Sprawozdanie Lista 1

Jakub Okła 287337

28 października 2025

1 Wprowadzenie

Celem ćwiczenia była analiza działania trzech klasycznych algorytmów sortowania: Insertion Sort, Merge Sort oraz Heap Sort, a także ich zmodyfikowanych wersji. W każdej implementacji dodane zostały liczniki porównań oraz przypisań, co pozwoliło porównać koszty operacyjne algorytmów dla danych o różnych wielkościach.

Badane algorytmy reprezentują różne sposoby sortowania: sortowanie przez wstawianie (INSERTION_SORT), sortowanie przez scalanie (MERGE_SORT) oraz sortowanie oparte na strukturze kopca (HEAP_SORT).

2 Opis Algorytmów

2.1 INSERTION_SORT

2.1.1 opis

Iteracyjnie pobiera kolejne elementy i wstawia je w odpowiednie miejsce w części już posortowanej, dobra dla prawie posortowanych danych.

2.1.2 modyfikacja

Zamiast wstawiać pojedynczy element, pobierane są dwa kolejne elementy, najpierw są porównywane między sobą, a następnie wstawiane partią do odpowiednich pozycji.

2.1.3 złożoność

- $\mathcal{O}(n^2)$ (najgorszy przypadek)
- $\mathcal{O}(n)$ (najlepszy przypadek)

2.2 MERGE_SORT

2.2.1 opis

Dzieli dane na 2 części, sortuje rekurencyjnie i scala, stabilny. Zysk przy dużych danych, o ile implementacja scalania jest efektywna.

2.2.2 modyfikacja

Tablica dzielona jest na 3 podtablice, po czym scala się trzy posortowane ciągi. Liczba poziomów rekursji maleje, ale operacja scalania staje się bardziej kosztowna.

2.2.3 złożoność

- $\mathcal{O}(n)$
- $O(n \log_3 n)$ (modyfikacja)

2.3 HEAP_SORT

2.3.1 opis

Buduje kopiec z danych, a następnie iteracyjnie usuwa element maksymalny (korzeń) i umieszcza go na końcu tablicy. Dzięki utrzymywaniu struktury kopca zapewnia stały czas dostępu do największego elementu.

2.3.2 modyfikacja

każdy węzeł ma 3 dzieci, co oznacza mniejsza wysokość kopca, równą $\log_3 n$. Jednak przy wybieraniu dziecka pojawia się więcej porównań

2.3.3 złożoność

- $\mathcal{O}(n \log(n))$ (gwarantowana)

3 Pomiary

Tabela 1: Liczba porównań (COMP) dla różnych algorytmów sortowania

Algorytm	**n = 7**	**n = 15**	**n = 100**	**n = 1000**
Insertion Sort	16	74	2615	121073
Insertion Sort (mod.)	14	71	1831	81864
Merge Sort	14	43	539	5693
Merge Sort (mod.)	15	52	665	7001
Heap Sort	19	69	1033	10956
Heap Sort (mod.)	23	75	998	10675

Tabela 2: Liczba przypisań (ASS) dla różnych algorytmów sortowania

Algorytm	**n = 7**	**n = 15**	**n = 100**	**n = 1000**
Insertion Sort	32	106	2817	122456
Insertion Sort (mod.)	47	170	3758	164243
Merge Sort	52	146	1542	14510
Merge Sort (mod.)	40	114	1053	9372
Heap Sort	42	135	1764	17730
Heap Sort (mod.)	36	117	1224	12369

4 Wnioski

- **Insertion Sort** jest efektywny tylko dla bardzo małych zbiorów danych lub danych prawie posortowanych. Wraz ze wzrostem n liczba porównań i przypisań rośnie bardzo szybko (dla $n = 1000$ ponad 120 tysięcy porównań). Modyfikacja wstawiająca dwa elementy jednocześnie pozwala ograniczyć liczbę porównań (ok. 30% mniej niż klasyczna wersja), jednak odbywa się to kosztem większej liczby przypisań. Daje to korzyść jedynie w przypadkach, gdy dane są częściowo uporządkowane.

- **Merge Sort** charakteryzuje się stabilnym i przewidywalnym wzrostem liczby operacji wraz z rozmiarem danych. Złożoność $O(n \log n)$ przekłada się na znacznie mniejszy wzrost liczby porównań i przypisań w porównaniu z algorytmami wstawiania. Modyfikacja dzieląca tablicę na trzy części zmniejsza głębokość rekursji i nieznacznie redukuje liczbę przypisań dla większych n, jednak zwiększa złożoność samej fazy scalania. Różnice są widoczne dopiero dla dużych tablic.

- **Heap Sort** jest nieco mniej wydajny od Merge Sort, ale oferuje stałą złożoność pamięciową i brak potrzeby alokacji pomocniczych tablic. Wersja **trójkowa (modified)** zmniejsza wysokość kopca, co teoretycznie redukuje liczbę poziomów rekursji, lecz zwiększa koszt pojedynczego porównania w funkcji `HEAPIFY`. W efekcie liczba porównań jest nieco większa, a liczba przypisań mniejsza, co daje podobny całkowity koszt.

- Ogólnie, **Merge Sort** i jego modyfikacja są najbardziej efektywne dla dużych zbiorów danych, natomiast **Insertion Sort** pozostaje najlepszy dla bardzo małych lub wstępnie uporządkowanych danych. **Heap Sort** plasuje się pomiędzy nimi pod względem wydajności. Najważniejszym czynnikiem wpływającym na rzeczywisty czas działania jest charakter danych wejściowych oraz koszty operacji pamięciowych i rekurencyjnych.