

# Algorytmy i struktury danych

## Sortowanie i kopce

### Przygotowanie do kolokwium

Przyjmując, że `tabA[] = {1, 2, 3, 4, 5, 6, 7}` oraz `tabB[] = {7, 6, 5, 4, 3, 2, 1}` i stosując algorytmy sortujące ściśle według procedur z pliku `sort2023.cc` wykonaj polecenia:

#### Zadanie 1

Ile dokładnie porównań (między elementami tablic) wykona `insertion_sort(tabB)`, a ile `insertion_sort(tabA)`?

##### Przypadek optymistyczny

Dla tablicy posortowanej rosnąco o długości  $n$ , `insertion_sort` wykona  **$n-1$  porównań**.

`insertion_sort` dla tablicy `tabA` wykona **6 porównań**.

Złożoność czasowa:  $O(n)$

##### Przypadek pesymistyczny

Dla tablicy posortowanej malejąco o długości  $n$ , `insertion_sort` wykona  $\frac{n^2-n}{2}$  **porównań**.

`insertion_sort` dla tablicy `tabA` wykona **21 porównań**.

Złożoność czasowa:  $O(n^2)$

#### Zadanie 2

Ile co najwyżej porównań (między elementami tablic) wykona procedura scalająca `merge` dwie tablice  $n$ -elementowe?

Procedura scalająca dwie tablice  $n$ -elementowe `merge` wykona co najwyżej  **$2n - 1$  porównań**

w przypadku, gdy elementy tablic są posortowane naprzemiennie rosnąco.

Złożoność czasowa procedury `merge` dla każdego przypadku:  $O(n)$

```
void merge(int n, int k, double leftTable[], double rightTable[]) {
    int i = 0;
    int j = k;
    int l = 0;
    while (i < k && j < n)
        if (leftTable[i] <= leftTable[j])
            rightTable[l++] = leftTable[i++];
        else
            rightTable[l++] = leftTable[j++];

    while (i < k)
        leftTable[--j] = leftTable[--k];

    for (i = 0; i < j; i++)
        leftTable[i] = rightTable[i];
}
```

### Zadanie 3

Jaka jest pesymistyczna złożoność czasowa procedury `merge_sort`? Odpowiedź uzasadnij.

Złożoność czasowa procedury `merge_sort` wynosi  $O(n \log n)$ , ponieważ każde wywołanie procedury `merge_sort` dzieli tablicę na dwie części o połowie długości, a następnie wywołuje procedurę `merge` na tych dwóch częściach. Złożoność czasowa nie zależy od ilości elementów i tego jak są posortowane.

Algorytm `merge_sort` składa się z dwóch etapów: `divide` i `merge`.

Złożoność czasowa etapu `divide` wynosi  $O(n)$ , ponieważ wykonujemy  $n - 1$  podziałów.

Złożoność czasowa etapu `merge` wynosi  $O(n \log n)$ , ponieważ wykonujemy  $\log n$  skaleń mając zawsze do dyspozycji  $n$  elementów.

Razem złożoność czasowa algorytmu `merge_sort` wynosi:  $O(n) + O(n \log n) = O(n \log n)$

Dowód metodą rekurencji uniwersalnej:

$$T(n) = 2T(n/2) + O(n)$$

$$f(n) = n^{\log_2 2} = n$$

Rozważamy drugi przypadek:

$$T(n) = O(n \log n)$$

### Zadanie 4

Ile co najwyżej porównań (między elementami tablicy) wykona procedura `partition`?

Procedura `partition` wykona co najwyżej  $n + 1$  porównań.

```
int partition(double t[], int n)
{
    int k = -1;
    double x = t[n / 2];
    for (;;) {
        do
            ++k;
        while (t[k] < x);

        do
            --n;
        while (t[n] > x);

        if (k < n)
            std::swap(t[k], t[n]);
        else
            return k;
    }
}
```

## Zadanie 5

Jak jest średnia a jaka pesymistyczna złożoność `quick_sort`. Odpowiedź uzasadnij.

Średnia złożoność czasowa algorytmu `quick_sort` wynosi  $O(n \log n)$ .  
pesymistyczna złożoność czasowa algorytmu `quick_sort` wynosi  $O(n^2)$ .

```
void quick_sort(double t[], int n)
{
    if (n > 1) {
        int k = partition(t, n); // podziel na dwie czesci
        quick_sort(t, k);        // posortuj lewa
        quick_sort(t + k, n - k); // posortuj prawa
    }
}
```

## Zadanie 6

Jaka jest złożoność funkcji `buildheap`? Przeprowadź dowód - uzasadnij swoją odpowiedź.

```
void sift_down(double t[], int n, int i)
{
    /*
     Przesiej element t[i] w dol kopca:
     t[2*i+1] - lewe dziecko t[i]
     t[2*i+2] - prawe dziecko t[i]
     jesli ktoreś z dzieci jest wieksze od t[i]:
     - zamien t[i] z tym dzieckiem miejscami
     - sprawdź ponownie t[i] w nowym miejscu
     */

    int k = i;
    double x = t[i];
    // x mniejszy od wiekszego syna
    while (((k += k + 2) < n && t[k - 1] < t[k]) || (k < n) && x < t[k])
    {
        t[i] = t[k];
        i = k;
    }
    t[i] = x;
}

void build_max_heap(double t[])
{
    for (int i = n / 2; i >= 0; i--)
        sift_down(t, i);
}
```

Złożoność czasowa funkcji `buildheap` wynosi  $O(n)$ . **Dowód:**

Na każdym poziomie oprócz ostatniego, węzły mogą mieć maksymalnie dwójkę dzieci: lewe i prawe. Dlatego przesiewając, porównujemy maksymalnie dwa razy. Ostatni poziom drzewa to  $n/2$  węzłów. Węzły te nie mają potomków, zatem nie możemy przesiewać. Poziom wyżej to  $n/4$  węzłów. Maksymalnie można wykonać 1 przesianie, ponieważ węzeł na tym poziomie może mieć tylko dzieci. Poziom wyżej to  $n/8$  węzłów. Węzły te mają maksymalnie dzieci i potomka. Maksymalnie można wykonać 2 przesiania, etc. Ilość porównań `sift_down` to ciąg:

$$2\left(\frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \dots\right)$$

Po przekształceniu wzory widoczne jest to, że składa się z on nieskończonych ciągów geometrycznych. Jeśli chcemy policzyć sumę tych ciągów, to możemy użyć wzoru na sumę nieskończonego ciągu geometrycznego. Można to zrobić pod warunkiem, Jeśli ciąg jest zbieżny. Iloczyn  $q$  wynosi  $1/2$ , zatem

spełniony jest warunek o zbieżności ciągu. Wzór na sumę wyrazów nieskończonego ciągu geometrycznego to:

$$S = \frac{a}{1-q}$$

Ilość porównań wykonywanych przez funkcję `sift_down` opisana jest za pomocą poniższego wzoru.

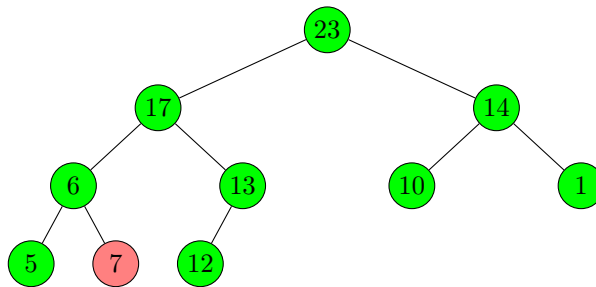
$$\begin{aligned}
 & 2 \sum_{i=1}^{h-1} \frac{n}{2^{i+1}} \cdot i \\
 & 2 \left( \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \frac{n}{32} \cdot 4 + \dots \right) \\
 & \frac{n}{2} \left( \frac{1}{1} + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \dots \right) \\
 & \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = \frac{\frac{1}{1}}{1 - \frac{1}{2}} = 2 \\
 & \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 1 \\
 & \frac{1}{4} + \frac{1}{8} + \dots = 0.5 \\
 & \frac{1}{8} + \dots = 0.25 \\
 & \dots
 \end{aligned}$$

$$\frac{n}{2} \cdot \frac{2}{1 - \frac{1}{2}} = \frac{n}{2} \cdot 4 = 2n = O(n)$$

### Zadanie 11

Czy ciąg {23, 17, 14, 6, 13, 10, 1, 5, 7, 12} jest kopcem?

Ciąg nie jest kopcem, ponieważ węzeł o wartości 7 nie spełnia warunku kopca.  
Niespełnionym warunkiem kopca jest to, że dziecko nie może być większe od swojego rodzica.



## Zadanie 12

Zilustruj działanie procedury buildheap dla ciągu 5,3,17,10,84,19,6,22,9. Narysuj na kartce wygląd tablicy/kopca po każdym wywołaniu procedury przesiej.

{5, 3, 17, 10, 84, 19, 6, 22, 9}  
{5, 3, 17, 22, 84, 19, 6, 10, 9}  
{5, 3, 19, 22, 84, 17, 6, 10, 9}  
{5, 84, 19, 22, 3, 17, 6, 10, 9}  
{84, 5, 19, 22, 3, 17, 6, 10, 9}  
{84, 22, 19, 5, 3, 17, 6, 10, 9}  
{84, 22, 19, 10, 3, 17, 6, 5, 9}