Lista 1

Anna Leszczyńska

Październik 2025r

1 Wstęp

W sprawozdraniu porównane są algorytmy insertion sort, merge sort, heap sort oraz ich modyfikacje.

2 Najciekawsze fragmenty kodu

2.1 Insertion sort: liczenie tylko zapisów do tablicy

W pętli przesuwającej elementy liczymy wyłącznie zapisy do tablicy (a nie przypisania do zmiennych lokalnych). To pozwala porównywać realny koszt pamięciowy między algorytmami.

Pętla przesuwa większe elementy w prawo (`ASSIGN++` za każdy zapis), a test porządkowy podbija `COMP++`.

```
while (j >= 0) {
    COMP++;
    if (!(a[j] > key)) break;
    a[j+1] = a[j]; ASSIGN++;
    --j;
}
a[j+1] = key; ASSIGN++;
```

2.2 DoubleInsertion: wstawianie pary bez nadpisywania

Każdą iterację traktujemy jako wstawienie dwóch elementów naraz. Najpierw porządkujemy parę w 2 zmiennych ($x \leq y$), potem wstawiamy najpierw x, a następnie y do poszerzonego prefiksu — bez ryzyka nadpisania.

```
int x = a[i-2], y = a[i-1];
COMP++; if (x > y) { int t=x; x=y; y=t; }

int j = i - 3;
while (j >= 0) { COMP++; if (!(a[j] > x)) break;
  a[j+1] = a[j]; ASSIGN++; --j; }
a[j+1] = x; ASSIGN++;

int k = i - 2;
while (k >= 0) { COMP++; if (!(a[k] > y)) break;
  a[k+1] = a[k]; ASSIGN++; --k; }
a[k+1] = y; ASSIGN++;
```

2.3 MergeSort(2): proste scalanie z jawnym kosztem buforów

Podkreślamy, że kopiowanie do buforów to też zapisy do tablic. Dzięki temu „koszt pamięciowy” mergesorta jest porównywalny z heapsortem.

Dwa bufory, liniowe scalanie, każdy zapis do `a[k]` zwiększa `ASSIGN`.

```
for (int i = 0; i < n1; ++i) { Lbuf[i] = a[L+i]; ASSIGN++; }
for (int j = 0; j < n2; ++j) { Rbuf[j] = a[M+1+j]; ASSIGN++; }

while (i < n1 && j < n2) {
    COMP++; // Lbuf[i] <= Rbuf[j] ?
    if (Lbuf[i] <= Rbuf[j]) { a[k++] = Lbuf[i++]; ASSIGN++; }
    else                    { a[k++] = Rbuf[j++]; ASSIGN++; }
}
```

2.4 MergeSort(3): wybór minimum z trzech w dwóch porównaniach

Wybieramy minimum z $\{A[i], B[j], C[k]\}$ tylko dwoma porównaniami: najpierw A z B, potem wynik z C. Minimalizujemy `COMP`, kosztem większej liczby zapisów (trzy bufory).

```
COMP++; bool Ai_le_Bj = (A[i] <= B[j]);
int vMinAB = Ai_le_Bj ? A[i] : B[j];
COMP++;
if (vMinAB <= C[k]) {
    if (Ai_le_Bj) { a[t++] = A[i++]; ASSIGN++; }
    else          { a[t++] = B[j++]; ASSIGN++; }
} else {
    a[t++] = C[k++]; ASSIGN++;
}
```

2.5 HeapSort (binarny)

Pokazujemy realny profil porównań: dwa potencjalne testy dzieci na poziom. Dzięki temu można porównać z kopcem trójdzielnym. Spychamy element w dół, każde porównanie zwiększa `COMP`, a zamiana to `countedSwap` (liczy tylko zapisy do tablicy).

```
int l = 2*i+1, r = 2*i+2;
if (l < n) { COMP++; if (a[l] > a[largest]) largest = l; }
if (r < n) { COMP++; if (a[r] > a[largest]) largest = r; }
if (largest != i) { countedSwap(a, i, largest); i = largest; } else break;
```

2.6 HeapSort (ternarny)

Ten kod oblicza indeksy trzech potencjalnych dzieci elementu o indeksie i $(3i+1, 3i+2, 3i+3)$. Następnie, za pomocą porównań (COMP++), ustala, który z czterech elementów (rodzic lub jedno z trzech dzieci) jest największy. Jeśli to dziecko jest największe, element jest zamieniany z rodzicem (countedSwap inkrementuje ASSIGN++) i proces przesiewania w dół jest kontynuowany (i = largest). Jeśli rodzic jest największy, działanie funkcji jest przerywane

```
int c1 = 3*i+1, c2 = 3*i+2, c3 = 3*i+3;
if (c1 < n) { COMP++; if (a[c1] > a[largest]) largest = c1; }
if (c2 < n) { COMP++; if (a[c2] > a[largest]) largest = c2; }
if (c3 < n) { COMP++; if (a[c3] > a[largest]) largest = c3; }
if (largest != i) { countedSwap(a, i, largest); i = largest; } else break;
```

3 Metodologia eksperymentów

Program generuje pięć scenariuszy danych: `Random`, `NearlySorted`, `Sorted`, `Reversed`, `FewUniques`. Dla rozmiarów $n \in \{50, 200, 800, 2000\}$ i $T = 5$ uruchamień dla każdego algorytmu, zapisuje wyniki do wyniki.csv w formacie: scenario,n,algorithm,comps,assigns, time(w ms) .

4 Wyniki: tabele

Poniższe tabele są tworzone bezpośrednio z danych zawartych w pliku wyniki.csv.

Scen.	n	Algorytm	Porównania	Przypisania	Czas [ms]
Random	50	Insertion	581	583	0.008200
Random	50	DoubleInsertion	591	569	0.008200
Random	50	MergeSort2	226	572	0.122100
Random	50	MergeSort3	274	384	0.100400
Random	50	HeapBinary	417	490	0.012400
Random	50	HeapTernary	412	366	0.010800

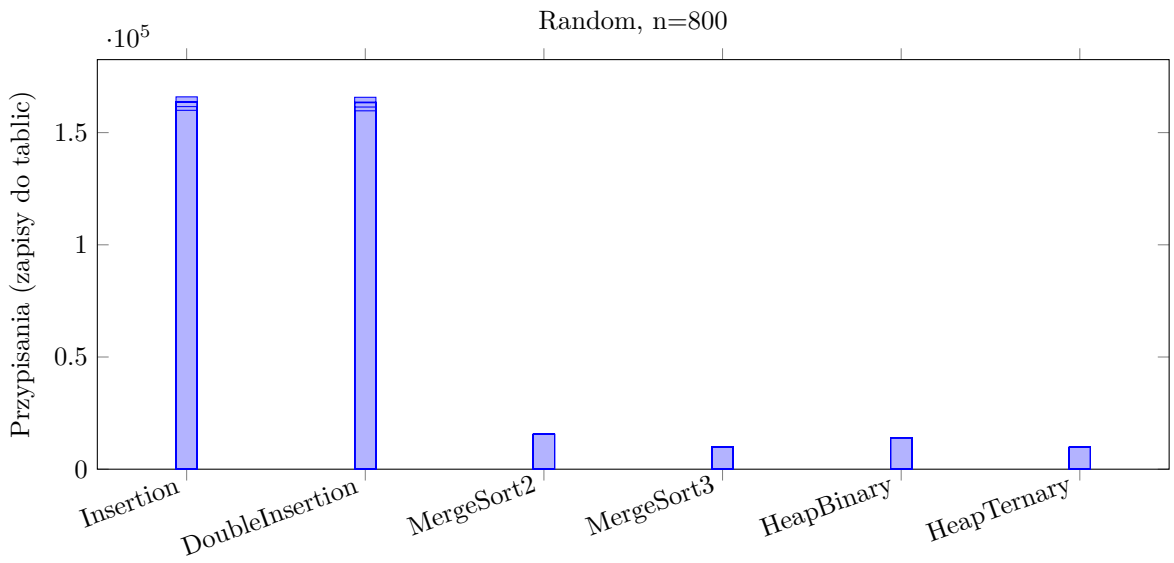
Tabela 1: Pierwsze 6 wyników dla $n = 50$.

Scen.	n	Algorytm	Porównania	Przypisania	Czas [ms]
Random	800	Insertion	161587	161592	1.976600
Random	800	DoubleInsertion	161798	161404	2.496900
Random	800	MergeSort2	6720	15552	2.248600
Random	800	MergeSort3	8295	9856	1.374700
Random	800	HeapBinary	13026	14072	0.357300
Random	800	HeapTernary	12728	9744	0.275400

Tabela 2: Pierwsze 6 wyników dla $n = 800$.

Scen.	n	Algorytm	Porównania	Przypisania	Czas [ms]
Random	2000	Insertion	983448	983457	12.112800
Random	2000	DoubleInsertion	983943	982951	12.459800
Random	2000	MergeSort2	19376	43904	5.152400
Random	2000	MergeSort3	23918	28964	4.312500
Random	2000	HeapBinary	37729	40394	0.970300
Random	2000	HeapTernary	36778	27738	0.803100

Tabela 3: Pierwsze 6 wyników dla $n = 2000$.



Rysunek 1: Zapis do tablic (Random, n=800).

5 Wnioski

- **Insertion** wymaga największej ilości przypisań i zajmuje najdłużej.
- **Heap (binarny) vs Heap (ternarny):** Różnice w liczbie porównań i zapisów są niewielkie, zależne od danych.
- **Rodziny Merge vs Heap:** MergeSort wymaga dodatkowych buforów, więc liczba zapisów do tablic jest wyższa niż w HeapSort.