Algorytmy i struktury danych lista 1

Bartosz Polak

27.10.2025

1 Wprowadzenie

Celem listy jest analiza oraz porównanie efektywności różnych algorytmów sortowania zaimplementowanych w C++. Mierzone były:

- liczba porównań elementów,
- liczba przypisań elementów.

Dla każdego algorytmu przeprowadzono testy na tablicach o różnych rozmiarach, a wyniki zestawiono w tabelach oraz na wykresach.

2 Analizowane algorytmy

- **Insertion Sort** oraz jego modyfikacja polegajaca na wstawianiu "na raz"dwóch elementó tablicy,
- **Merge Sort** 2-dzielny i 3-dzielny,
- **Heap Sort** binarny i ternarny.

3 Najciekawsze fragmenty kodu

3.1 Algorytm sortowania przez wstawianie

Poniżej przedstawiono fragment klasycznego algorytmu sortowania przez wstawianie wraz z liczeniem operacji:

```
for (int i = 1; i < n; i++) {
    x = A[i];
    przypisania++;
    j = i - 1;
    przypisania++;
    while (true) {
        porownania++;
        if (j < 0) break;
        porownania++;
        if (A[j] > x) {
            A[j + 1] = A[j];
```

```
            przypisania++;
            j--;
            przypisania++;
        } else break;
    }
    A[j + 1] = x;
    przypisania++;
}
```

Listing 1: Fragment funkcji INSERTION_SORT

Druga wersja, `INSERTION_SORT2`, wprowadza jednoczesne wstawianie dwóch elementów, co potencjalnie mogłoby redukować liczbe porównań i przypisań.

3.2 Sortowanie przez scalanie 2-dzielne i 3-dzielne

Fragment implementacji funkcji scalajacej dla sortowania 3-dzielnego:

```
while (i < n1 || j < n2 || m < n3) {
    int min_val = 1000000000;
    int min_index = 0;
    if (i < n1 && L1[i] < min_val) { min_val = L1[i]; min_index =
       red↪  1; }
    if (j < n2 && L2[j] < min_val) { min_val = L2[j]; min_index =
       red↪  2; }
    if (m < n3 && L3[m] < min_val) { min_val = L3[m]; min_index =
       red↪  3; }
    if (min_index == 1) A[r++] = L1[i++];
    else if (min_index == 2) A[r++] = L2[j++];
    else if (min_index == 3) A[r++] = L3[m++];
}
```

Listing 2: Fragment funkcji MERGE3

Algorytm 3-dzielny cechuje sie wieksza złożonościa implementacyjna, ale w praktyce może lepiej równoważyć rekurencje. Aby zmodyfikować merge sort 2-dzielny do 3-dzielnego bez skomplikowanego zagnieżdżania "if", musiałem wprowadzić zmienne min_val i min_index, co jest ciekawym rozwiazaniem problemu.

3.3 Sortowanie przez kopcowanie binarne i trójkowe

Porównano wersje klasyczna kopca binarnego z wersja kopca trójkowego:

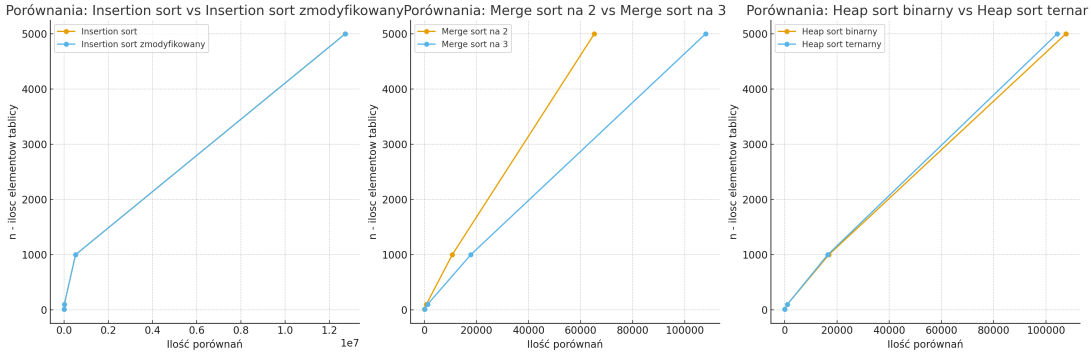
```
int largest = i;
int c1 = 3 * i + 1;
int c2 = 3 * i + 2;
int c3 = 3 * i + 3;

if (c1 < n && A[c1] > A[largest]) largest = c1;
if (c2 < n && A[c2] > A[largest]) largest = c2;
if (c3 < n && A[c3] > A[largest]) largest = c3;

if (largest != i) {
```

```
11      swap(A[i], A[largest]);
12      przypisania += 3;
13      heapify_ternary(A, n, largest, porownania, przypisania);
14  }
```

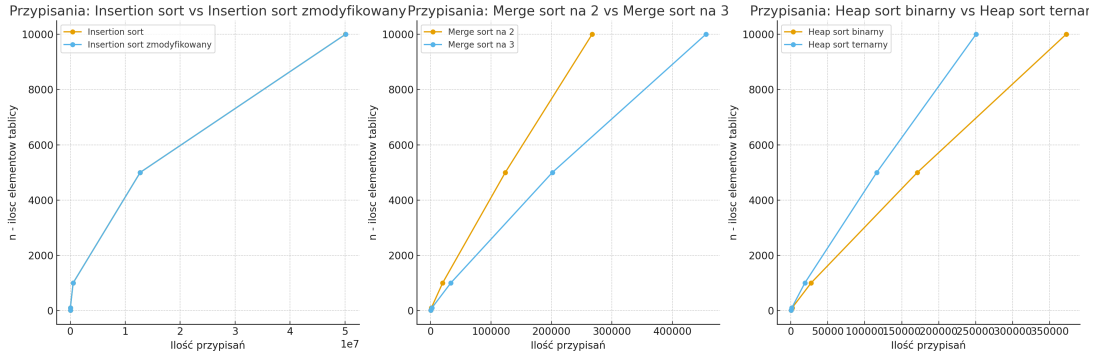
Listing 3: Fragment funkcji heapify_ternary

4 Porównanie wyników działania algorytmów

Dla różnych rozmiarów tablic ($n = 10, 100, 1000, 5000, 10000$) zmierzono liczbe porównań i przypisań. Poniżej przedstawiono zestawienie wyników w postaci wykresów.



Rysunek 1: Porównanie liczby porównań dla różnych algorytmów.



Rysunek 2: Porównanie liczby przypisań dla różnych algorytmów.

5 Wnioski

- Klasyczny **Insertion Sort** jest najmniej efektywny dla dużych zbiorów danych, jego złożoność $O(n^2)$ powoduje gwałtowny wzrost liczby operacji.
- Modyfikacja **Insertion Sort** przynosi nieznaczaca różnice, niezauważalna na wykresie.
- **Merge Sort 3-dzielny** generuje wiecej operacji przypisań i porównań niż klasyczny Merge Sort 2-dzielny.

- **Heap Sort** binarny oraz trójkowy maja porównywalne wyniki w ilości porównań, gdzie wersja binarna wykonuje mało znaczaco mniej przypisań, ale też bardziej znaczaco wiecej przypisań.
- Dla dużych danych najlepsze rezultaty osiagały algorytmy o złożoności $O(n \log n)$ – **Merge Sort** oraz **Heap Sort**.

6 Podsumowanie

Wszystkie zaimplementowane algorytmy poprawnie sortuja dane wejściowe, jednak różnia sie efektywnością. Z punktu widzenia praktycznego — dla dużych tablic — najbardziej opłacalne sa algorytmy typu **Merge Sort** i **Heap Sort**, natomiast **Insertion Sort** pozostaje użyteczny jedynie dla bardzo małych zbiorów.