Raport AiSD lista 1

Amelia Dorożko

24 października 2024

Do analizy poszczególnych algorytmów sortujących, bazowałam na przykładowo 8 wygenerowanych listach o zadanych długościach oraz elementach z zakresu od 0 do 999.

1 Sortowanie przez wstawianie

1.1 `INSERTION_SORT`

W tej metodzie każda kolejna liczba z tablicy jest porównywana z elementami po jej lewej stronie i wstawiana na odpowiednie miejsce.

Długość tablicy (n)	**Liczba porównań**	**Liczba przypisań**
5	4	12
10	22	40
100	2615	2813
200	10045	10443
400	37341	38139
600	86800	87998
800	163804	165402
1000	249193	251191

Tabela 1: INSERTION_SORT

1.2 `MODIFIED_INSERTION_SORT`

Sortowanie przez wstawianie oraz jego zmodyfikowana wersja, jest algorytmem o złożoności czasowej $O(n^2)$ w najgorszym z możliwych wypadków.
`MODIFIED_INSERTION_SORT` okazuje się nie być najkorzystniejszym rozwiązaniem, co spowodowane jest przykładowo użyciem `swap` czy większą ilością zmiennych do porównania. Dodatkowo, wprowadzanie operacji `swap` dla par elementów może prowadzić do sytuacji, gdy elementy są przesuwane więcej razy niż w standardowej wersji.Zmodyfikowany algorytm lepiej radzi sobie z danymi, które są już w pewnym stopniu uporządkowane

```
    if (pierwszy > drugi) {
        swap(pierwszy, drugi);
        liczbaPrzypisan += 2;
    }
    int j = i - 1;
    while (j >= 0 && A[j] > pierwszy) {
        liczbaPorownan++;
        A[j+1] = A[j];
        liczbaPrzypisan++;
        j--;
    }
    A[j+1] = pierwszy;
    liczbaPrzypisan++;
```

W celu uzyskania pełnej poprawności algorytmu, należało rozważyć przypadek dla tablic o parzystej ilości elementów,

```
        if (n % 2 == 0) {
        liczbaPorownan++;
        int ostatni = A[n - 1];
        liczbaPrzypisan++;
        int j = n - 2;
        while (j >= 0 && A[j] > ostatni) {
            liczbaPorownan++;
            A[j + 1] = A[j];
            liczbaPrzypisan++;
            j--;
        }
        A[j + 1] = ostatni;
        liczbaPrzypisan++;
    }
```

Długość tablicy (n)	Liczba porównań	Liczba przypisań
5	4	14
10	23	43
100	2616	2834
200	10046	10491
400	37342	38222
600	86801	88139
800	163805	165599
1000	249194	251446

Tabela 2: MODIFIED_INSERTION_SORT

2 Sortowanie przez scalanie

2.1 MERGE_SORT

Klasyczny MERGE_SORT dzieli tablicę na dwie części, sortuje je rekurencyjnie, a następnie scala dwie posortowane części, co daje całkowitą złożoność $O(n \log(n))$.

Długość tablicy (n)	Liczba porównań	Liczba przypisań
5	16	32
10	43	86
100	771	1542
200	1743	3486
400	3887	7774
600	6175	12350
800	8575	17150
1000	10975	21950

Tabela 3: MERGE_SORT

2.2 MODIFIED_MERGE_SORT

W przypadku zmodyfikowanego `MERGE_SORT`, dzielimy tablicę na trzy części zamiast dwóch. Mimo że scalanie trzech tablic wymaga więcej porównań, mniejsza liczba poziomów rekurencyjnych sprawia, że całkowity koszt operacji może być niższy. Tutaj mamy więcej segmentów, co może prowadzić do lepszej wydajności, zwłaszcza dla dużych danych.

```
int s1 = p + (k - p) / 3;
int s2 = p + 2 * (k - p) / 3;
MODIFIED_MERGE_SORT(A, p, s1);
MODIFIED_MERGE_SORT(A, s1 + 1, s2);
MODIFIED_MERGE_SORT(A, s2 + 1, k);
MODIFIED_MERGE(A, p, s1, s2, k);
```

Również należy zwrócić uwagę poniższy fragment kodu:

```
    L[n1] = INT_MAX;
    M[n2] = INT_MAX;
    P[n3] = INT_MAX;
```

Umożliwia on algorytmowi efektywne zakończenie procesu łączenia, gdy wszystkie elementy zostały już przetworzone.

Długość tablicy (n)	Liczba porównań	Liczba przypisań
5	12	27
10	27	59
100	497	1053
200	1121	2363
400	2592	5462
600	3964	8292
800	5377	11189
1000	7177	14989

Tabela 4: MODIFIED_MERGE_SORT

3 Sortowanie przez kopcowanie

3.1 HEAP_SORT

HEAP_SORT tworzy kopiec binarny z tablicy, a następnie wielokrotnie usuwa największy element, który umieszcza na końcu tablicy, odbudowując kopiec dla pozostałych elementów, co łącznie daje złożoność $O(n \log(n))$.

Długość tablicy (n)	Liczba porównań	Liczba przypisań
5	13	27
10	47	86
100	1146	1823
200	2828	4403
400	6608	10131
600	10740	16362
800	15132	22936
1000	19751	29817

Tabela 5: HEAP _SORT

3.2 MODIFIED_HEAP_SORT

Modyfikacja algorytmu HEAP_SORT polega na wprowadzeniu kopca ternarnego.Mniejsza "głębokość" kopca oznacza, że operacje przesiewania MODIFIED_HEAPIFY muszą być wykonywane rzadziej, ponieważ mniejsze jest "drzewo", które algorytm musi przetworzyć. Warto wyróżnić sposób modyfikacji funkcji HEAPIFY, która obsługuje dodatkowych potomków:

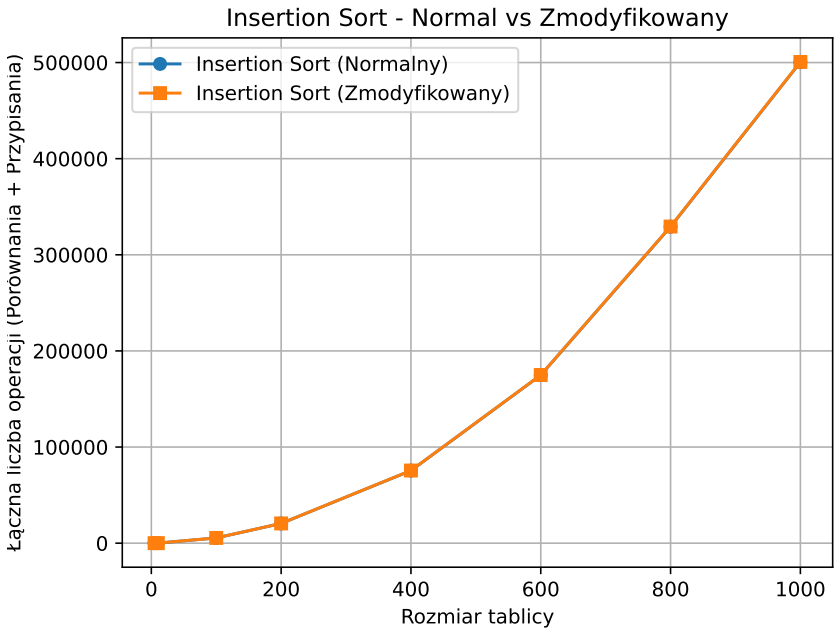
```
1
2       if (l < n && A[l] > A[largest])  {
3           liczbaPorownan++;
4           liczbaPrzypisan++;
5           largest = l;
6       }
7       if (m < n && A[m] > A[largest]) {
8           liczbaPorownan++;
9           liczbaPrzypisan++;
10          largest = m;
11      }
12      if (r < n && A[r] > A[largest]) {
13          liczbaPorownan++;
14          liczbaPrzypisan++;
15          largest = r;
16      }
17      if (largest != i) {
18          swap(A[i], A[largest]);
19          liczbaPorownan++;
20          liczbaPrzypisan += 2;
21          MODIFIED_HEAPIFY(A, n, largest);
22      }
```

Długość tablicy (n)	Liczba porównań	Liczba przypisań
5	10	22
10	32	63
100	849	1365
200	2039	3216
400	4655	7196
600	7728	11831
800	10903	16558
1000	14248	21504

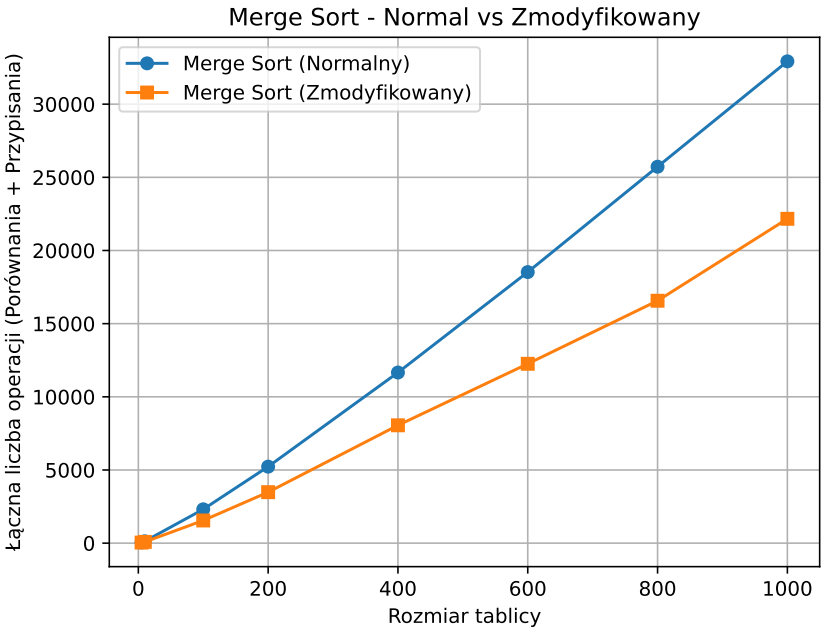
Tabela 6: MODIFIED_HEAP_SORT

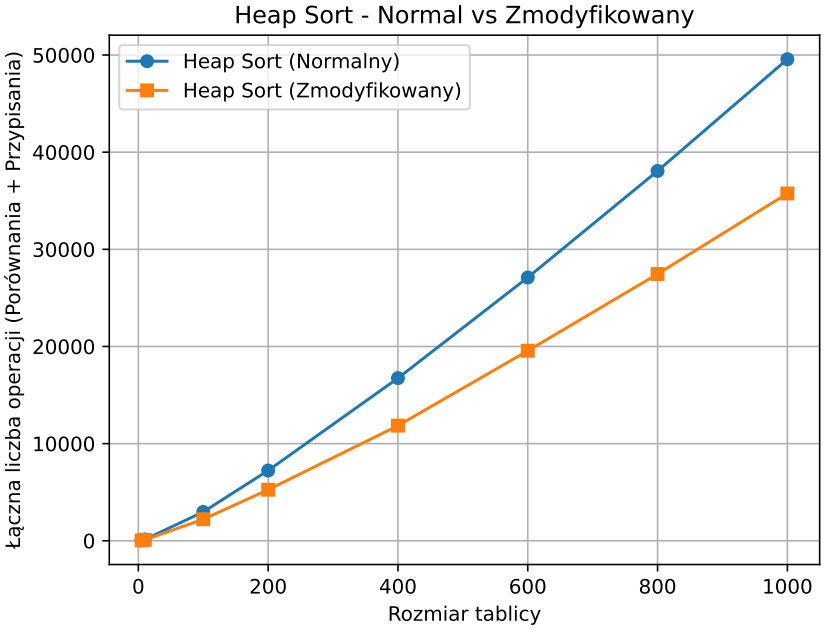
Porównanie klasycznych i zmodyfikowanych algorytmów sortowania

1. Przyrównując obie wersje algorytmu `INSERTON_MERGE`, łączna ilość operacji jest niemal identyczna, zarówno dla dużych jak i małych danych.

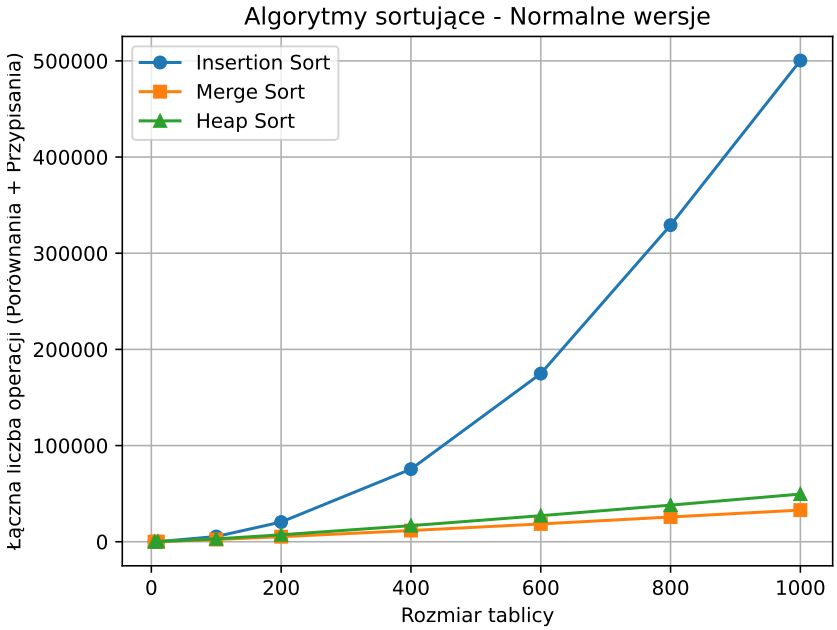


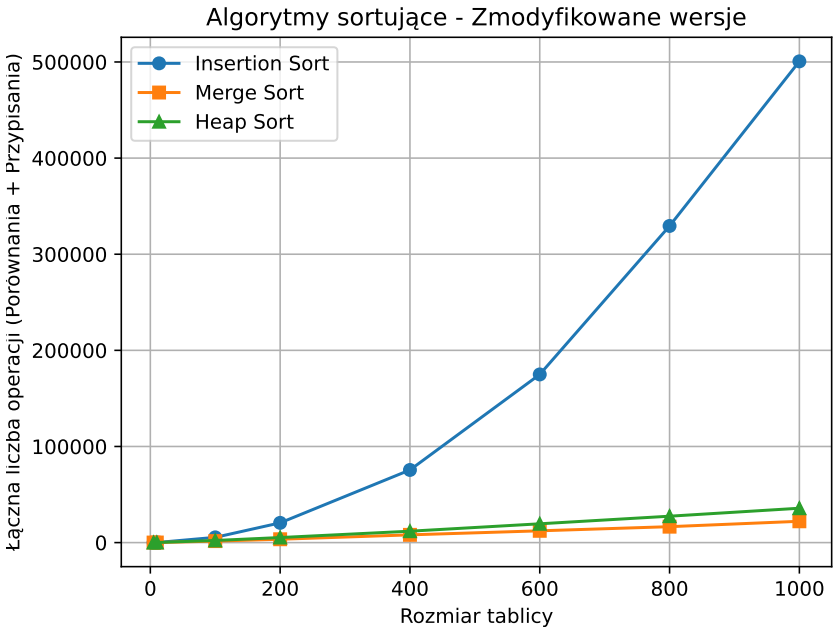
2. Kolejne dwie wersje modyfikacji, są korzystniejszym rozwiązaniem.


Merge Sort - Normal vs Zmodyfikowany
Merge Sort (Normalny)
Merge Sort (Zmodyfikowany)
Łączna liczba operacji (Porównania + Przypisania)
0
5000
10000
15000
20000
25000
30000
0
200
400
600
800
1000
Rozmiar tablicy


Heap Sort - Normal vs Zmodyfikowany
Heap Sort (Normalny)
Heap Sort (Zmodyfikowany)
Łączna liczba operacji (Porównania + Przypisania)
0
10000
20000
30000
40000
50000
0
200
400
600
800
1000
Rozmiar tablicy

3. Zarówno klasyczny `INSERTION_SORT`, jak i jego modyfikacja są najmniej korzystnym rozwiązaniem.





Poprzez porównanie poprzednio opisanych algorytmów sortujących, w przypadku danych o dużej objętości lub nieprzewidywalnych zestawów danych, algorytmy `MERGE_SORT` i `HEAP_SORT` będą bardziej optymalne, podczas gdy `INSERTION_SORT` będzie miał sens głównie dla niewielkich zbiorów, lub w pewnej części uporządkowanych.