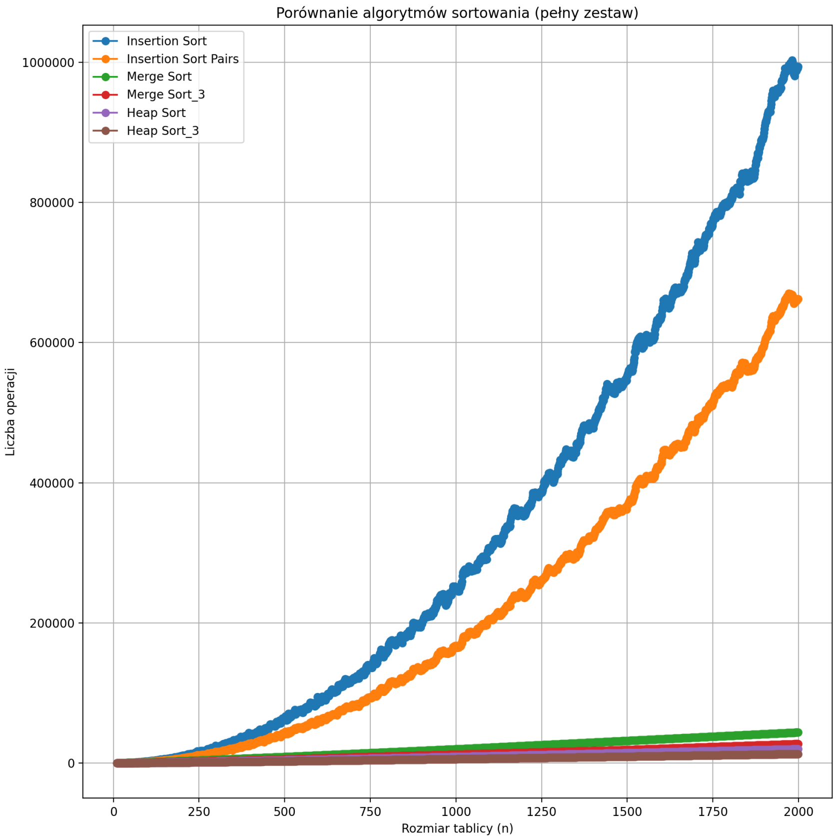
Sprawozdanie: Analiza wybranych algorytmów sortowania

(Insertion Sort, Insertion Sort Pairs, Merge Sort, Merge Sort 3, Heap Sort, Heap Sort 3)

25 października 2025

1 Wstęp

Celem pracy była analiza sześciu zaimplementowanych algorytmów sortowania: *Insertion Sort*, jego modyfikacji wstawiającej elementy parami (*Insertion Sort Pairs*), *Merge Sort* (klasyczny dwudzielny), jego wariantu trójdzielnego (*Merge Sort 3*), a także *Heap Sort* w wersji binarnej i ternarnej. Dla każdego algorytmu zliczano liczbę elementarnych operacji (przestawień), a następnie zebrano wyniki dla rozmiarów tablic $n = 10, \ldots, 1999$. Generator danych używa stałego ziarna `srand(10)`, dzięki czemu każde uruchomienie daje identyczne wektory wejściowe. Wszystkie pomiary wykonano tym samym programem testującym.



Rysunek 1: Liczba operacji w funkcji rozmiaru wejścia — porównanie wszystkich metod.

2 Opis algorytmów

- **Insertion Sort** — sortowanie przez wstawianie, złożoność średnia i pesymistyczna $O(n^2)$. Dobre dla prawie posortowanych danych.

- **Insertion Sort Pairs** — modyfikacja, w której do listy wstawia się jednocześnie parę elementów (po wcześniejszym ich uporządkowaniu). Teoretycznie nadal $O(n^2)$, w praktyce redukuje liczbę przesunięć.
- **Merge Sort** — klasyczne dziel i zwyciężaj; rekurencyjnie dzieli tablicę na dwie połowy. Złożoność $O(n \log n)$.
- **Merge Sort 3** — wariant dzielący na trzy części i łączący. Teoretycznie również $O(n \log n)$, lecz z inną stałą.
- **Heap Sort (binarny)** — budowa kopca. Złożoność $O(n \log n)$.
- **Heap Sort 3** — analogiczny do wersji binarnej, lecz kopiec ternarny (trzech potomków); mniejsza wysokość kopca, ale droższe pojedyncze `heapify`.

3 Najciekawsze fragmenty kodu: *Merge Sort*

Poniżej pokazano dwa kluczowe fragmenty: funkcję sklejającą i rekurencyjne dzielenie. Pierwsza z nich odpowiada za liniowe łączenie dwóch posortowanych list (lewego i prawego) w jeden przedział tablicy wejściowej. Zmienna `number` służy jako licznik operacji przestawień wykorzystywany w analizie.

Funkcja `merge` (fragment z `merge_sort.cpp`):

Listing 1: Sklejanie dwóch przedziałów w Merge Sort.

```
int merge(int list[], int begin, int middle, int end, int number) {
    int len_left = middle - begin + 1;
    int len_right = end - middle;
    int left_list[len_left];
    int right_list[len_right];
    for (int i = begin; i < begin+len_left; i++) {
        left_list[i-begin] = list[i];
        number++;
    }
    for (int i = middle; i < middle+len_right; i++) {
        right_list[i-middle] = list[i+1];
        number++;
    }
    int left_index = 0;
    right_index = 0;
    for (int i = begin; i <= end; i++) {
      if ((left_index < len_left and left_list[left_index] <= right_list[
         right_index]) or right_index >= len_right){
        list[i] = left_list[left_index];
        left_index++;
        number++;

      }
      else {
        list[i] = right_list[right_index];
        right_index++;
        number++;

      }

    }
    return number;
}
```

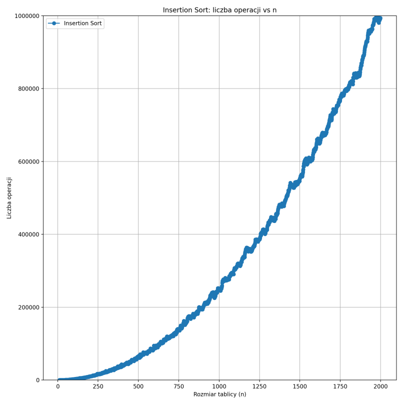
Podział i wywołania rekurencyjne:

Listing 2: Rekurencyjny podział i łączenie.

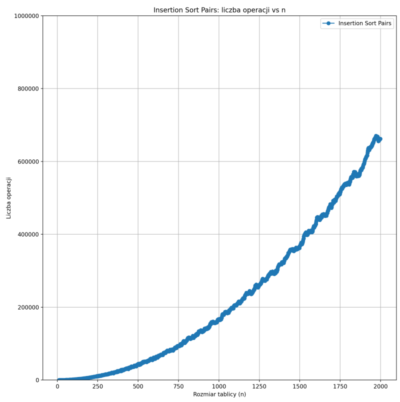
```
int merge_sort(int list[], int begin, int end, int number) {
    if (begin < end) {
        int middle = floor((begin+end)/2);
        number = merge_sort(list, begin, middle, number);
        number = merge_sort(list, middle+1, end, number);
        number = merge(list, begin, middle, end, number);
    }
    return number;
}
```

4 Porównania i wyniki

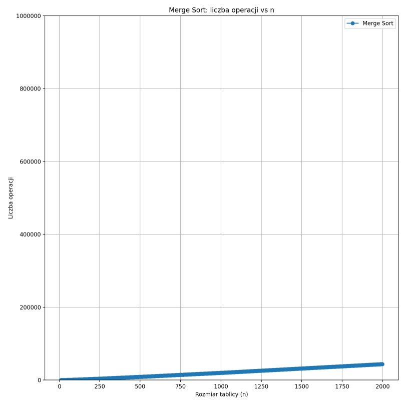
Na rys. 1 widać wyraźny rozjazd między metodami $O(n^2)$ (*Insertion*) i $O(n \log n)$ (rodzina *Merge*/*Heap*). Aby lepiej zobaczyć indywidualne przebiegi, poniżej zamieszczono mini-wykresy (oryginały zapisane w plikach PNG).



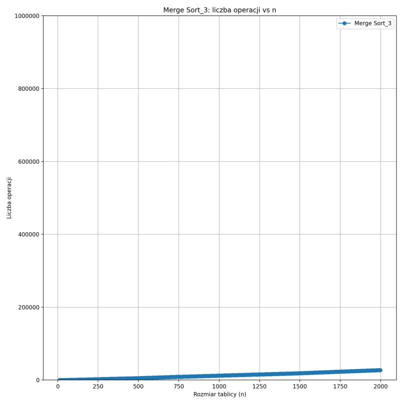
(a) Insertion Sort



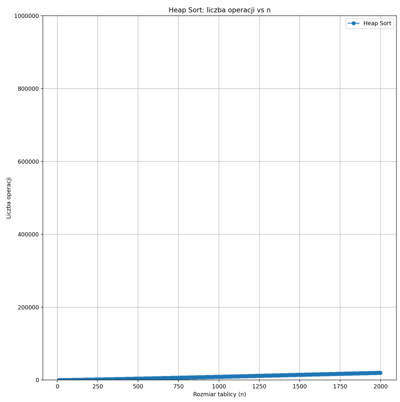
(b) Insertion Sort Pairs



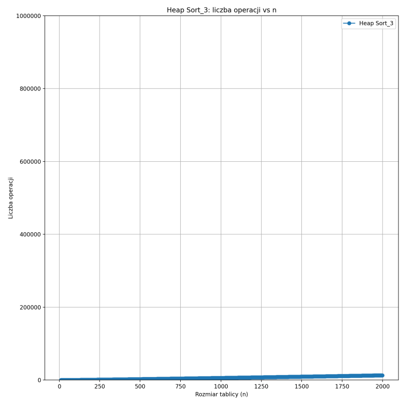
(c) Merge Sort



(d) Merge Sort 3



(e) Heap Sort



(f) Heap Sort 3

Rysunek 2: Mini-wykresy: liczba operacji w funkcji n dla każdego algorytmu.

4.1 Tabela porównawcza złożoności i własności

Algorytm	Złożoność średnia	Złożoność pesymistyczna
Insertion Sort	$O(n^2)$	$O(n^2)$
Insertion Sort Pairs	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Merge Sort 3	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Heap Sort 3	$O(n \log n)$	$O(n \log n)$

4.2 Komentarz do wyników

- **Insertion vs Insertion Pairs:** Obie krzywe rosną kwadratowo; wariant parowy redukuje liczbę przesunięć o stały czynnik, ale nie zmienia rzędu złożoności.
- **Merge Sort klasyczny vs Merge Sort 3:** Wariant trójdzielny osiąga *nieco mniej* operacji dla dużych n (mniej poziomów rekurencji), ale cena to bardziej skomplikowane scalanie.
- **Heap Sort binarny vs Heap Sort ternarny:** Kopiec trójdzielny ma mniejszą wysokość, jednak `heapify` musi porównać do trzech dzieci. W praktyce różnice są niewielkie; dla naszych danych *Heap Sort 3* zwykle wykonuje mniej operacji.

5 Wnioski

1. Dla dużych n należy preferować algorytmy $O(n \log n)$: *Merge Sort* i *Heap Sort*. Wśród nich warianty modyfikowane do trójek bywają korzystne, ale zysk zależy od tego czy liczymy operacji oprócz przepisywań.
2. *Insertion Sort* pozostaje dobrym wyborem dla bardzo małych lub prawie posortowanych danych; wariant wstawiania parami może ograniczyć liczbę przesunięć, lecz nie przełamuje bariery $O(n^2)$.
3. Zliczanie elementarnych operacji potwierdziło trendy teoretyczne: przebiegi dla *Merge*/*Heap* są bliskie $n \log n$, a dla *Insertion* — kwadratowe.