Sprawozdanie AiSD Lista 1

Michał Kasjaniuk, 287392

October 2025

1 Wprowadzenie

Celem niniejszego sprawozdania jest analiza i porównanie algorytmów sortowania: Insertion Sort, Heap Sort oraz Merge Sort wraz z ich modyfikacjami. Algorytmy zostały ocenione pod względem liczby porównań, liczby przypisań oraz czasu wykonania dla różnych danych wejściowych.

2 Opis algorytmów

W tej sekcji zajmiemy się przedstawieniem zasotowanych algorytmów wraz z ich modyfikacjami.

2.1 Insertion Sort

Insertion Sort to prosty algorytm sortowania działający na zasadzie wstawiania elementów w odpowiednie miejsce w już posortowanej części tablicy.

```
void INSERTION_SORT(int A[], int n) {
    int x, j;
    for (int i = 1; i < n; i++) {
        x = A[i];
        j = i - 1;
        while (j >= 0 && A[j] > x) {
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = x;
    }
}
```

Wersja zmodyfikowana polega na sortowaniu dwóch elementów naraz. Najpierw wybieramy dwa elementy i porównujemy je ze sobą. Następnie mniejszy z nich wstawiamy w odpowiednie miejsce w już posortowanej części tablicy, korzystając z pętli while i dbając o to, aby nie wyjść poza granice tablicy. Po znalezieniu miejsca dla mniejszego elementu wykonujemy analogicznie wstawienie większego elementu. W przypadku, gdy rozmiar tablicy jest nieparzysty, ostatni element sortujemy standardowym algorytmem Insertion Sort.

```
void INSERTION_SORT_MODIFIED(int A[], int n) {
    int i, j, k, x, y, temp;
    for (i = 1; i < n; i += 2) {
        x = A[i];
        y = A[i - 1];
        if (y > x) {
            temp = x;
            x = y;
            y = temp;
        }
        k = i - 2;
        while (k >= 0 && A[k] > y) {
            A[k + 1] = A[k];
            k = k - 1;
        }
        A[k + 1] = y;
```

```
        j = i - 1;
        while (j >= 1 && A[j] > x) {
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = x;
    }
    if (n % 2 == 1) {
        i = n - 1;
        x = A[i];
        j = i - 1;
        while (j >= 0 && A[j] > x) {
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j + 1] = x;
    }
}
```

2.2 Merge Sort

Merge Sort jest algorytmem sortowania metodą „dziel i zwyciężaj". Tablica wejściowa jest rekurencyjnie dzielona na dwie (lub w modyfikacji – trzy) mniejsze części aż do pojedynczych elementów. Następnie części te są scalane w taki sposób, aby powstała uporządkowana tablica. Merge Sort jest stabilny i ma złożoność czasową O(n log n) w najlepszym, średnim i najgorszym przypadku, jednak wymaga dodatkowej pamięci na tablice pomocnicze.

```
    void MERGE(vector<int>& A, int p, int s, int k) {
    int n1 = s - p + 1;
    int n2 = k - s;
    vector<int> L(n1 + 1);
    vector<int> R(n2 + 1);
    for (int i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[s + 1 + j];
    L[n1] = numeric_limits<int>::max();
    R[n2] = numeric_limits<int>::max();
    int i = 0, j = 0;
    for (int l = p; l <= k; l++) {
        if (L[i] <= R[j]) {
            A[l] = L[i];
            i++;
        } else {
            A[l] = R[j];
            j++;
        }
    }
}

void MERGE_SORT(vector<int>& A, int p, int k) {
    if (p < k) {
        int s = floor((p + k) / 2); // punkt podziału
        MERGE_SORT(A, p, s);
        MERGE_SORT(A, s + 1, k);
        MERGE(A, p, s, k);
    }
}
```

Merge Sort zmodyfikowany - w tej wersji algorytmu tablica wejściowa jest dzielona na trzy części zamiast dwóch. Każda część jest rekurencyjnie sortowana, a następnie trzy uporządkowane podtablice są scalane w jedną. Dzięki podziałowi na trzy części liczba poziomów rekurencji jest mniejsza niż w klasycznym Merge Sort, co może skutkować mniejszą liczbą przypisań, choć liczba porównań może wzrosnąć. Algorytm nadal zachowuje stabilność i działa w czasie O(n log n), przy czym logarytm liczony jest w podstawie 3 zamiast 2.Poniższy fragment to funkcja Merge Modified, która stanowi serce algorytmu Merge Sort Modified.

```
    void MERGE_MODIFIED(vector<int>& A, int p, int s1, int s2, int k) {
    int n1 = s1 - p + 1;
    int n2 = s2 - s1;
    int n3 = k - s2;
    vector<int> L(n1 + 1);
    vector<int> M(n2 + 1);
    vector<int> R(n3 + 1);

    for (int i = 0; i < n1; i++) {
        L[i] = A[p + i];
    }
    for (int j = 0; j < n2; j++) {
        M[j] = A[s1 + 1 + j];
    }
    for (int l = 0; l < n3; l++) {
        R[l] = A[s2 + 1 + l];
    }
    L[n1] = numeric_limits<int>::max();
    M[n2] = numeric_limits<int>::max();
    R[n3] = numeric_limits<int>::max();
    int i = 0, j = 0, l = 0;
    for (int m = p; m <= k; m++) {
        if (L[i] <= M[j] && L[i] <= R[l]) {
            A[m] = L[i];
            i++;
        }
        else {
            if (M[j] <= L[i] && M[j] <= R[l]) {
                A[m] = M[j];
                j++;
            }
            else {
                A[m] = R[l];
                l++;
            }
        }
    }
}
```

2.3 Heap Sort

Heap Sort to algorytm sortowania oparty na strukturze danych kopca binarnego. Proces sortowania polega na dwóch głównych etapach:

1. Budowanie kopca: Tworzymy kopiec maksymalny z danych wejściowych, tak aby największy element znajdował się w korzeniu.
2. Sortowanie: Usuwamy korzeń kopca (największy element) i umieszczamy go na końcu tablicy, a następnie przywracamy własność kopca dla pozostałej części tablicy. Proces powtarzamy, aż cała tablica zostanie posortowana.

Heap Sort charakteryzuje się złożonością czasową O(n log n) w każdym przypadku (najlepszym, średnim i najgorszym).

```
int LEFT(int i) {
    return 2 * i + 1;
}
int RIGHT(int i) {
    return 2 * i + 2;
}
void HEAPIFY(vector<int>& A, int heap_size, int i) {
    int l = LEFT(i);
    int r = RIGHT(i);
    int largest;
    if (l < heap_size && A[l] > A[i])
        largest = l;
    else
        largest = i;
    if (r < heap_size && A[r] > A[largest])
        largest = r;
    if (i != largest) {
        swap(A[i], A[largest]);
        HEAPIFY(A, heap_size, largest);
    }
}
void BUILD_HEAP(vector<int>& A) {
    int n = A.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        HEAPIFY(A, n, i);
    }
}
void HEAP_SORT(vector<int>& A) {
    int n = A.size();
    BUILD_HEAP(A);
    int heap_size = n;
    for (int i = n - 1; i >= 1; i--) {
        swap(A[0], A[i]);
        heap_size--;
        HEAPIFY(A, heap_size, 0);
    }
}
```

W wersji zmodyfikowanej kopiec jest ternarny, czyli każdy węzeł ma trzech potomków zamiast dwóch. Budowanie kopca oraz przywracanie własności kopca przebiega podobnie jak w klasycznej wersji, lecz przy każdej operacji porównujemy trzy dzieci zamiast dwóch, co może zmniejszać wysokość kopca i przyspieszać niektóre operacje sortowania. W gruncie rzeczy najwiekszą modyfikację możemy zauważyć w funkcji Heapify. Natomiast reszta funkcji typu void jest bliźniaczo podobna.

```
    void HEAPIFY_MODIFIED(vector<int>& A, int heap_size, int i) {
    int l = LEFT_MODIFIED(i);
    int m = MIDDLE_MODIFIED(i);
    int r = RIGHT_MODIFIED(i);
    int largest;
    if (l < heap_size && A[l] > A[i]) {
        largest = l;
    }
    else {
        largest = i;
    }
    if (m < heap_size && A[m] > A[largest]) {
        largest = m;
    }
    if (r < heap_size && A[r] > A[largest]) {
        largest = r;
    }
    if (i != largest) {
        int temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        HEAPIFY_MODIFIED(A, heap_size, largest);
    }
}
```

3 Porównanie algorytmów

W tej sekcji zajmiemy się porównywaniem algortymów: Insertion Sort, Heap Sort oraz Merge Sort wraz z ich modyfikacjami. Analiza została przeprowadzona pod kątem liczby wykonanych porównań, liczby przypisań oraz czasu wykonania dla różnych danych wejściowych (pesymistycznych, optymistycznych, losowych, "dużych" danych). Celem jest ocena efektywności poszczególnych algorytmów oraz sprawdzenie, czy wprowadzone modyfikacje przynoszą korzyści.

3.1 Porównanie algorytmów dla losowej tablicy o rozmiarze n = 1000

Algorytm	Porównania	Przypisania	Czas [ms]
INSERTION_SORT	249645	250644	1.757
INSERTION_SORT_MODIFIED	249903	251134	1.555
MERGE_SORT	11975	19952	0.610
MERGE_SORT_MODIFIED	23150	13084	0.512
HEAP_SORT	19148	18148	0.248
HEAP_SORT_MODIFIED	20184	12456	0.222

Analiza zaimplementowanych algorytmów pokazuje, że Insertion Sort generuje najwięcej porównań i przypisań, a modyfikacja polegająca na sortowaniu dwóch elementów naraz w niewielkim stopniu poprawia jego efektywność. Merge Sort jest znacznie szybszy, a trójdzielna wersja zmniejsza liczbę przypisań kosztem większej liczby porównań. Heap Sort wykazuje umiarkowaną liczbę operacji, a jego ternarna wersja dodatkowo skraca czas wykonania i redukuje liczbę przypisań. Ogólnie rzecz biorąc, algorytmy dzielące dane na podtablice lub korzystające z kopca są bardziej wydajne niż klasyczny Insertion Sort, zwłaszcza dla większych danych.

3.2 Porównanie algorytmów dla optymistycznego układu elementów w tablicy o rozmiarze n = 1000

Algorytm	Porównania	Przypisania	Czas [ms]
INSERTION_SORT	999	1998	0.007
INSERTION_SORT_MODIFIED	1501	2000	0.007
MERGE_SORT	11975	19952	0.525
MERGE_SORT_MODIFIED	23150	13084	0.456
HEAP_SORT	20416	19416	0.230
HEAP_SORT_MODIFIED	21810	13540	0.207

Dla najbardziej optymistycznej tablicy widać, że Insertion Sort w obu wersjach osiąga minimalną liczbę porównań i przypisań, co wynika z faktu, że dane są już posortowane. Modyfikacja Insertion Sort nie przynosi znaczącej poprawy w tym przypadku. Z kolei Merge Sort i Heap Sort wykazują większą liczbę porównań, ale ich czas wykonania pozostaje niski i stabilny, a wersje zmodyfikowane (Merge Sort Modified, Heap Sort Modified) zmniejszają liczbę przypisań kosztem większej liczby porównań. Wniosek: przy danych już posortowanych najprostsze algorytmy adaptują się najlepiej, natomiast modyfikacje wpływają głównie na balans między porównaniami a przypisaniami.

3.3 Porównanie algorytmów dla pesymistycznego układu elemntów w tablicy o rozmiarze n = 1000

Algorytm	Porównania	Przypisania	Czas [ms]
INSERTION_SORT	500499	501498	2.936
INSERTION_SORT_MODIFIED	500501	502500	2.738
MERGE_SORT	11975	19952	0.482
MERGE_SORT_MODIFIED	23150	13084	0.398
HEAP_SORT	17632	16632	0.184
HEAP_SORT_MODIFIED	19368	11912	0.17

Dla tablicy w przypadku pesymistycznym widać wyraźnie, że algorytmy oparte na prostym sortowaniu przez wstawianie (Insertion Sort i jego modyfikacja) wymagają bardzo dużej liczby porównań i przypisań, co przekłada się na najwyższy czas wykonania. Modyfikacja Insertion Sort nie przynosi znaczącej poprawy w tym scenariuszu – liczba operacji jest zbliżona, a czas wykonania nieco krótszy. Algorytmy dziel i zwyciężaj, takie jak Merge Sort i jego modyfikacja, wykazują znacznie mniejszą liczbę przypisań w wersji zmodyfikowanej, co przekłada się również na szybszy czas wykonania. Heap Sort i jego wersja ternarna (Heap Sort Modified) osiągają najlepsze czasy spośród wszystkich analizowanych algorytmów. Modyfikacja ternarna zmniejsza liczbę przypisań, kosztem niewielkiego wzrostu liczby porównań, co w praktyce skutkuje krótszym czasem działania. Podsumowując, dla pesymistycznej tablicy najefektywniejsze są algorytmy typu Heap Sort i Merge Sort, natomiast Insertion Sort pozostaje znacząco mniej wydajny.

3.4 Porówananie algorytmów dla dużych danych losowych - tablicy o rozmiarze n = 100000

Algorytm	Porównania	Przypisania	Czas [ms]
INSERTION_SORT	2 500 797 827	2 500 897 826	7633.18
INSERTION_SORT_MODIFIED	2 500 822 726	2 500 948 034	7739.84
MERGE_SORT	1 868 927	3 337 856	36.652
MERGE_SORT_MODIFIED	3 773 494	2 163 804	32.297
HEAP_SORT	3 248 748	3 148 748	20.112
HEAP_SORT_MODIFIED	3 286 506	2 091 004	17.902

Dla dużej tablicy Insertion Sort i jego modyfikacja okazują się bardzo nieefektywne – czas wykonania oraz liczba operacji rosną dramatycznie. Algorytmy o złożoności O(n log n), takie jak Merge Sort i Heap Sort, działają znacznie szybciej. Wersje modyfikowane, dzielące tablicę na trzy podtablice lub stosujące ternarny Heap Sort, zmniejszają liczbę przypisań i czas wykonania kosztem niewielkiego wzrostu liczby porównań, co czyni je bardziej wydajnymi dla dużych danych.

4 Wnioski

1. Złożoność algorytmów – Algorytm ma złożoność $O(n^2)$. (Insertion Sort i jego modyfikacja) działają dobrze dla małych tablic, ale dla dużych danych stają się bardzo nieefektywne. Algorytmy o złożoności O(n log n) (Merge Sort i Heap Sort) są zdecydowanie szybsze i skalowalne.

2. Wpływ modyfikacji – Wersje modyfikowane algorytmów Merge Sort i Heap Sort zmniejszają liczbę przypisań i czas wykonania, choć liczba porównań może wzrosnąć. Oznacza to, że modyfikacje są korzystne dla dużych danych. W przypadku Insertion Sort jego modyfikacja wpływa na wzrost liczby porównań, a czas wykonywania algorytmu przy większych danych zazwyczaj jest podobny.

3. Rodzaj danych – Wydajność algorytmów zależy od typu danych: Insertion Sort najlepiej działa dla danych już częściowo posortowanych (optymistyczny przypadek), a najgorzej dla danych odwrotnie posortowanych (pesymistyczny przypadek). Nie najlepiej idzie mu również z dużymi danymi. Algorytmy o złożoności O(n log n) są mniej wrażliwe na charakter danych, co prowadzi do tego, że w przypadku dużych, bądź pesymistycznych danych działają znacznie lepiej niż Insertion Sort.

4. Porównanie ogólne – Najbardziej uniwersalne i efektywne są algorytmy Merge Sort i Heap Sort, szczególnie w wersjach modyfikowanych dla dużych tablic. Insertion Sort jest użyteczny głównie dla małych lub częściowo posortowanych danych.