

# Algorytmy i Struktury Danych

Wojciech Typer

---

**Algorithm 1** Insertion Sort

---

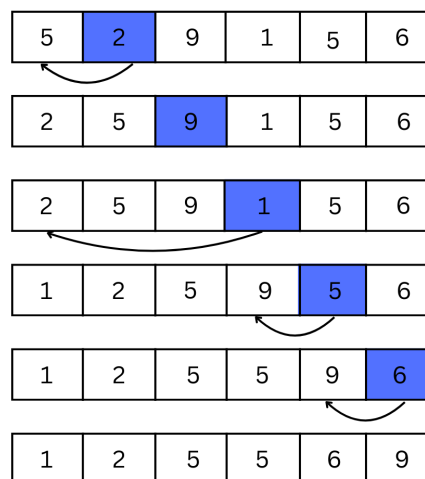
```
1: procedure INSERTIONSORT( $A, n$ )
2:   for  $i = 1$  to  $n - 1$  do
3:      $key = A[i]$ 
4:      $j = i - 1$ 
5:     while  $j \geq 0$  and  $A[j] > key$  do
6:        $A[j + 1] = A[j]$ 
7:        $j = j - 1$ 
8:     end while
9:      $A[j + 1] = key$ 
10:  end for
11: end procedure
```

---

**Złożoność czasowa:**  $O(n^2)$

**Best case:** w najlepszym przypadku złożoność czasowa będzie wynosić  $O(n)$

**Złożoność pamięciowa:**  $O(1)$



---

**Algorithm 2** Merge Sort

---

```
1: procedure MERGESORT( $A, 1, n$ )
2:   if  $|A[1..n]| == 1$  then
3:     return  $A[1..n]$ 
4:   else
5:      $B = \text{MergeSort}(A, 1, \lfloor n/2 \rfloor)$ 
6:      $C = \text{MergeSort}(A, \lfloor n/2 \rfloor, n)$ 
7:     return Merge( $B, C$ )
8:   end if
9: end procedure
```

---

---

**Algorithm 3** Merge

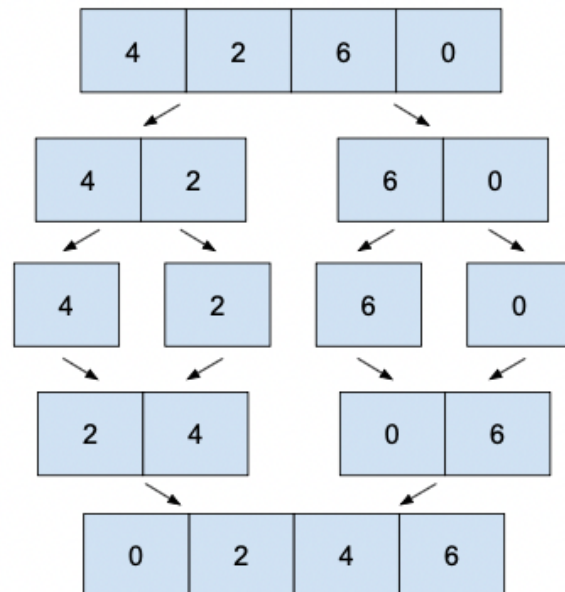
---

```
1: procedure MERGE( $X[1..k], Y[1..n]$ )
2:   if  $X = \emptyset$  then
3:     return  $Y$ 
4:   else if  $Y = \emptyset$  then
5:     return  $X$ 
6:   else if  $X[1] \leq Y[1]$  then
7:     return  $[X[1]] \times \text{Merge}(X[2..k], Y[1..n])$ 
8:   else
9:     return  $[Y[1]] \times \text{Merge}(X[1..k], Y[2..n])$ 
10:  end if
11: end procedure
```

---

**Złożoność czasowa Merge Sort:**  $O(n \log n)$

**Złożoność pamięciowa Merge Sort:**  $O(n)$



Istnieje również iteracyjna wersja algorytmu Merge, sort, która została przedstawiona poniżej w postaci pseudokodu.

---

**Algorithm 4** IterativeMergeSort

---

```

1: procedure ITERATIVEMERGESORT( $A[1..n]$ )
2:   for  $size = 1$  to  $n - 1$  by  $size \times 2$  do
3:     for  $left = 0$  to  $n - 1$  by  $2 \times size$  do
4:        $mid \leftarrow \min(left + size - 1, n - 1)$ 
5:        $right \leftarrow \min(left + 2 \times size - 1, n - 1)$ 
6:       MERGE( $A$ ,  $left$ ,  $mid$ ,  $right$ )
7:     end for
8:   end for
9: end procedure

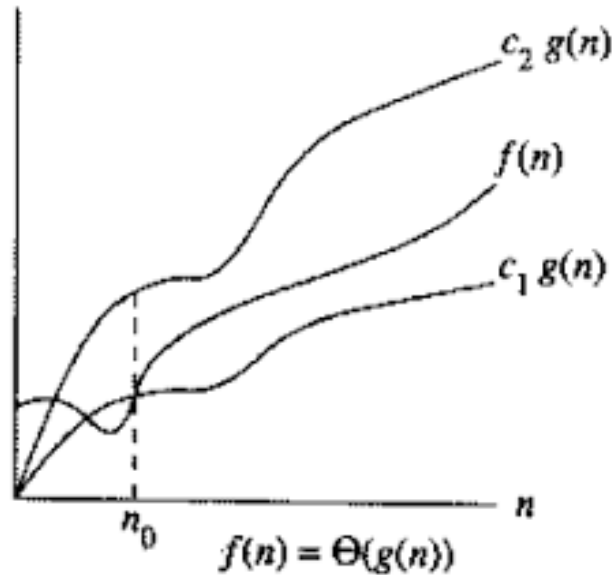
```

---

**Złożoność czasowa Iterative Merge Sort:**  $O(n \log n)$  - dzieje się tak, ponieważ  $size$  jest podwajany o 2 w każdej iteracji, więc potrzebujemy około  $\log_2 n$  iteracji, a w każdej z nich wykonujemy  $O(n)$  operacji.

**Złożoność pamięciowa Iterative Merge Sort:**  $O(n)$

**Notacja asymptotyczna**  $O: f(n) = O(g(n)) \rightarrow (\exists c > 0)(\exists n_0 \in N) : \forall n \geq n_0 \rightarrow 0 \leq f(n) \leq c \cdot g(n)$



$$f(n) = O(g(n)) \rightarrow \lim_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} < \infty$$

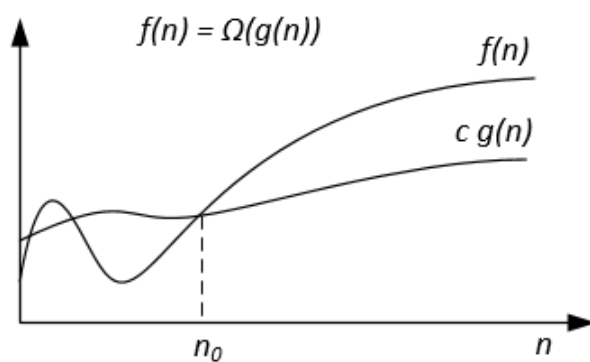
**Notacja asymptotyczna - własności**

a)  $f(n) = n^3 + O(n^2) \rightarrow (\exists h(n) = O(n^2))(f(n) = n^3 + h(n))$

b)  $n^2 + O(n) = O(n^2) \rightarrow (\forall f(n) = O(n))(\exists h(n) = O(n^2))(n^2 + f(n) - h(n))$

**Notacja  $\Omega$**

$$f(n) = \Omega(g(n)) \rightarrow (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c * g(n) \leq |f(n)|)$$



**Notacja  $\Omega$  - własności**

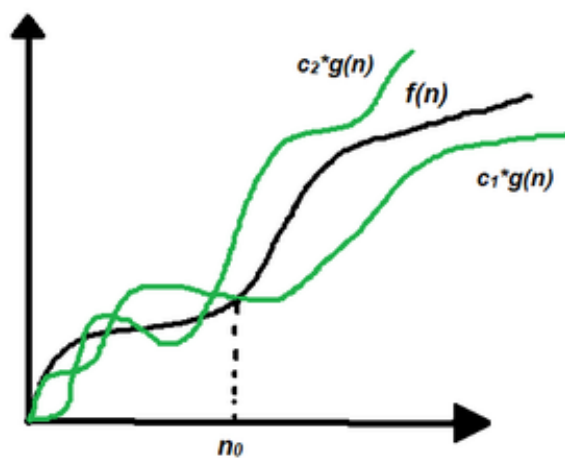
a)  $n^3 = \Omega(2n^2)$

b)  $n = \Omega(\log(n))$

c)  $2n^2 = \Omega(n^2)$

**Notacja  $\Theta$**

$$f(n) = \Theta(g(n)) \rightarrow (\exists c_1, c_2 > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c_1 g(n) \leq |f(n)| \leq c_2 g(n))$$



$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

### Notacja o- małe

$$f(n) = o(g(n)) \rightarrow (\forall c > 0)(\exists n_0 \in N)(\forall n \geq n_0)(|f(n)| < c * |g(n)|)$$

### Notacja o- małe - przykłady

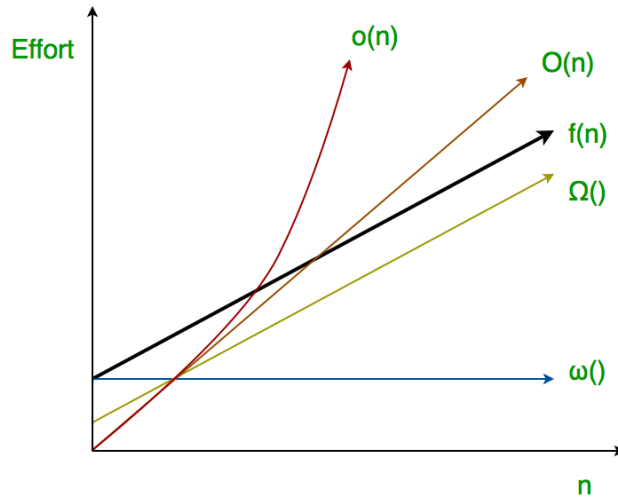
a)  $117n \log(n) = o(n^2)$

b)  $n^2 = o(n^3)$

### Notacja $\omega$

$$f(n) = \omega(g(n)) \rightarrow (\forall c > 0)(\exists n_0 \in N)(\forall n \geq n_0)(|f(n)| > c * |g(n)|)$$

$$\lim_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} = \infty$$



### Rekurencje

Metoda podstawiania (metoda dowodzenia indukcyjnego)

1. Zgadnij odpowiedź (bez stałych)
2. Sprawdź przez indukcję, czy dobrze zgadliśmy
3. Znajdź stałe

Przykład 1:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Pierwszy strzał:  $T(n) = O(n^3)$

Cel: pokazać, że  $(\exists c > 0) T(n) \leq c * n^3$

Krok początkowy:  $T(1) = \Theta(1) = c * 1^3 = c$

Krok indukcyjny: zał. że,  $(\forall k < n)(T(k) \leq c * k^3) =$

Dowód:  $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4c * \left(\frac{n}{2}\right)^3 + n = \frac{1}{2}cn^3 + n =$   
 $= cn^3 - \frac{1}{2}cn^3 + n = cn^3 - \left(\frac{1}{2}cn^3 - n\right) \leq cn^3$

Pokazaliśmy, że  $T(n) = O(n^2)$

Spróbujmy wzmocnić zał. indukcyjne:  $T(n) \leq c_1 n^2 - c_2 n$

$$T(n) \leq 4T\left(\frac{n}{2}\right) + n \leq 4\left(c_1\left(\frac{n}{2}\right)^2 - c_2\frac{n}{2}\right) + n = \\ = c_1 n^2 - 2c_2 n + n = c_1 n^2 - (2c_2 - 1)n \leq c_1 n^2 - c_2 n$$

Musimy dobrać takie  $c_1, c_2$ , aby  $2c_1 \geq c_2$

Wówczas otrzymamy  $T(1) = O(1) \leq c_1 1^2 - c_2 1$

Przykład 2:

$$T(n) = 2T(\sqrt{n}) + \log(n)$$

Założmy, że  $n$  jest potęgą dwójki  $n = 2^m \rightarrow m = \log(n)$

$$T(2^m) = 2T(2^{m/2}) + m$$

oznaczymy  $T(2^m) = S(m)$

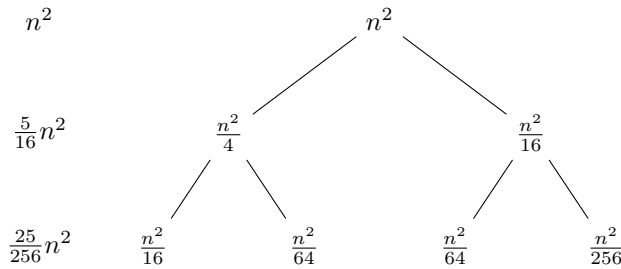
$$T(2^m) = 2T(2^{m/2}) + m \rightarrow 2S(m/2) + m$$

$$S(m) = O(m \log(m))$$

$$T(n) = O(\log(n) \log(\log(n))) \text{ (formalnie powinniśmy to udowodnić)}$$

## Drzewo rekursji

$$\text{Przykład : } T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n^2$$



Trzeba pamiętać, że drzewo rekursji samo w sobie nie jest formalnym rozwiązaniem problemu. Nie można go używać do dowodzenia złożoności algorytmów. Jest to jedynie intuicyjne podejście do problemu. Formalnie  $T(n)$  należałoby policzyć jako sumę wszystkich wierzchołków w drzewie rekursji:

$$T(n) = \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k \cdot n^2 = n^2 \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k = n^2 \frac{1}{1 - \frac{5}{16}} = n^2 \frac{16}{11} = \frac{16}{11} n^2$$

Widzimy zatem, że  $T(n) = O(n^2)$

## Master Theorem

Niech  $a \geq 1, b > 1, f(n), d \in \mathbb{N}$  oraz  $f(n)$  będzie funkcją nieujemną. Rozważmy rekurencję:

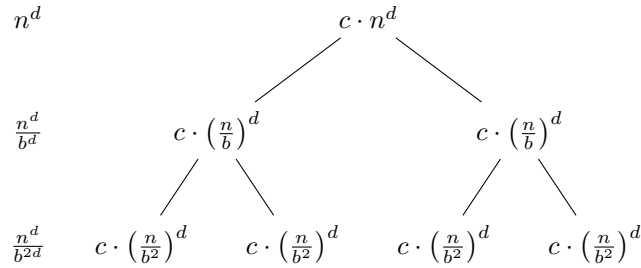
$$T(n) = aT\left(\frac{a}{b}\right) + \Theta(n^d)$$

Wówczas:

- $\Theta(n^d)$ , jeśli  $d > \log_b a$
- $\Theta(n^d \log(n))$ , jeśli  $d = \log_b a$
- $\Theta(n^{\log_b a})$ , jeśli  $d < \log_b a$

Do przedstawienia problemu użyjemy drzewa rekursji. Rozważmy rekurencję:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$$



1. suma kosztów w  $k$ -tym kroku

$$a^k c \left(\frac{n}{b^k}\right)^d = c \left(\frac{a}{b^d}\right)^k n^d$$

gdzie  $c \left(\frac{n}{b^k}\right)^d$  to koszt jednego podproblemu w  $k$ -tym kroku

2. obliczenie wysokości drzewa:

$$\frac{n}{b^h} = 1 \rightarrow h = \log_b n$$

3. Obliczenie  $T(n)$

$$\begin{aligned} T(n) &= \Theta \left( \sum_{k=0}^{\log_b n} c \frac{a}{b^k} n^d \right) \\ &= \Theta \left( c \cdot n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \\ &= \Theta \left( c \cdot n^d \frac{1 - \left( \frac{a}{b^d} \right)^{\log_b n + 1}}{1 - \frac{a}{b^d}} \right) \\ &\Rightarrow T(n) = \Theta(n^d) \end{aligned}$$

4. rozważmy 3 przypadki:

(a)  $d > \log_b a$

$$T(n) = \Theta(n^d)$$

(b)  $d = \log_b a$

$$T(n) = \Theta(n^d \log n)$$

(c)  $d < \log_b a$

$$T(n) = \Theta(n^{\log_b a})$$

## Przykłady

- $T(n) = 4T(\frac{n}{2}) + 11n$

Wtedy korzystając z **Master Theorem** mamy:

$$a = 4, b = 2, d = 1$$

Jak i również

$$\log_b a = \log_2 4 = 2 > 1 = d \implies T(n) = \Theta(n^2)$$

- $T(n) = 4T(\frac{n}{3}) + 3n^2$

Wtedy

$$a = 4, b = 3, d = 2$$

Jak i również

$$\log_b a = \log_3 4 < 2 = d \implies T(n) = \Theta(n^2)$$

- $T(n) = 27T(\frac{n}{3}) + \frac{n^2}{3}$

Wtedy

$$a = 27, b = 3, d = 2$$

Jak i również

$$\log_b a = \log_3 27 = 3 > 2 = d \implies T(n) = \Theta(n^3 \log n)$$

## Metoda dziel i zwyciężaj (D&C)

Na czym ona polega?

1. Podział problemu na mniejsze podproblemy
2. Rozwiązanie rekurencyjnie mniejsze podproblemy
3. połącz rozwiązania podproblemów w celu rozwiązania problemu wejściowego

## Algorytm – Binary Search

- **Input:** posortowana tablica  $A[1..n]$  oraz element  $x$
- **Output:** indeks  $i$  taki, że  $A[i] = x$  lub 0 jeśli  $x$  nie występuje w  $A$
- przebieg algorytmu:

---

### Algorithm 5 Binary Search

---

```
1: procedure BINARYSEARCH( $A, x$ )
2:    $l = 1$ 
3:    $r = |A|$ 
4:   while  $l \leq r$  do
5:      $m = \lfloor \frac{l+r}{2} \rfloor$ 
6:     if  $A[m] = x$  then
7:       return  $m$ 
8:     else if  $A[m] < x$  then
9:        $l = m + 1$ 
10:    else
11:       $r = m - 1$ 
12:    end if
13:  end while
14:  return 0
15: end procedure
```

---



- **Asymptotyka** Algorytm spełnia następującą rekurencję:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Rozwiązując za pomocą **Master Theorem** otrzymujemy:

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

## Divide & Conquer

**Problem:** Obliczenie  $x^n$ .

Rozwiązanie naiwną metodą iteracyjną:

$$x^n = x \cdot x \cdot \dots \cdot x \Rightarrow \Theta(n)$$

Rozwiązanie za pomocą Divide & Conquer:

$$x^n = \begin{cases} (x^{\frac{n}{2}}) \cdot (x^{\frac{n}{2}}), & \text{gdy } n \text{ jest parzyste} \\ (x^{\frac{n-1}{2}}) \cdot (x^{\frac{n-1}{2}}) \cdot x, & \text{gdy } n \text{ jest nieparzyste} \end{cases}$$

Rekurencyjna złożoność czasowa:

$$T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$$

---

**Problem:** Obliczenie  $n$ -tej liczby Fibonacciego

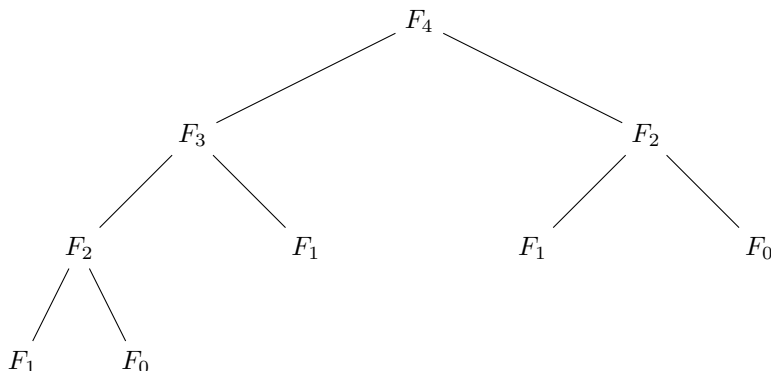
Metoda rekurencyjna:

$$F(n) = F(n-1) + F(n-2)$$

Ma ona złożoność wykładniczą:

$$\Theta(\phi^n), \quad \text{gdzie } \phi = \frac{1 + \sqrt{5}}{2}$$

Drzewo rekurencyjne dla  $F_4$ :



**Wzór jawny:**

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - (-\phi)^{-n})$$

Obliczanie  $F_n$  macierzą: Zamiast rekurencji można użyć potęgowania macierzy, co daje optymalną złożoność. Dla każdego  $n \geq 0$  zachodzi:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Potęgowanie macierzy metodą szybkiego potęgowania daje czas:

$$\Theta(\log n)$$

co jest znaczną poprawą w porównaniu do wykładniczej rekurencji.

**Mnożenie liczb binarnych metodą Divide & Conquer**

**Wejście:**  $x, y$  **Wyjście:**  $x \cdot y$

Każdą liczbę można rozbić na dwie połowy:

$$x = x_L \cdot 2^{\frac{n}{2}} + x_R$$

$$y = y_L \cdot 2^{\frac{n}{2}} + y_R$$

Podstawiając do iloczynu:

$$xy = (x_L \cdot 2^{\frac{n}{2}} + x_R) \cdot (y_L \cdot 2^{\frac{n}{2}} + y_R)$$

Po rozwinięciu:

$$xy = x_L y_L \cdot 2^n + (x_L y_R + x_R y_L) \cdot 2^{\frac{n}{2}} + x_R y_R$$

Rekurencyjna zależność czasowa:

$$T(n) = 4T(n/2) + \Theta(n)$$

Zastosowanie **Master Theorem** daje:

$$T(n) = \Theta(n^2)$$

co pokazuje, że metoda ta nie poprawia złożoności względem standardowego mnożenia.

**Optymalizacja: metoda Gaussa**

Zamiast wykonywać 4 mnożenia rekursywne, można zastosować zasadę Gaussa:

$$xy = x_L y_L \cdot 2^n + ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) \cdot 2^{\frac{n}{2}} + x_R y_R$$

Dzięki temu zamiast 4 mnożeń wykonujemy tylko 3:

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

Zastosowanie **Master Theorem** daje:

$$T(n) = \Theta(n^{\log_2 3})$$

---

**Algorithm 6** Multiply - Mnożenie dużych liczb binarnych metodą Gaussa

---

```
1: procedure MULTIPLY( $x, y$ )
2:    $n \leftarrow \max(|x|, |y|)$ 
3:   if  $n = 1$  then
4:     return  $x \cdot y$ 
5:   end if
6:    $m \leftarrow \lceil n/2 \rceil$ 
7:    $x_L, x_R \leftarrow$ 
8:    $y_L, y_R \leftarrow$ 
9:    $p_1 \leftarrow \text{MULTIPLY}(x_L, y_L)$ 
10:   $p_2 \leftarrow \text{MULTIPLY}(x_R, y_R)$ 
11:   $p_3 \leftarrow \text{MULTIPLY}((x_L + x_R), (y_L + y_R))$ 
12:  return  $p_1 \cdot 2^{2m} + (p_3 - p_1 - p_2) \cdot 2^m + p_2$ 
13: end procedure
```

---

**QuickSort**

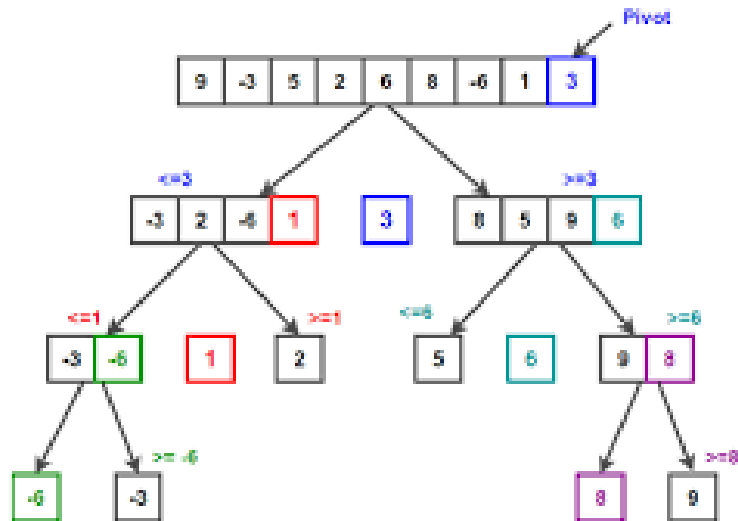
---

**Algorithm 7** QuickSort - Sortowanie szybkie

---

```
1: procedure QUICKSORT( $A, low, high$ )
2:   if  $low < high$  then
3:      $p \leftarrow \text{PARTITION}(A, low, high)$ 
4:     QUICKSORT( $A, low, p - 1$ )
5:     QUICKSORT( $A, p + 1, high$ )
6:   end if
7: end procedure
8:
9: procedure PARTITION( $A, low, high$ )
10:   $pivot \leftarrow A[high]$ 
11:   $i \leftarrow low - 1$ 
12:  for  $j \leftarrow low$  to  $high - 1$  do
13:    if  $A[j] \leq pivot$  then
14:       $i \leftarrow i + 1$ 
15:      SWAP( $A[i], A[j]$ )
16:    end if
17:  end for
18:  SWAP( $A[i + 1], A[high]$ )
19:  return  $i + 1$ 
20: end procedure
```

---



---

**Algorithm 8** Hoare Partition

---

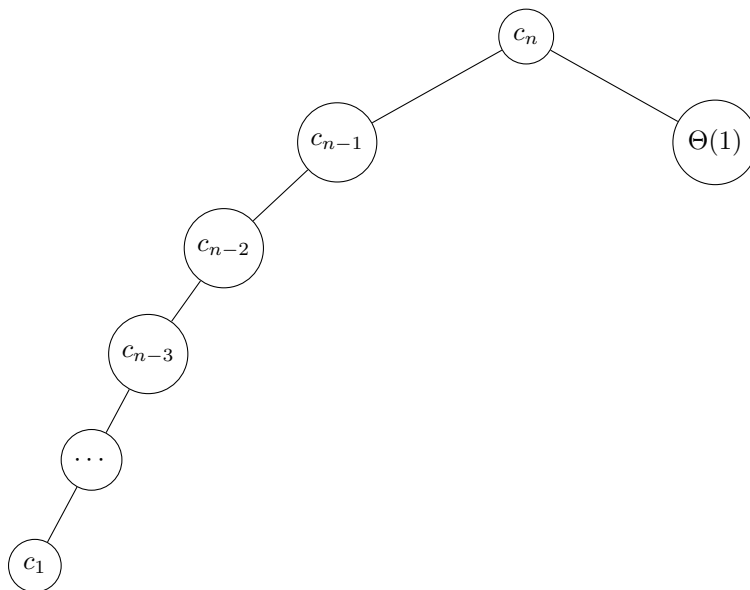
```
1: procedure HOARE_PARTITION( $A, p, q$ )
2:    $pivot \leftarrow A \left[ \left\lfloor \frac{p+q}{2} \right\rfloor \right]$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow q + 1$ 
5:   while true do
6:     repeat
7:        $i \leftarrow i + 1$ 
8:     until  $A[i] \geq pivot$ 
9:     repeat
10:       $j \leftarrow j - 1$ 
11:    until  $A[j] \leq pivot$ 
12:    if  $i \geq j$  then
13:      return  $j$ 
14:    end if
15:    swap( $A[i], A[j]$ )
16:  end while
17: end procedure
```

---

**Analiza worst-case QuickSorta**

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

Drzewo rekurencji (dla przypadku pesymistycznego, tj. jednostronny podział):



$$T(n) \leq \sum_{i=1}^n c \cdot i = c \cdot \sum_{i=1}^n i = \Theta(n^2)$$

**Analiza best-case**

Jeśli pivot zawsze dzieli tablicę na dwie równe części:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$$

**Analiza average-case**

Niech  $T_n$  oznacza liczbę porównań dla tablicy długości  $n$ .

$$x_k = \begin{cases} 1, & \text{jeśli partition dzieli tablicę na } (k, n-k-1) \\ 0, & \text{w przeciwnym wypadku} \end{cases}$$

$$T_n = \sum_{k=0}^{n-1} x_k \cdot (T_k + T_{n-k-1}) + (n-1)$$

Liczmy wartość oczekiwaną:

$$E(T_n) = \sum_{k=0}^{n-1} \mathbb{E}(x_k) \cdot (\mathbb{E}(T_k) + \mathbb{E}(T_{n-k-1})) + (n-1)$$

$$\mathbb{E}(x_k) = \frac{1}{n} \quad (\text{bo pivot jest losowy})$$

$$\begin{aligned} E(T_n) &= \frac{1}{n} \sum_{k=0}^{n-1} (E(T_k) + E(T_{n-k-1})) + (n-1) \\ &= \frac{2}{n} \sum_{k=0}^{n-1} E(T_k) + (n-1) \\ &\Rightarrow E(T_n) = \Theta(n \log n) \end{aligned}$$

**Analiza avg Case'a**  $T_n \rightarrow$  Liczba porównań elementów sortowanej tablicy:  $|A| = n$

$$x_k = \begin{cases} 1, & \text{jeśli partition dzieli tablicę na } (k, n-k-1) \\ 0, & \text{w przeciwnym wypadku} \end{cases}$$

$$T_n = \begin{cases} T_0 + T_{n-1} + n - 1, & \text{gdy}(0, n-1) - \text{split} \\ T_1 + T_{n-2} + n - 1, & \text{gdy}(1, n-2) - \text{split} \\ \dots \\ T_k + T_{n-1-k} + n - 1, & \text{gdy}(k, n-k-1) - \text{split} \\ \dots \\ T_{n-1} + T_0 + n - 1, & \text{gdy}(n-1, 0) - \text{split} \end{cases}$$

$$T_n = \sum_{k=0}^{n-1} x_k (T_k + T_{n-k-1}) + n - 1$$

liczymy wartość oczekiwaną:

$$\begin{aligned} E(T_n) &= E(\sum_{k=0}^{n-1} x_k (T_k + T_{n-k-1}) + n - 1) \\ E(T_n) &= \sum_{k=0}^{n-1} E(x_k \cdot (T_k + T_{n-k-1}) + n - 1) \\ E(T_n) &= \sum_{k=0}^{n-1} E(x_k) \cdot E(T_k + T_{n-k-1} + n - 1) \\ E(T_n) &= \frac{1}{n} \cdot \sum_{k=0}^{n-1} E(T_k) + \sum_{k=0}^{n-1} E(T_{n-k-1}) \end{aligned}$$

**Dual pivot quicksort**

< LP	LP	LP ≤ & ≤ RP	RP	RP <
------	----	-------------	----	------

**Wartość oczekiwana:**

$$E(\text{liczba porównań w dual pivot partition}) \approx \frac{16}{9}n$$

$E(\text{liczba porównań w dual pivot qs sedwick}) \approx \frac{32}{15}n \log n$

#### Yaroslavsky dual pivot qs

$E(\text{liczba porównań w partition}) \approx \frac{19}{12}n$

$E(\text{liczba porównań w Dual Pivot qs Yaroslavsky}) \approx 1.9n \log n$

#### Strategia count

$E(\text{liczba porównań w Count Partition}) \approx \frac{3}{2}n$

$E(\text{liczba porównań w Dual Pivot qs z count}) \approx 1.8n \log n$

#### Comparsion Model

Dolne ograniczenie na liczbę porównań w problemie sortowania w Comparsion Model wynosi  $\Omega(n \log n)$

D-d:

- dla dowolnego algorytmu sortującego możemy znaleźć odpowiadające mu drzewo decyzyjne
- $n!$  liści w binarnym drzewie decyzyjnym
- drzewo binarne pełne o wysokości  $h$  ma co najmniej  $2^h$  liści
- ale liści w drzewie decyzyjnym powinno być co najmniej  $n!$ , zatem:

$$2^h \leq n! / \lg$$

$$h \leq \log_2 n!$$

$$\lg n! = \lg(\sqrt{s\pi n}(\frac{n}{e})^n(1 + o(1)))$$

$$\lg(\frac{n}{e})^n + \lg(\sqrt{2\pi n}(1 + o(1)))$$

$$n \log n - n \lg e + \lg(\sqrt{2\pi n}(2 + o(1))) = \Omega(n \log n)$$

Sortowanie:

Input:  $|a| = n, \forall i \in \{1, \dots, k\}$

Output: posortowana rosnąco tablica  $A$

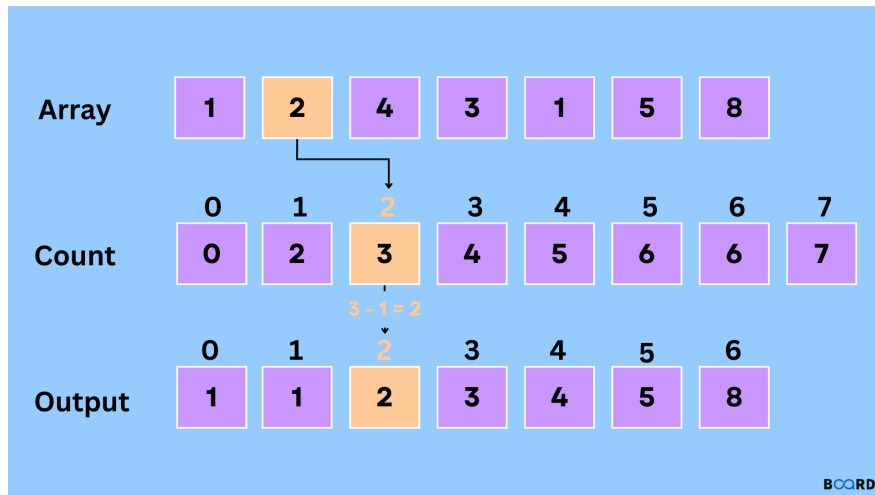
---

#### Algorithm 9 CountingSort

---

```
1: procedure COUNTING_SORT( $A, n, k$ )
2:   for  $i = 1$  to  $k$  do
3:      $C[i] \leftarrow 0$ 
4:   end for
5:   for  $i = 1$  to  $n$  do
6:      $C[A[i]] \leftarrow C[A[i]] + 1$ 
7:   end for
8:   for  $i = 2$  to  $k$  do
9:      $C[i] \leftarrow C[i] + C[i - 1]$ 
10:  end for
11:  for  $i = n$  downto  $1$  do
12:     $B[C[A[i]]] \leftarrow A[i]$ 
13:     $C[A[i]] \leftarrow C[A[i]] - 1$ 
14:  end for
15:  return  $B$ 
16: end procedure
```

---



Złożoność obliczeniowa Counting Sorta:  
 $\Theta(n + k)$  gdzie  $k = O(n)$

### Stable Sorting Property

Algorytm zachowuje kolejność równych sobie elementów z tablicy wejściowej

## RadixSort

### Algorithm 10 RadixSort

```

1: procedure RADIX_SORT( $A, n, d$ )
2:   for  $i = 1$  to  $d$  do
3:     counting_sort( $A, n, 9$ )
4:   end for
5:   return  $A$ 
6: end procedure

```

### Złożoność obliczeniowa RadixSorta

- $n$  liczb  $b$ -bitowych
- liczb  $b$  bitowych dzielimy na  $r$ -bitowe cyfry
- cyfry są z  $|0, \dots, 2^r - 1| = 2^r$
- Counting Sort sortujący  $n$  liczb względem jednej cyfry

Zatem RadixSort będzie miał złożoność obliczeniową:

$$\Theta\left(\frac{b}{r} \cdot (n + 2^r)\right)$$

Co po wykonaniu skomplikowanej analizy daje:

$$\Theta(d \cdot n)$$

### Statystyki pozycyjne

**Def:**  $k$ -tą statystyką pozycyjną nazywam  $\leftarrow k$ -tą najmniejszą wartość z zadanego zbioru  
 przykład:

- $k = 1 \rightarrow O(n)$
- $k = n \rightarrow O(n)$
- $k = \lfloor \frac{n}{2} \rfloor \rightarrow$  sortowanie  $O(n \log n)$

---

**Algorithm 11** RandomSelect

---

```

1: procedure RANDOM_SELECT( $A, p, q, i$ )
2:   if  $p = q$  then
3:     return  $A[p]$ 
4:   end if
5:    $r \leftarrow \text{RandPartition}(A, p, q)$ 
6:    $k \leftarrow r - p + 1$ 
7:   if  $i = k$  then
8:     return  $A[r]$ 
9:   else if  $i < k$  then
10:    return RANDOM_SELECT( $A, p, r - 1, i$ )
11:  else
12:    return RANDOM_SELECT( $A, r + 1, q, i - k$ )
13:  end if
14: end procedure

```

---

**Select algorithm**

- dzielimy  $A[p..q]$  na  $\lceil \frac{n}{5} \rceil$  pięcioelementowych części oraz ostatnią część na  $\leq 5$  elementów
- Sortujemy te grupy i wybieramy z każdej z nich medianę
- Znajdujemy medianę  $M$ . Select( $M, 1, \frac{n}{5}, \frac{n}{10}$ )
- Ustalamy  $X$  jako pivot; Partition( $A, p, q$ ) i tak samo jak w RandomSelect

**Select**

Select( $A, K$ )  $\rightarrow T(n)$

- Dziel na 5 elementowe tablice i znajdź ich medianę  $\rightarrow \Theta(n)$
- Select (...)  $\rightarrow$  znajdź medianę median  $\rightarrow T(\lceil \frac{n}{5} \rceil)$
- Użyj mediany median jako pivot w Partition  $\rightarrow \Theta(n)$
- Idź do lewej albo prawej podtablicy w zależności od indeksu pivota i szukaj statystyki pozycyjnej

Otrzymujemy:  $t(n) = T(\lceil \frac{n}{5} \rceil) + \Theta(n)$

**Struktury danych**



Set interface:

- `build (A)` - buduje set z danych zawartych w A
- `length` - zwraca moc zbioru
- `find (k)` - zwraca element zbioru o kluczu równym k
- `insert (k)` - dodaje element o kluczu k do zbioru
- `delete (k)` - usuwa element o kluczu k ze zbioru
- `find_max` - zwróć element o największym kluczu
- `find_min` - zwróć element o najmniejszym kluczu
- `find_prev` - zwraca element poprzedni od klucza

Struct	build	find	insert / delete	find min / find max	find_prev	sort
unsorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
sorted array	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
pointers list	$\Theta(n)$	$\Theta(n)$	$\Theta(1) / \Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
DAA	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
BST	$\Theta(n)$					

## Binary Search Tree

BST property :

- $x \in T$  -  $x$  jest węzłem drzewa  $T$
- Wówczas każdy  $y$  *in*  $x$ .left ma  $y.key < x.key$
- key  $y$  *in*  $x$ .right ma  $y.key > x.key$

## Inorder Tree Walk

---

**Algorithm 12** Inorder Tree Walk

---

```
1: procedure INORDERTREEWALK( $x \in T$ )
2:   if  $x \neq \text{null}$  then
3:     INORDERTREEWALK( $x$ .left)
4:     print( $x$ )
5:     INORDERTREEWALK( $x$ .right)
6:   end if
7: end procedure
```

---

## Tree Search

---

**Algorithm 13** TreeSearch

---

```
1: procedure TREESEARCH( $x \in T, k$ )
2:   if  $x = \text{null} \vee k = x.\text{key}$  then
3:     return  $x$ 
4:   else if  $k < x.\text{key}$  then
5:     return TREESEARCH( $x.\text{left}, k$ )
6:   else
7:     return TREESEARCH( $x.\text{right}, k$ )
8:   end if
9: end procedure
```

---

### BST - Delete

- $x$  jest liściem - zwolnij pamięć zajmowaną przez  $x$ , wstaw wskaźnik na jego ojca (na niego / na null'a)
- $x$  ma jedno poddrzewo -  $x$  ma syna  $v$  to:
  - zwalniamy pamięć  $x$
  - ojciec  $x$  wskazuje na  $v$
  - $v.p$  wskazuje na  $x.p$
- $x$  ma dwa poddrzewa:
  - znajdź następnika  $x \rightarrow y$
  - zastąp dane  $x$  danymi z  $y$
  - skasuj  $y$

**Twierdzenie:** Niech  $T$  będzie losowym drzewem BST o  $n$ -węzłach. wtedy:

$$E(h(t)) \leq 3\log_2 n = o(\log n)$$

D-d:

Nierówność Jensena: f-wypukła:

$$f(E(x)) \leq E(f(x))$$

Zamiast analizować zmienną losową  $h(t)$  będziemy się zajmować zmienną losową  $H_n$ , będziemy się zajmować  $Y_n = 2^{H_n}$

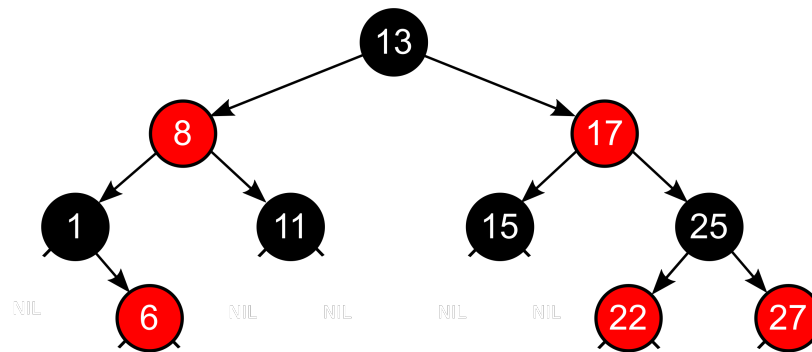
Pokażemy, że  $E(Y_n) = O(n^3)$

$$2^{H_n} \leq E(2^{H_n}) = E(Y_n) = O(n^3) / \log_2$$

$$E(H_n) = 3 \cdot \log_2 n + o(\ln n)$$

## Drzewa czerwono-czarne

- Drzewo czerwono-czarne jest drzewem BST
- Każdy węzeł jest czerwony albo czarny
- Korzeń oraz liście są czarne
- Czerwony węzeł nie może mieć czerwonego ojca
- Każda ścieżka od węzła do liścia ma tę samą liczbę czarnych węzłów (ścieżkę tę będziemy nazywać black-height i oznaczać jako  $bh(x)$ )



**Lemat:** Niech  $T$  będzie drzewem czerwono-czarnym o  $n$  węzłach. Wówczas wysokość drzewa  $T$  jest z góry ograniczona przez:

$$\text{wysokość}(T) \leq 2 \cdot \log_2(n + 1)$$

### RB - Insert

- Wstawiamy węzeł  $z$  w taki sposób jak w BST
- $z.\text{kolor} = \text{czerwony}$
- FixUp (nie chodzi o zespół punkowy)

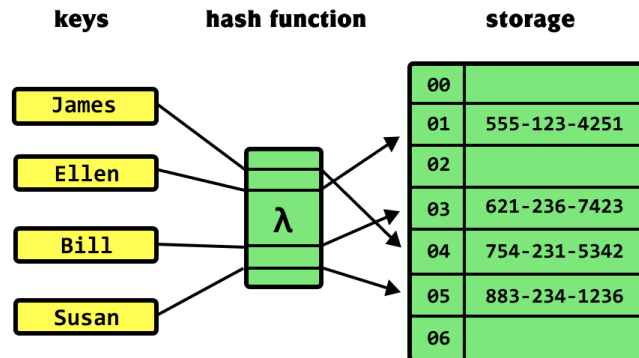
Więcej o drzewach czerwono - czarnych można znaleźć pod linkiem:

<https://inf.ug.edu.pl/~pmp/Z/ASDwyklad/czczWUd.pdf>

## Directed Access Array

- klucze należą do 0, ..., k - 1, k = moc zbioru kluczy
- klucze będziemy utożsamiać z adresami w pamięci komputera

## Hash Tables

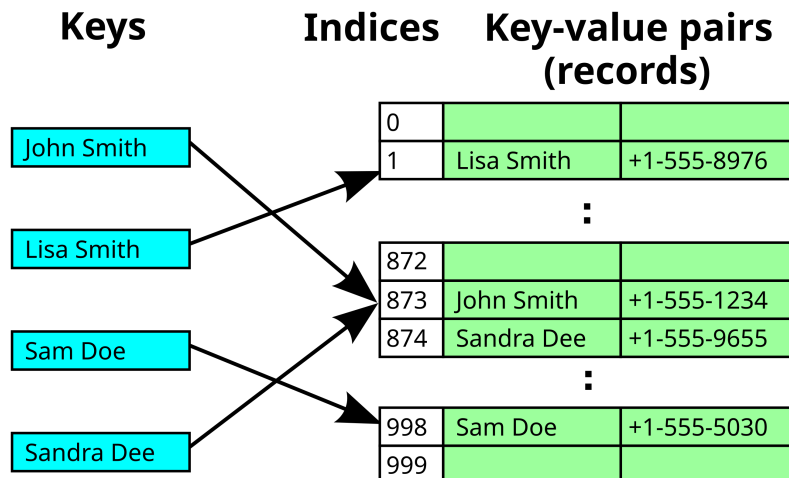


Ponieważ  $k \gg m$  (k - zbiór kluczy, m - wartości hash): funkcja hash nie będzie różnowartościowa:

$$\exists k_1, k_2 : h(k_1) = h(k_2)$$

Aby poradzić sobie z tym problemem:

- open addressing



Chcemy aby funkcja  $h(\text{hash})$  zwracała wyniki z rozkładu jednostajnego na zbiorze 0, ..., m-1

- Division hash function:  $h(k) = k \bmod m$
- Universal hash function:  $H_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$

$$H(p, m) = h_{a,b} : a, b \in 0, \dots, p-1, a \neq 0$$

Własności Universal hash function:

- Universal hash property:

$$P_{h \in H}(h(k_i) = h(k_j)) \leq \frac{1}{m}, \forall k_i, k_j \in \{0, \dots, k-1\}$$

D - d:

zakładamy, że  $h_{a,b}$  ma Universal Hash Property.

$$a \cdot x + b \equiv a \cdot y + b + i \cdot m \bmod p$$

Skoro  $p > k$ ,  $x \neq y \rightarrow x - y \neq 0$  oraz ma odwrotność mod  $p$ .

$$a \equiv i \cdot m \cdot (x - y)^{-1} \bmod p$$

zatem otrzymujemy:

$$P_r(\text{kolizja}) \leq \frac{\lfloor \frac{p-1}{m} \rfloor}{p-1} \leq \frac{\frac{p-1}{m}}{p-1} = \frac{1}{m} \text{ Wartość oczekiwana kolizji jest rzędu } \Theta(1)$$

## Wzbogacanie struktur danych

- Ex - dynamiczne statystyki pozycyjne
  - dynamiczna struktura danych
  - OS-Select(i) - zwróci i-tą statystykę pozycyjną z danych
  - OS-Rank(x) - numer statystyki pozycyjnej X w danych

---

### Algorithm 14 OS-Select

---

```

1: procedure OS-SELECT( $x, i$ )
2:    $k \leftarrow \text{size}(x.\text{left}) + 1$ 
3:   if  $i = k$  then
4:     return  $x$ 
5:   else if  $i < k$  then
6:     return OS-SELECT( $x.\text{left}, i$ )
7:   else
8:     return OS-SELECT( $x.\text{right}, i - k$ )
9:   end if
10: end procedure

```

---



---

### Algorithm 15 OS-Rank

---

```

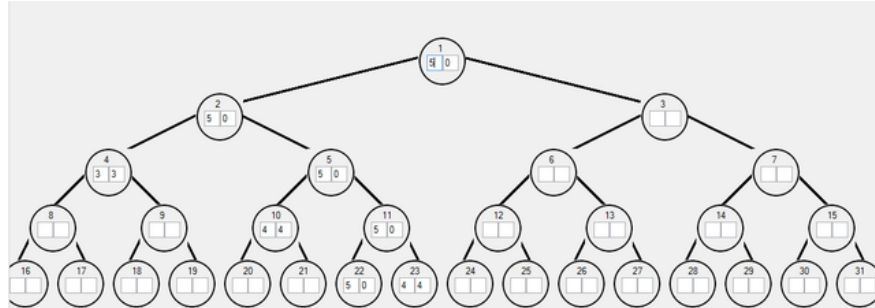
1: procedure OS-RANK( $x$ )
2:    $r \leftarrow \text{size}(x.\text{left}) + 1$ 
3:    $y \leftarrow x$ 
4:   while  $y \neq \text{root}$  do
5:     if  $y = y.\text{parent}.\text{right}$  then
6:        $r \leftarrow r + \text{size}(y.\text{parent}.\text{left}) + 1$ 
7:     end if
8:      $y \leftarrow y.\text{parent}$ 
9:   end while
10:  return  $r$ 
11: end procedure

```

---

- Metodologia wzbogacenia struktur danych
  - Wybierz strukturę
  - Wybrać dodatkową informację, która ma być przechowywana w strukturze
  - Upewnić się, że złożoność obliczeniowa operacji na strukturze danych nie ulegnie pogorszeniu
  - Zaprojektować dodatkowe operacje na strukturze danych, wykorzystującą dodatkową informację

### Drzewa przedziałowe



Własności:

- Przechowuje punkty oraz wszystkie przedziały związane z tymi punktami
- Jest zrównoważone
- Odpowiada na pytania dotyczące przedziałów w czasie logarytmicznym

---

#### Algorithm 16 IntervalSearch

---

```

1: procedure INTERVALSEARCH( $i, T$ )
2:    $x \leftarrow \text{root}(T)$ 
3:   while  $x \neq \text{null}$  and  $(i.\text{low} > x.\text{high} \text{ or } x.\text{low} > i.\text{high})$  do
4:     if  $x.\text{left} \neq \text{null}$  and  $i.\text{low} < x.\text{left}.m$  then
5:        $x \leftarrow x.\text{left}$ 
6:     else
7:        $x \leftarrow x.\text{right}$ 
8:     end if
9:   end while
10:  return  $x$ 
11: end procedure

```

---



### Programowanie dynamiczne

Programowanie dynamiczne to technika projektowania algorytmów polegająca na rozwiązywaniu podproblemów i zapamiętywaniu ich wyników  
Przykład:  $n$ -ta liczba Fibonacciego (memoizacja):

---

#### Algorithm 17 FibonacciMemo

---

```
1: procedure FIBONACCI MEMO( $n, memo$ )
2:   if  $n \leq 1$  then
3:     return  $n$ 
4:   end if
5:   if  $memo[n] \neq -1$  then
6:     return  $memo[n]$ 
7:   end if
8:    $memo[n] \leftarrow \text{FIBONACCI MEMO}(n - 1, memo) + \text{FIBONACCI MEMO}(n - 2, memo)$ 
9:   return  $memo[n]$ 
10: end procedure
```

---

### Problem wydawania reszty

Dane:  $c_1 < c_2 < \dots < c_k$  - zbiór nominałów  $\in \mathbb{N}$

Szukamy minimalnej liczby monet, które sumują się do kwoty  $n$

Niech  $L(i)$  będzie minimalną liczbą monet dla reszty  $i$

$$L(i) = 1 + \min_{1 \leq j \leq n} \{L(i - c_j) : c_j \leq i\}$$

### Problem plecakowy

Dane:  $n$  przedmiotów,  $w_i$  - waga przedmiotu  $i$ ,  $v_i$  - wartość przedmiotu  $i$

Ograniczenie górne na pojemność plecaka:  $W$

Szukamy maksymalnej wartości przedmiotów, które zmieszczą się w plecaku:  $\mathbb{I} \in \{1, \dots, n\}$  takie, że:

- $\sum_{i \in \mathbb{I}} w_i \leq W$
- $\sum_{i \in \mathbb{I}} v_i$  jest maksymalne

---

#### Algorithm 18 Algorytm plecakowy

---

```
1: procedure KNAPSACK( $W, w, c, n$ )
2:   utwórz tablicę  $A[0 \dots W]$ 
3:   for  $i \leftarrow 0$  to  $W$  do
4:      $A[i] \leftarrow 0$ 
5:   end for
6:   for  $i \leftarrow 0$  to  $W$  do
7:     for  $j \leftarrow 1$  to  $n$  do
8:       if  $w[j] \leq i$  then
9:          $A[i] \leftarrow \max(A[i], A[i - w[j]] + c[j])$ 
10:      end if
11:    end for
12:  end for
13:  return  $A[W]$ 
14: end procedure
```

---