Sprawozdanie z listy 1 (AiSD)

Szymon Wojtasik, nr albumu: 287306

1 Idea trywialnego algorytmu

Teoretycznie o zwykłym insertion sort można niewiele powiedzieć, chociażby z uwagi na to, że sam algorytm jest bardzo krótki, jednakże ma pewną ideę by zrozumieć cięższe sortowania.
Kody będę prezentował bez przypisów i porównań i komentarzy, żeby było czytelniej.

```
1  for (int i=1; i<n; i++){
2      int x=A[i];
3      int j = i-1;
4      while (j>=0 && A[j] > x){
5          A[j+1]=A[j];
6          j--;
7          }
8      A[j+1]=x;
9      }
10 }
```

W całym tym algorytmie najważniejsze dla mnie było:
$\Longrightarrow$ "odkrycie"nadpisywania elementów(linijka 5)
$\Longrightarrow$ wypadanie z pętli wskutek czego element lądował na samym końcu(linijka 8)
Wskutek tego stopniowo sortujemy, jednak co się okaże potem jest to bardzo nieoptymalne względem późniejszych algorytmów

2 Modyfikacja Insertion Sort

2.1 Mała zagwostka

Przejdźmy do modyfikacji, która jest zdecydowanie ciekawsza.

```
1      if (A[i] < A[i-1]) {
2          int temp = A[i];
3          A[i] = A[i-1];
4          A[i-1] = temp;
```

Tutaj zorientowałem się, że gdybym nie nadpisywał elementów(linijka 2) to wtedy okazało by się, że kod jest 25 razy szybszy od normalnego, co skutkowało tym, że sortowania były oczywiście złe.
Po analizie okazało się, że bez nadpisania elementu algorytm wypadał szybko z następnej pętli, co kończyło się złym sortowaniem.
Mam na myśli, że powstawało A[-1] przez co elementy(zazwyczaj te najmniejsze) się dublowały.

2.2 Idea algorytmu

```
while (j>=0 && A[j] > wiekszy_element){
    A[j+2]=A[j];
        j--;
        A[j+2]= wiekszy_element;
    int k = j;
    while (k>=0 && A[k] > mniejszy_element){
        A[k+1]=A[k];
        k--;
    }
    A[k+1] = mniejszy_element;
```

Warto tutaj wspominieć o tym, że ideą algorytmu jest porównanie ze sobą trzech elementów i w zależności, który z nich jest największy, najmniejszy itd. to taką ustawiamy między nimi zależność.

2.3 Wyjątek

Występuje tutaj pewien wyjątek, o którym trzeba wspomnieć. Co jeżeli n jest nieparzyste?

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

Zauważmy, że jeżeli n jest nieparzyste to wyraz i = n - 1 jest parzysty, ale skoro zaczynamy liczyć 'i' od 1 to przechodzimy po nieparzystych.
Wniosek: Staniemy na nieparzystym i = n-2 i zostanie nam tylko jeden element do wybrania(bo tablica ma indeks do n-1).
Wypadlibyśmy tutaj z pętli przez co trzeba wprowadzić powyższy wyjątek

3 Wnioski do Insertion Sort

Algorytm nie jest najbardziej optymalny, ponieważ przechodzi przez n elementów w pętli for, a potem wchodzi w pętle while.
$\Longrightarrow$ Złożoność czasowa to $O(n^2)$ przez co jest to bardzo czasochłonny algorytm.

4 Merge sort(zwykły)

4.1 Pomijalny warunek

```
1     while(i < n_1){
2         A[t++] = L[i++];
3     }
```

Tutaj ciekawe było dla mnie odkrycie, że tak naprawdę $n_1 >= n_2$.
Moglibyśmy to rozwiązać równaniami, jednakże skupię się na czymś innym.
Tym sposobem omijamy jedną rzecz, którą byśmy musieli sprawdzić.
Na samym końcu może zostać tylko przecież jakiś element z lewej tablicy(nigdy z prawej).
$\implies$ Wniosek: Nie musimy już tworzyć kolejnej pętli while

5 Merge sort(zmodyfikowany)

5.1 Inicjalizacja zmiennych

Zauważmy, że w lewej i środkowej tablicy musi być albo tyle samo albo więcej elementów niż w prawej tablicy

```
1     int n_1, n_2, n_3;
2     n_1 = mid_1 - low + 1;
3     n_2 = mid_2 - mid_1;
4     n_3 = high - mid_2;
```

Można rozpatrzeć 3 przypadki, ponieważ dzielimy tablicę na trzy(n to ilość elementów).

$n = 3m \implies \text{mid1} = m - 1 \quad \text{mid2} = 2m - 1 \quad \text{high} = 3m - 1$ gdzie $n_1 = n_2 = n_3 = m$

$n = 3m + 1 \implies \text{mid1} = m \quad \text{mid2} = 2m \quad \text{high} = 3m$ gdzie $n_2 = n_3 = m \quad n_1 = m + 1$

$n = 3m+2 \implies \text{mid1} = m \quad \text{mid2} = 2m+1 \quad \text{high} = 3m+1$ gdzie $n_1 = n_2 = m+1 \quad n_3 = m$

Z każdego z trzech przypadków(przy założeniu, że low to 0) wychodzi, że:
$n_1 = \text{mid1} + 1$
$n_2 = \text{mid2} - \text{mid1}$
$n_3 = \text{high - mid2}$

5.2 Nadpisywanie elementów

Tak samo jak w zwykłym merge sort, trzeba porównywać pewne elementy, ale są aż trzy tablice. Ciekawym, choć czasochłonnym jest to, że musimy wyprodukowac 3 warunki, w których musi w dodatku być koniunkcja, ponieważ chcemy mieć pewność, że dany element o tym samym indeksie jest największy.

```
    while(i < n_1 && j < n_2 && k < n_3){
        if (L[i] <= M[j] && L[i] <= R[k]){
            A[t++] = L[i++];
        }
        else {
            if (M[j] <= R[k] && M[j] <= L[i]){
            A[t++] = M[j++];
        }
        else{
            A[t++] = R[k++];
        }
    }
```

Dalej, jeżeli coś wypadnie z tablicy powtarzamy procedurę ze zwykłego Merge sort(ale trzykrotnie). Poniżej wkleję jeden z trzech takich kodów.

```
    while(i < n_1 && j < n_2) {
        if(L[i] <= M[j]) {
            A[t++] = L[i++];
        }
        else {
            A[t++] = M[j++];
        }
    }
```

Ten konkretny pojawi się wtedy, gdy skończą nam się już elementy z prawej tablicy. Zauważmy, że jest tu zwykłe powielenie typowego merge sorta. Zwykłe porównanie co jest większe.

5.3 Inicjalizacja zmiennych i rekurencja

```
1  void MERGE_SORT_2(int A[], int low, int high, int &porownania,
       int &przypisania){
2      if (low < high){
3          int mid_1;
4          int mid_2;
5          mid_1 = (2 * low + high)/3;
6          mid_2 = (low + 2 * high)/3;
7          MERGE_SORT_2(A, low, mid_1, porownania, przypisania);
8          MERGE_SORT_2(A, mid_1 + 1, mid_2, porownania, przypisania
               );
9          MERGE_SORT_2(A, mid_2 + 1, high, porownania, przypisania)
               ;
10         MERGE_2(A, low, mid_1, mid_2, high, porownania,
               przypisania);
11     }
12 }
```

Warto w tym miejscu również wspomnieć o rekurencji, która gra największą rolę w całym algorytmie.
$\Longrightarrow$ Algorytm dzieli tablicę na 3 podtablice
$\Longrightarrow$ Na początku dzieli na podtablicę lewą do momentu, jak jest spełniony warunek początkowy(low $<$ high)
$\Longrightarrow$ Kiedy skończy cofa się do poprzedniej tablicy i zaczyna dzielić na podtablicę środkową
$\Longrightarrow$ Po skończonej pracy dzieli tą samą tablicę na podtablice prawą. $\Longrightarrow$ Jeżeli są już utworzone wszystkie podtablicy to tablica wyjściowa zostaje scalona i tak robimy do momentu aż warunek początkowy jest spełniony

Teraz krótkie wytłumaczenie skąd pojawiło się równanie na mid1 oraz mid2.
mid1 = low + (mid2 - low)/2
mid2 = mid1 + (high - mid1)/2
mid1 = low + (mid1 + (high - mid1)/2 - low)/2 = low + ((mid1)/2 + high/2 -low)/2
mid1 = low/2 + (mid1)/4 + high/4 $\iff$ $\frac{3}{4}\cdot$mid1 = $\frac{1}{2}\cdot$low + $\frac{1}{4}\cdot$high
$\iff$ mid1 = (2low + high)/3
Po wstawieniu do mid2, tego co wyszło w mid1, otrzymujemy rzeczywiście to co mamy w kodzie.

6 Wnioski do Merge sort

Algorytm tworzy drzewo binarne(dla zwykłego Merge), które w każdym rzędzie ma o 2 razy więcej elementów. Ponadto w każdym rzędzie ma do podzielenia wiecej tablic ale o mniejszej ilości elementów przez co tak naprawdę dzieli m tablic o ilości elementów $\frac{n}{m}$
Wnioskiem jest to że w każdym rzędzie dzieli $m \cdot \frac{n}{m} = n$ elementów.
Ilość gałęzi jest wynikiem logarytmu o podstawie 2.
$\Longrightarrow$ Wniosek: Złożoność czasowa to $O(n \cdot \log n)$

7 Heap sort

7.1 Kopcowanie

Najciekawszym elementem w całym heap sorcie jest kopcowanie. To tutaj można poznać całą ideę tego algorytmu.

```
    if (l < heap_size && A[l] > A[largest])
        largest = l;
    if (r < heap_size && A[r] > A[largest])
        largest = r;
```

W tym momencie porównujemy, które z dzieci jest większe. Jest to o tyle potrzebne, że wtedy będziemy wiedzieli, które dziecko wyląduje w miejscu rodzica.

```
    if (largest != i) {
        swap(A[i], A[largest]);
        HEAPIFY(A, largest, heap_size, por, prz);
    }
```

Sprawdzamy czy któryś z dzieci jest większt od rodzica i jeżeli tak to zamieniamy je miejscami.
W dalszej części znowu kopcujemy(stosujemy rekurnecje), ponieważ kopiec mógł zostać zdeformowany.

7.2 Warunek konieczny

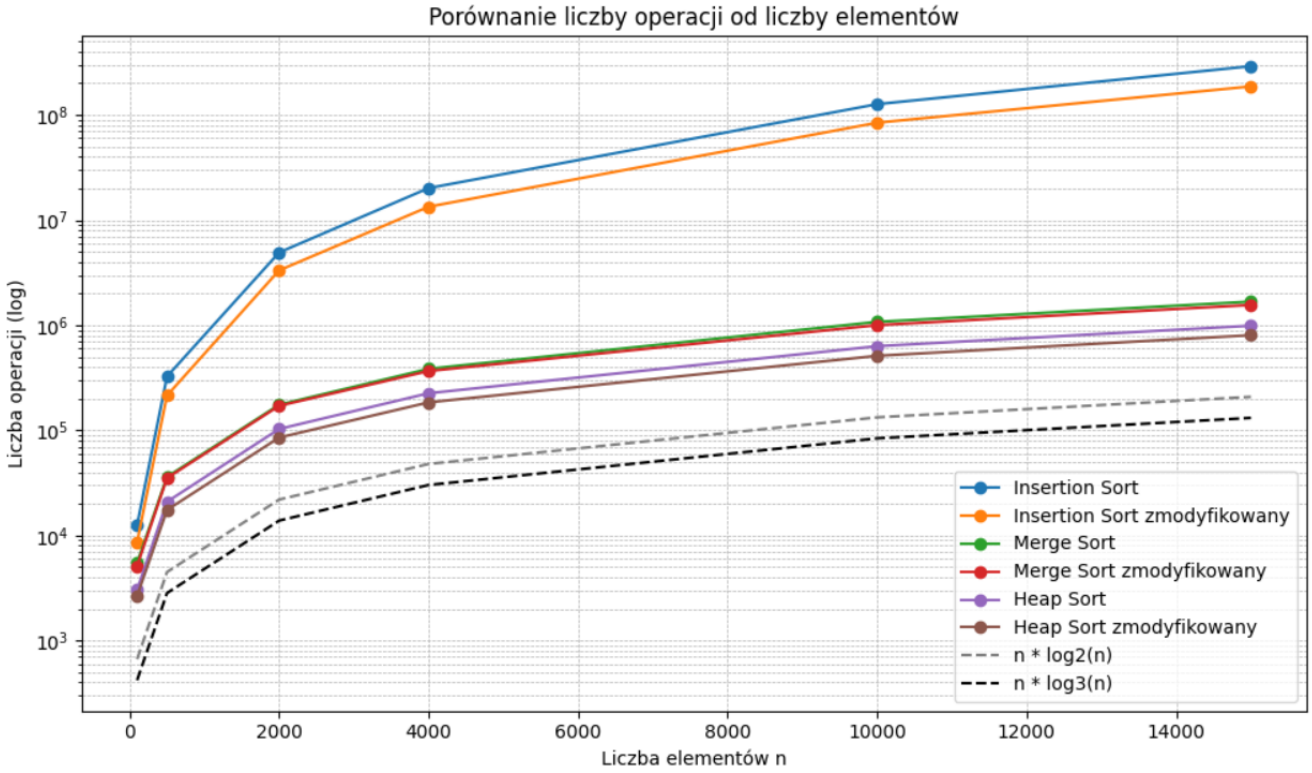
Warto również wspomnieć o warunku koniecznym istnienia kopca binarnego.
$\Longrightarrow$ Nie może być przerwy w tablicy. Z tego też powodu możemy jedynie wstawić za element z indeksem 0(nigdy gdzieś w środku). O tym mówi poniższy kod

```
    swap(A[0], A[i]);
    heap_size--;
```

7.3 Kopiec ternarny(albo d-arny)

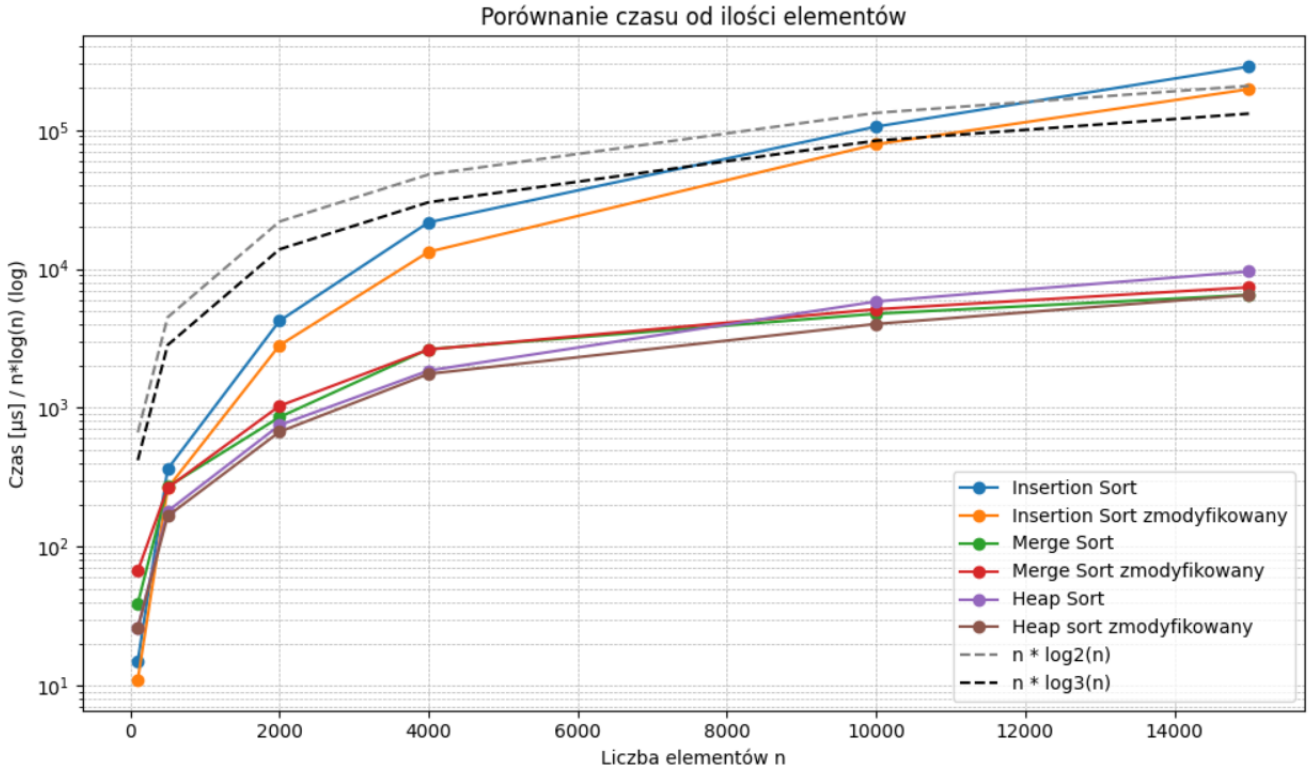
Celowo pominąłem już modyfikację, ponieważ jest ona synonimiczna do binarnej, a jedynie trzeba zmienić ilość gałęzi z dwóch na trzy. Tak samo byśmy postępowali gdybyśmy musieli swtorzyć kopiec d - arny.
Teoretycznie tworzymy więcej dzieci w kopcu ternarnym, ale kończy się szybciej niż binarny, przez co paradoksalnie na wykresach widać, że ternarny jest szybszy.

8 Wykres wykonywanych operacji od elementów



Z wykresu możemy odczytać, jak znacząca różnica czasowa jest pomiędzy słabo zoptymalizowanym insertion sort, a już dużo lepiej heap sort i merge sort.
Ponadto widać, że Merge i Heap na początku rośnie szybko, a potem coraz wolniej co wskazuje na złożoność czasową: $O(n \log n)$
Tak jak wcześniej wspomniałem Insertion sort jest czasochłonny i to nie bez przyczyny, ponieważ z samego początku sprawozdania wynikło, że jego złożoność czasowa to: $O(n^2)$
Każda modyfikacja nieco wnosi, aczkolwiek najmniej modyfikacja merge, co zobaczymy na kolejnym wykresie

9 Wykres czasu od elementów



Wykres jest bardzo podobny, jednakże biorąc pod lupę inny wykres jakim jest czas widać, że najwięcej dała optymalizacja do Insertion i Heap. Czemu w takim razie nie dało dużo do Merge, a nawet wskazuje, że jest gorszy.
Odpowiedź nie jest oczywista. Z poprzednich rozważań wiemy, że Merge Sort ma złożoność $O(n \log n)$ w tym dla binarnego na pewno $O(n \log_2 n)$. Na logikę można by powiedzieć, że będzie $O(n \log_3 n)$ więc musi być szybszy, ale kiedy już scalamy mamy warunek z koniunkcją co daje nam dodatkową rzecz do sprawdzania.
Z tego mamy wniosek, że mimo tego iż drzewa binarne jest wyższe od drzewa ternarnego w Merge to przez bardziej czasochłonne scalanie dostajemy efekt odwrotny do zamierzonego. Sytuacja wygląda odwrotnie dla kopca binarnego i ternarnego.
Tam przez to, że wprowadzamy minimalne poprawki do kodu, nasza złożoność czasowa nie cierpi, a nawet jej to pomaga.

10 Tabela z wynikami oraz wnioski ostateczne

	n	Sortowanie	Czas [µs]	Operacje
0	100	Insertion Sort	15	12562
1	100	Insertion Sort zmodyfikowany	11	8612
2	100	Merge Sort	39	5473
3	100	Merge Sort zmodyfikowany	67	5127
4	100	Heap Sort	26	3035
5	100	Heap Sort zmodyfikowany	26	2655
6	500	Insertion Sort	362	325000
7	500	Insertion Sort zmodyfikowany	264	214073
8	500	Merge Sort	270	36061
9	500	Merge Sort zmodyfikowany	265	35127
10	500	Heap Sort	180	20890
11	500	Heap Sort zmodyfikowany	166	17454
12	2000	Insertion Sort	4224	4915401
13	2000	Insertion Sort zmodyfikowany	2815	3322841
14	2000	Merge Sort	856	175096
15	2000	Merge Sort zmodyfikowany	1031	171061
16	2000	Heap Sort	749	102945
17	2000	Heap Sort zmodyfikowany	671	85308
18	4000	Insertion Sort	21648	20145904
19	4000	Insertion Sort zmodyfikowany	13271	13388674
20	4000	Merge Sort	2644	384483
21	4000	Merge Sort zmodyfikowany	2629	366495
22	4000	Heap Sort	1853	225330
23	4000	Heap Sort zmodyfikowany	1754	184419
24	10000	Insertion Sort	105795	126652394
25	10000	Insertion Sort zmodyfikowany	78901	84362784
26	10000	Merge Sort	4755	1075786
27	10000	Merge Sort zmodyfikowany	5121	1000876
28	10000	Heap Sort	5825	631610
29	10000	Heap Sort zmodyfikowany	4015	511287
30	15000	Insertion Sort	286321	290871903
31	15000	Insertion Sort zmodyfikowany	196872	186754302
32	15000	Merge Sort	6504	1676090
33	15000	Merge Sort zmodyfikowany	7372	1557649
34	15000	Heap Sort	9582	988820
35	15000	Heap Sort zmodyfikowany	6488	802962

Z tabeli możemy wyciągnąć już ostateczne wnioski. Najważniejszym moim zdaniem jest już taki, o którym wspomniałem na samym początku, czyli Insertion bardzo odstaje od pozostałych przez swoją słabą złożoność czasową: $O(n^2)$
Ponadto widać, że jak dla n=10000 tak dla n = 15000, Merge(zwykły) oraz Heap(zmodyfikowany) mają bardzo podobny czas wywoływania.
Jest to co prawda jakiś jeden z wielu przypadków, ale śmiało można wyprowadzić wniosek, że zdecydowanie te dwa algortymy są najbardziej optymalne