

# sprawozdanie lista 1

Jakub Knaspki

November 2025

## 1 Wprowadzenie

Sprawozdanie przedstawia analizę algorytmów sortujących: Insertion Sort z modyfikacją podwójnego wstawiania, Merge Sort z podziałem na trzy części oraz Heap Sort z kopcem ternarnym. Analiza obejmuje porównanie liczby operacji (przypisań i porównań) oraz czasu wykonania dla różnych charakterystyk danych wejściowych.

## 2 INSERTION SORT

Klasyczny algorytm Insertion Sort polega na iteracyjnym wstawianiu każdego kolejnego elementu w odpowiednie miejsce w uporządkowanej części tablicy.

**Modyfikacja z podwójnym wstawianiem:** Zaproponowana modyfikacja wprowadza równoczesne przetwarzanie dwóch sąsiednich elementów. Algorytm został podzielony na dwa przypadki w zależności od parzystości liczby elementów:

- **Przypadek parzysty:** Rozpoczynamy od pierwszej pary elementów. Najpierw porównujemy je ze sobą i odpowiednio układamy, następnie wstawiamy oba elementy (najpierw mniejszy, potem większy) na właściwe pozycje w posortowanej części tablicy.
- **Przypadek nieparzysty:** Rozpoczynamy od drugiego elementu, traktując pierwszy jako początkowo posortowany fragment, a następnie postępujemy analogicznie jak w przypadku parzystym dla kolejnych par elementów.

Głównym celem tej modyfikacji jest optymalizacja polegająca na tym, aby w każdej iteracji ostatnim przetwarzanym elementem był zawsze element o indeksie  $i+1$ , co pozwala na efektywniejsze wykorzystanie już posortowanej części tablicy.

Poniżej przedstawiono omawiany fragment kodu.

```
if (rozmiar % 2==0)
{
    for (int i = 0;i<rozmiar;i+=2)
    {
        Wstawianie(i);
    }
}
else
{
    for (int i = 1;i<rozmiar;i+=2)
    {
        Wstawianie(i);
    }
}
```

### 3 Porównanie z standardowym Insertion Sort

#### 3.1 Wyniki dla standardowego Insertion Sort

```
2 14 5 1 33 85
1 2 5 14 33 85
```

w tym algorytmie masz 22 przypisan  
w tym algorytmie masz 18 porównan

#### 3.2 Wyniki dla zmodyfikowanego Insertion Sort

```
2 14 5 1 33 85
1 2 5 14 33 85
```

w tym algorytmie masz 18 przypisan  
w tym algorytmie masz 19 porównan

#### 3.3 Analiza porównawcza

Algorytm	Przypisania	Porównania
Standardowy Insertion Sort	22	18
Zmodyfikowany Insertion Sort	18	19

**Wnioski:**

- Zmodyfikowana wersja algorytmu redukuje liczbę przypisań
- Następuje porównania pozostają na prawie równe
- Łączna liczba operacji zmniejsza się z 40 do 37
- Modyfikacja okazuje się korzystna, redukując ogólną liczbę operacji

### 3.4 Analiza czasu

Testy na 100-elementowych tablicach:

Przypadek	Normalny	Zmodyfikowany
Losowe	9.2 $\mu$ s	10.4 $\mu$ s
Prawie posortowane	<1 $\mu$ s	<1 $\mu$ s
Odwrotne	15 $\mu$ s	19 $\mu$ s

**Wnioski:**

- Dla losowych: normalny szybszy
- Dla prawie posortowanych: tak samo
- Dla odwrotnych: normalny lepszy

Modyfikacja wykazuje, że w niektórych przypadkach oferuje redukcję operacji, ale kosztem czasu wykonania w scenariuszach pesymistycznych.

## 4 MERGE SORT

Klasyczny algorytm merge sort dzieli tablicę na dwie połowy, rekurencyjnie je sortuje, a następnie łączy (merge) w jedną posortowaną całość.

### Modyfikacja Merge sort z podziałem na trzy części:

W tej modyfikacji algorytm dzieli tablicę na trzy części zamiast dwóch, następnie rekurencyjnie sortuje każdą z nich ,a na końcu łączy trzy posortowane fragmenty w jedną całość.

```
void SortMerge(int p, int k)
{
    porownania +=1;
    if (p<k)
    {
        int jedtrz = p+(k-p)/3;
        int dwitrz = p+2*(k-p)/3;
        przypisania += 2;
        SortMerge(p,jedtrz);
        SortMerge(jedtrz+1,dwitrz);
        SortMerge(dwitrz+1,k);
        Merge(p,jedtrz,dwitrz,k);
    }
}
```

Jest to jeden z najważniejszych fragmentów tego algorytmu. Na początku funkcja sprawdza, czy dany fragment można jeszcze podzielić. Następnie dzieli tablicę na trzy części i rekurencyjnie sortuje każdą z nich. Po zakończeniu tych wywołań wykonuje funkcję Merge, która scalą trzy posortowane części w jedną całość.

### 4.1 Porównanie Orginalnego Merge z Zmodyfikowanym

**Normalny Merge Sort:**

```
2 14 5 1 33 85
1 2 5 14 33 85
w tym algorytmie masz 73 przypisan
w tym algorytmie masz 69 porownan
```

**Zmodyfikowany Merge Sort:**

```
2 14 5 1 33 85
1 2 5 14 33 85
w tym algorytmie masz 68 przypisan
w tym algorytmie masz 54 porownan
```

Moja wersja ma mniej operacji o 5 przypisań i 15 porównań mniej.

	Przypisania	Porownania
Normalny	73	69
Zmodyfikowany	68	54

## 4.2 Czas wykonania

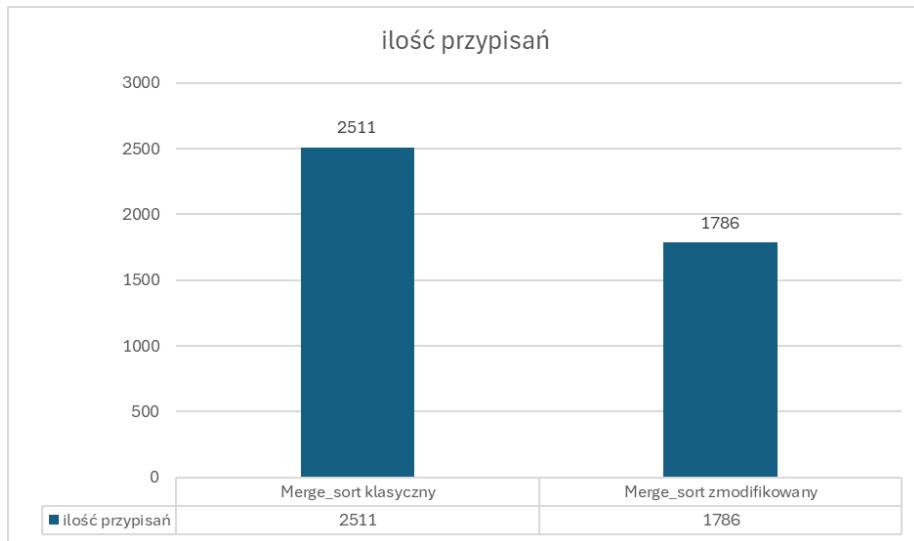
Średni czas dla 100 elementów:

- Normalny Merge Sort: 14.1  $\mu$ s
- Mój Merge Sort: 13.7  $\mu$ s

Przypadek	Normalny	Zmodyfikowany
Losowe	14 $\mu$ s	14.1 $\mu$ s
Prawie posortowane	11 $\mu$ s	10 $\mu$ s
Odwrotne	10 $\mu$ s	19 $\mu$ s

Moja wersja jest szybsza o 0.4  $\mu$ s.(Choć patrzenie sie na tak mała próbka może być zmylne o ile jest szybsze czy wogóle jest)

Różnica pomiędzy obiema wersjami jest wyraźnie widoczna przy analizie liczby przypisań.



Rysunek 1: Porównanie liczby przypisań w algorytmach Merge Sort

Jak widać na wykresie, nawet dla 10 list 100-elementowych zmodyfikowany Merge Sort wykonuje około **2/3 przypisań** w porównaniu do wersji oryginalnej. Oznacza to, że nasz algorytm charakteryzuje się **znacznie mniejszą liczbą operacji przypisania**.

W kwestii czasu wykonania przy tak małej próbce testowej trudno wyciągać wnioski,

## 5 HEAP SORT

Heap Sort polega na zbudowaniu z tablicy struktury kopca, gdzie rodzic jest zawsze większy od swoich dzieci. Następnie wielokrotnie usuwa się korzeń kopca (największy element) i odtwarza strukturę kopca. Elementy usuwane z kopca są umieszczane w posortowanej części tablicy. Inaczej też jest nazywany kopcem binarnym

**Modyfikacja z kopca binarnego na ternarne:**

Główna różnica polega na strukturze kopca - wersja binarna ma 2 dzieci dla każdego węzła, podczas gdy ternarna ma 3 dzieci. Dzięki temu kopiec ternarny jest niższy, co redukuje liczbę iteracji w głównej pętli. Jednak w procedurze Heapify potrzeba więcej porównań (3 zamiast 2) do znalezienia największego dziecka.

### 5.1 Ciekawy element kodu

Przedstawiony fragment poniżej moglibyśmy nazwać silnikiem Heap Sort. Na początku używając funkcji **Build\_Heap** potem od ostatniego elementu bierze, zamienia się z największym, który jest na samym początku kopca i największy element kasujemy z kopca i naprawiamy kopiec i powtarzamy.

```
void HeapSort(int heapsize)
{
    Build_Heap(heapsize);
    for(int i = rozmiar - 1; i >= 1; i--)
    {
        int temp = A[0];
        A[0] = A[i];
        A[i] = temp;
        przypisania += 3;
        heapsize -=1;
        Heapify(0, heapsize);
    }
}
```

## 5.2 Porównanie Orginalnego Heap Sort z Zmodyfikowanym

Zmodyfikowany Heap Sort:

```
2 14 5 1 33 85
1 2 5 14 33 85
w tym algorytmie masz 45 przypisan
w tym algorytmie masz 73 porównan
```

Klasyczny Heap Sort:

```
2 14 5 1 33 85
1 2 5 14 33 85
w tym algorytmie masz 53 przypisan
w tym algorytmie masz 67 porównan
```

	Przypisania	Porównania
Zmodyfikowany	45	73
Originalny	53	67

Mój zmodyfikowany Heap Sort z kopcem trójkowym ma mniej operacji o **8 przypisań** mniej i, o **6 przyrównań** więcej. Kopiec ternarny okazał się bardziej wydajny dla testowanych danych.

## 5.3 Czas wykonania

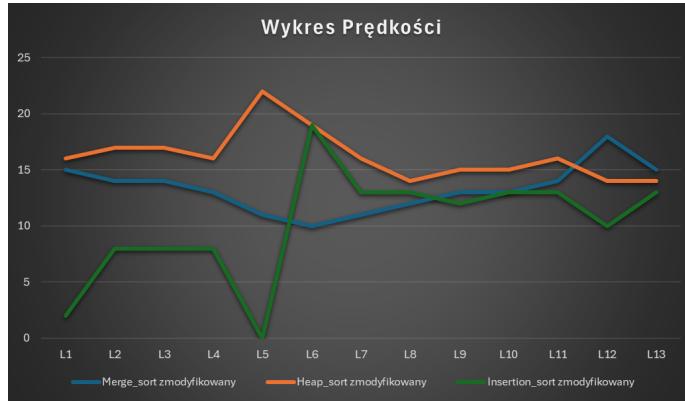
Średnie wyniki dla 10 list 100-elementowych:

	Zmodyfikowany	Normalny
Przypisania	1803	2421
Porównania	2875	2927
Czas	15.4 µs	13 µs

Mój algorytm działa trochę wolniej na podstawie tych danych, mimo że wykonuje mniej operacji (przypisań + porównań). Różnica w czasie jest jednak niewielka, więc trudno jednoznacznie stwierdzić, czy rzeczywiście jest wolniejszy, czy może to wynika ze zbyt małej liczby danych testowych.

## 6 Przyrównanie zmodyfikowanych algorytmów

Należy zaznaczyć, że analiza opiera się na stosunkowo małej próbce danych, co może wpływać na wiarygodność ostatecznych wniosków.

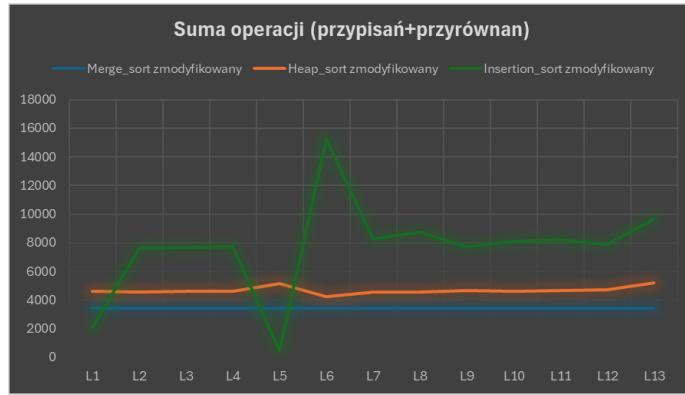


## 6.1 Który najszybszy

Na wykresie przedstawiono czasy wykonania dla każdej z list 100-elementowych, co pozwala określić, który ze zmodyfikowanych algorytmów jest najszybszy.

Z analizy wynika, że dla danych o takiej wielkości zmodyfikowany Insertion Sort jest najszybszy w większości przypadków. Co ciekawe, gdy lista jest odwrotnie posortowana (L6), zmodyfikowany Insertion Sort sortuje dłużej od zmodyfikowanego Merge Sorta. Natomiast dla list prawie posortowanych (L5) Insertion Sort radzi sobie znakomicie, wykonując sortowanie niemal natychmiast.

Można zatem stwierdzić, że dla testowanej wielkości danych najszybszy jest zmodyfikowany Insertion Sort, jednak jego efektywność jest uzależniona od rodzaju otrzymanej listy. Na drugim miejscu plasuje się zmodyfikowany Merge Sort, który można pochwalić za stabilność działania. Najgorzej w testach wyypadł zmodyfikowany Heap Sort.



## 6.2 Który ma najmniejszą ilość operacji

Z analizy wynika, że Merge Sort ma przeważnie najmniej operacji i jest stabilny. Heap Sort jest zazwyczaj na drugim miejscu utrzymując się delikatnie nad Mergem. Insertion Sort tylko dla prawie posortowanych danych oraz dla L1 ma mniej operacji niż Insertion Sort i Merge Sort, w pozostałych przypadkach wypada najgorzej.

## 6.3 Podsumowanie efektywności algorytmów

Na podstawie przeprowadzonych testów można stwierdzić, że wybór najlepszego algorytmu zależy od kryterium oceny. Pod względem prędkości wykonania zwycięzcą jest zmodyfikowany Insertion Sort, jednak jego wydajność jest niestabilna i zależy od charakteru danych. W kategorii liczby operacji bezkonkurencyjny okazuje się zmodyfikowany Merge Sort, oferujący zarówno niskie wartości jak, przewidywalność i stabilność. Podobnie zmodyfikowany Heap Sort, choć generuje nieco więcej operacji niż Merge Sort, zachowuje w miarę stabilny czas wykonania i oferuje bardzo dobrą przewidywalność niezależnie od charakteru danych wejściowych.