (k <= right) {
38        temp[idx++] = arr[k++];
39        assignments++;
40    }
41
42    for (int l = 0; l < idx; ++l) {
43        arr[left + l] = temp[l];
44        assignments++;
45    }

```

```

46 }
47
48 void mergeSort3way(float arr[], int left, int right, unsigned
    long long& comparisons, unsigned long long& assignments)
    {
49     if (left < right) {
50         int third = (right - left) / 3;
51         int mid1 = left + third;
52         int mid2 = right - third;
53
54         mergeSort3way(arr, left, mid1, comparisons,
            assignments);
55         mergeSort3way(arr, mid1 + 1, mid2, comparisons,
            assignments);
56         mergeSort3way(arr, mid2 + 1, right, comparisons,
            assignments);
57
58         merge3way(arr, left, mid1, mid2, right, comparisons,
            assignments);
59     }
60 }

```

Listing 2: Merge Sort z podziałem na trzy części

## 2.3 Heap Sort z Kopcem Ternarnym

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

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

W modyfikacji *Heap Sort* zastosowano kopiec ternarny, w którym każdy węzeł może mieć do trzech potomków. Dzięki tej zmianie zmniejsza się głębokość kopca, co wpływa na zmniejszenie liczby operacji porównawczych przy wyprowadzaniu największego elementu na szczyt kopca.

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

```
1 void ternaryHeapify(float arr[], int n, int i, unsigned long
   long& comparisons, unsigned long long& assignments) {
2     int largest = i;
3     int left = 3 * i + 1;
4     int middle = 3 * i + 2;
5     int right = 3 * i + 3;
6
7     if (left < n && ++comparisons && arr[left] > arr[largest
   ]) largest = left;
8     if (middle < n && ++comparisons && arr[middle] > arr[
   largest]) largest = middle;
9     if (right < n && ++comparisons && arr[right] > arr[
   largest]) largest = right;
10
11    if (largest != i) {
12        std::swap(arr[i], arr[largest]);
13        assignments += 3;
14        ternaryHeapify(arr, n, largest, comparisons,
   assignments);
15    }
16}
17
18 void ternaryHeapSort(float arr[], int n, unsigned long long&
   comparisons, unsigned long long& assignments) {
19     for (int i = n / 3; i >= 0; --i) ternaryHeapify(arr, n, i
   , comparisons, assignments);
20
21     for (int i = n - 1; i >= 0; --i) {
22         std::swap(arr[0], arr[i]);
23         assignments += 3; // Przypisania dla zamiany
24         ternaryHeapify(arr, i, 0, comparisons, assignments);
25     }
26}
```

Listing 3: Heap Sort z kopcem ternarnym

### 3 Analiza i Wyniki

Testy przeprowadzono na losowych tablicach o rozmiarach 10, 10000, 100000 elementów. Dla każdego algorytmu zliczono średnia liczbę porównań, przypisań i czas wykonania.

#### 3.1 Tabela Wyników

Rozmiar danych	Algorytm	Porównania	Przypisania	Czas(s)
10	Insertion Sort	16	34	1.8e-07
10	Insertion Sort (Double)	15	27	9e-08
10	Merge Sort	23	68	2.41e-06
10	Merge Sort (Three-Way)	22	52	3.1e-07
10	Heap Sort	23	78	3.3e-07
10	Heap Sort (Ternary)	40	69	9.3e-07
10000	Insertion Sort	25377684	25397682	0.0454163
10000	Insertion Sort (Double)	12743307	12755796	0.0266696
10000	Merge Sort	120423	267232	0.00202331
10000	Merge Sort (Three-Way)	149314	173914	0.00093419
10000	Heap Sort	166316	372396	0.00142246
10000	Heap Sort (Ternary)	226539	250245	0.00134903
100000	Insertion Sort	2495403337	2495603335	4.485
100000	Insertion Sort (Double)	1246284306	1246409672	2.25296
100000	Merge Sort	1536679	3337856	0.0197074
100000	Merge Sort (Three-Way)	1911782	2175230	0.0120563
100000	Heap Sort	2163214	4725123	0.0175235
100000	Heap Sort (Ternary)	2897570	3136902	0.0172788

Table 1: Porównanie liczby operacji dla różnych rozmiarów danych

#### 3.2 Wykresy Wyników

Poniżej przedstawiono wykresy liczby porównań, przypisań i czasów dla Merge i Heap sorta w zależności od rozmiaru danych.



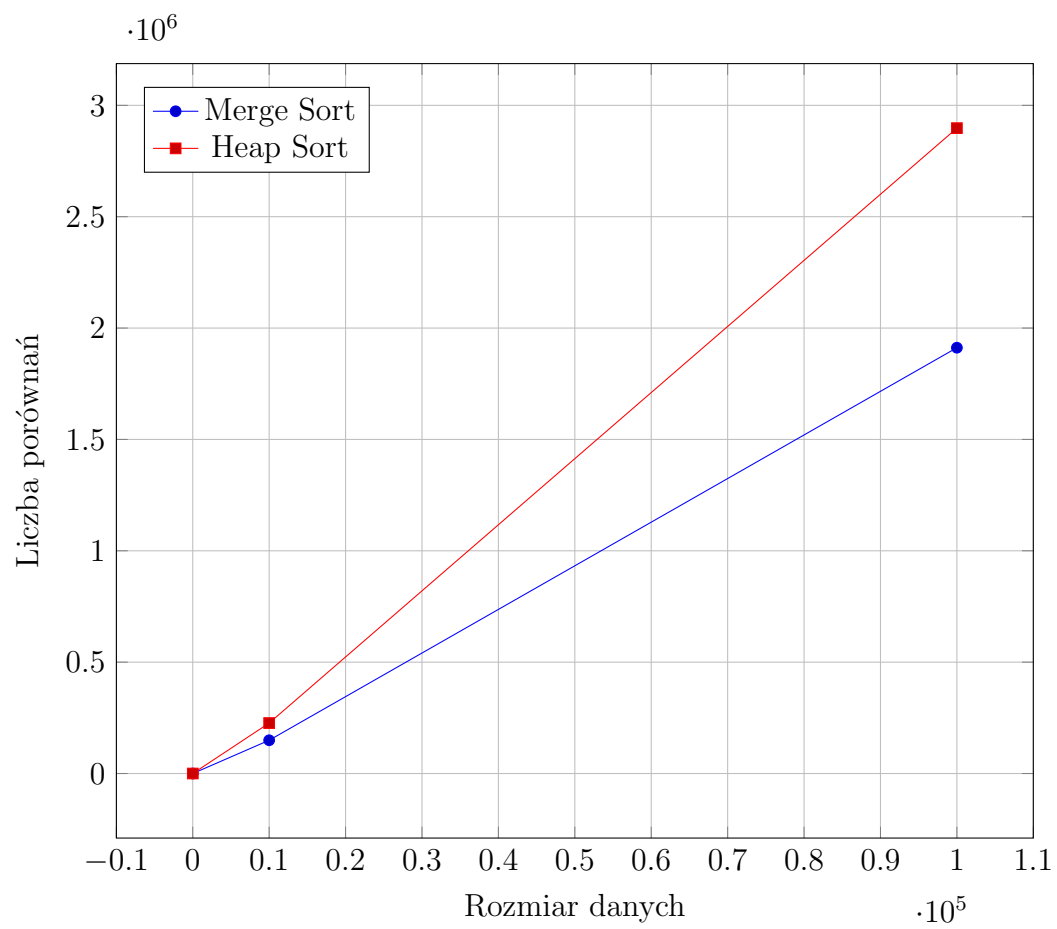


Figure 1: Liczba porównań w zależności od rozmiaru danych

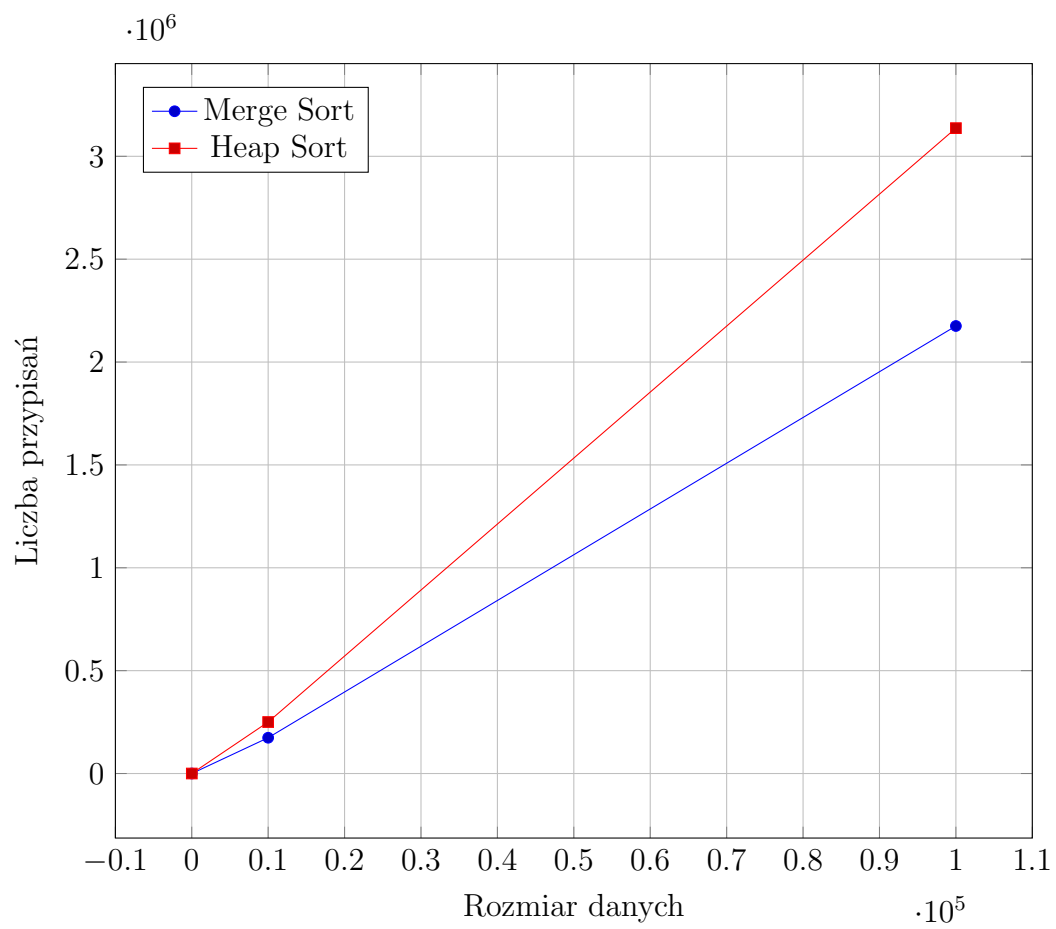


Figure 2: Liczba przypisań w zależności od rozmiaru danych

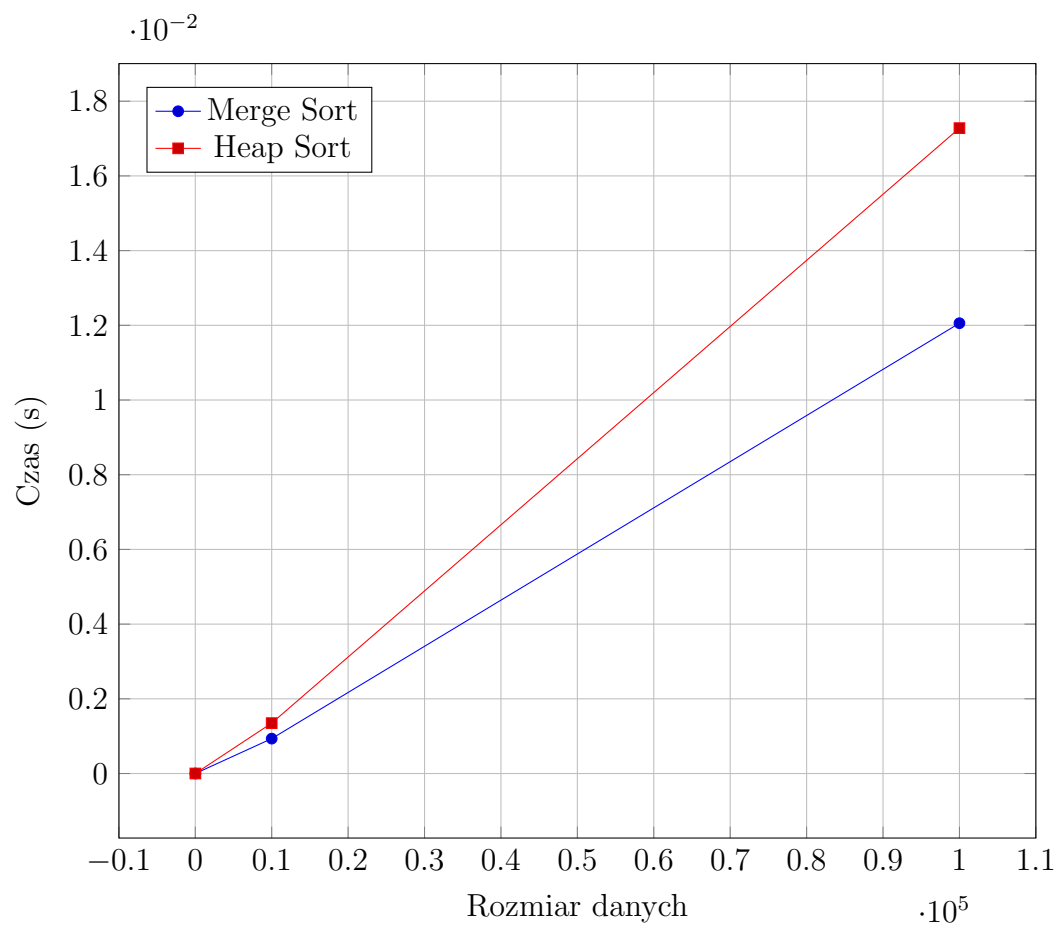


Figure 3: Czas wykonania w zależności od rozmiaru danych

## 4 Wnioski

Przeprowadzona analiza wykazała, że:

- *Insertion Sort* z wstawianiem dwóch elementów zmniejsza liczbę operacji dla małych zestawów danych, lecz liczba operacji, które wykonuje powoduje, że jest mniej efektywny niż Merge i Heap sort.
- *Merge Sort* z podziałem na trzy części zwiększa złożoność algorytmu, co skutkuje większą liczbą porównań, ale zmniejsza liczbę przypisań w porównaniu do tradycyjnego *Merge Sort*.
- *Heap Sort* z kopcem ternarnym zmniejsza głębokość kopca, co skraca liczbę operacji porównawczych przy dużych zbiorach danych, jednak zwiększa liczbę przypisań.

Z tego względu, klasyczna wersja i zmodyfikowana wersja *Merge Sort* okazały się najbardziej efektywne przy dużych danych, zachowując korzystną równowagę między liczbą porównań a liczbą przypisań.