# Algorytmy i Struktury Danych

## Wykład 2

### Problem sortowanie

Dane: ciąg $a_0, \dots, a_{n-1}$ danych z operatorem $\leqslant$

Wynik: Permutacja $a'_0, \dots, a'_{n-1}$, taka że

$$a'_0 \leqslant a'_1 \leqslant \dots \leqslant a'_{n-1}$$

### Uwagi

reprezentacja danych
- tablica
- lista
  - 1-kier.
  - 2-kier.
- plik

algorytmy sortowania
- proste $O(n^2)$
- szybkie $O(n \log n)$

## Sortowanie przez scalanie (Merge Sort)

1 8 2 5 | 4 3 7 9

sort. rek. lewą stronę — sort. rek. prawą stronę

1 2 5 8 | 3 4 7 9

1 2 3 4 | 1 2 5 8 | 3 4 7 9

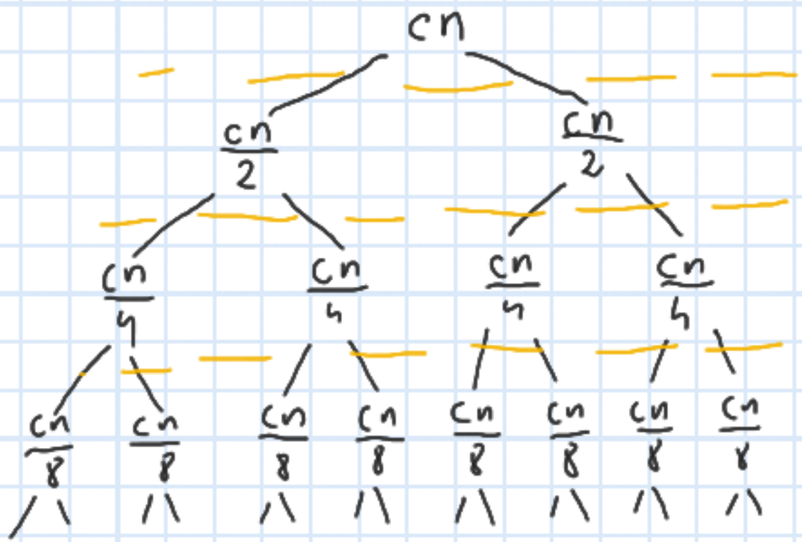
① Posortuj lewą połowę danych

② Posortuj prawą połowę danych

③ Scal posortowane fragmenty

Złożoność czasowa

$$T(n) = \begin{cases} c, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn, & n > 1 \end{cases}$$



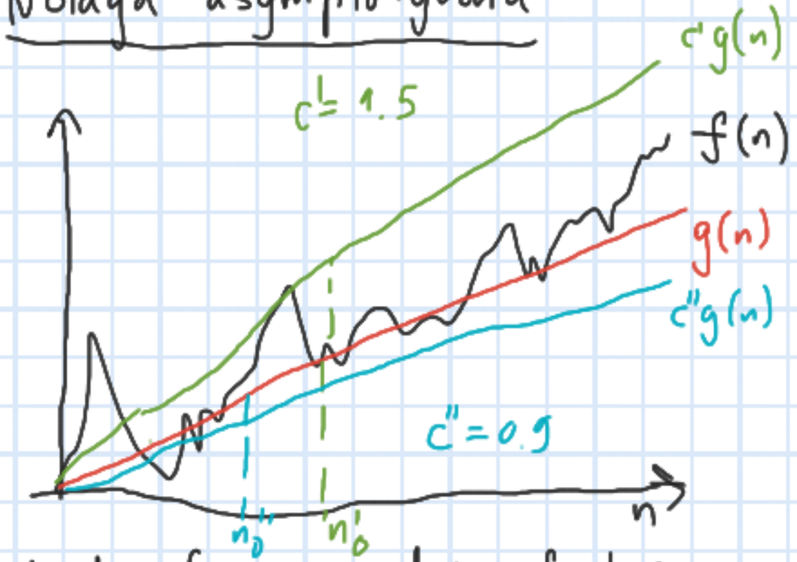

cn
cn
cn
cn
⋮

poziomów rekurencji jest $\approx \log n$

Złożoność czasowa to $T(n) \approx cn \log n$

$$\frac{n}{2^{\log n}} = \frac{n}{n} = 1$$

# Notacja asymptotyczna




Niech $f$, $g$ – dwie funkcje

$f : \mathbb{N} \to \mathbb{N}$

$g : \mathbb{N} \to \mathbb{N}$

$15n^3 + 7n^2$ jest $\Theta(n^3)$

$n^4$ jest $O(n^5)$

$20n^2 + 7$ jest $\Omega(n)$

**def**

Mówimy, że $f$ jest $O(g(n))$ jeśli:

$(\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\left[ f(n) \leq c \cdot g(n) \right]$

Mówimy, że $f$ jest $\Omega(g(n))$ jeśli:

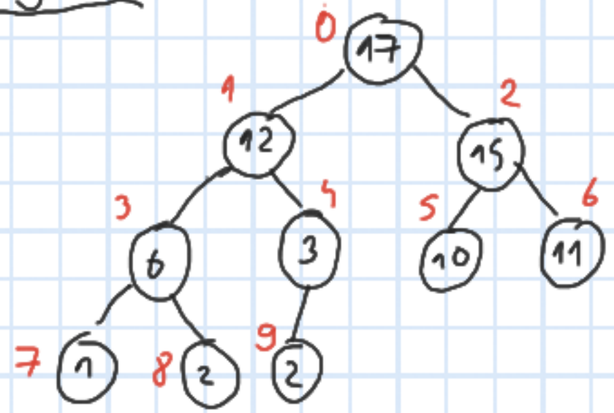
$(\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\left[ f(n) \geq c \cdot g(n) \right]$

Mówimy, że $f$ jest $\Theta(g(n))$ jeśli jest $O(g(n))$ oraz $\Omega(g(n))$.

# Sortowanie kopcowe (Heapsort)

Kopiec - drzewo binarne, w którym
każdy węzeł wewnętrzny przechowuje
wartość większą lub równą niż
jego dzieci

Przykład

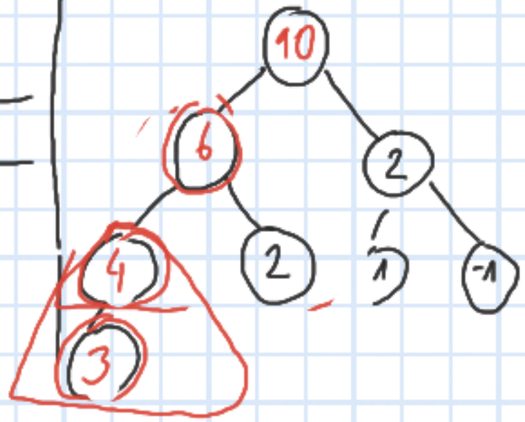



$left(i) = 2i + 1$

$right(i) = 2i + 2$

$parent(i) = \lfloor i - 1/2 \rfloor$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 17 | 12 | 15 | 6 | 3 | 10 | 11 | 1 | 2 | 2 |




```
def left(i): return 2*i + 1
def right(i): return 2*i + 2
def parent(i): return (i-1) // 2
```

## Przywracanie własności kopca

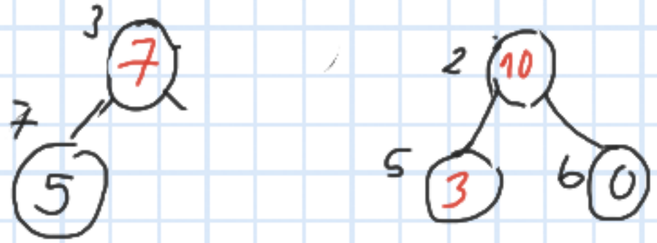

```
def heapify(A, i, n)

    l = left(i)
    r = right(i)
    max_ind = i
    if l < n and A[l] > A[max_ind]: max_ind = l
    if r < n and A[r] > A[max_ind]: max_ind = r

    if max_ind ≠ i:
        swap(A[i], A[max_ind])
        heapify(A, max_ind, n)
```

$O(\log n)$

## Budowanie kopca

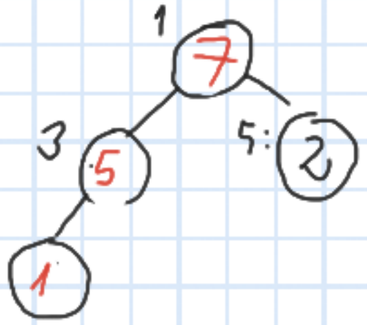
```
def build_heap(A):
    n = len(A)
    for i in range(parent(n-1), -1, -1):
        heapify(A, i, n)
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 8 | 7 | 10 | 5 | 2 | 3 | 0 | 1 |










```
def heap_sort(A):
    n = len(A)
    build_heap(A)
    for i in range(n-1, 0, -1):
        swap(A[0], A[i])
        heapify(A, 0, i)
```

$O(n \log n)$