

Algorytmy i Struktury Danych

Wojciech Typer

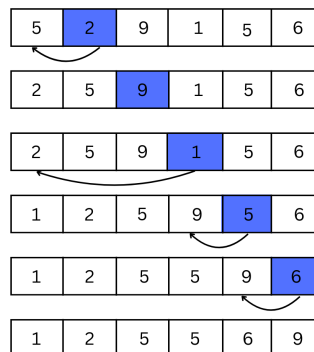
Algorithm 1 Insertion Sort

```
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

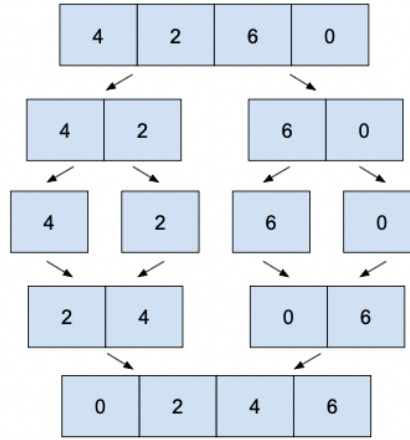
```
1: procedure MERGESORT( $A, 1, n$ )
2:   if  $|A[1..n]| == 1$  then
3:     return  $A[1..n]$ 
4:   else
5:      $B = \text{MergeSort}(A, 1, \lfloor n/2 \rfloor)$ 
6:      $C = \text{MergeSort}(A, \lfloor n/2 \rfloor, n)$ 
7:     return  $\text{Merge}(B, C)$ 
8:   end if
9: end procedure
```

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

```

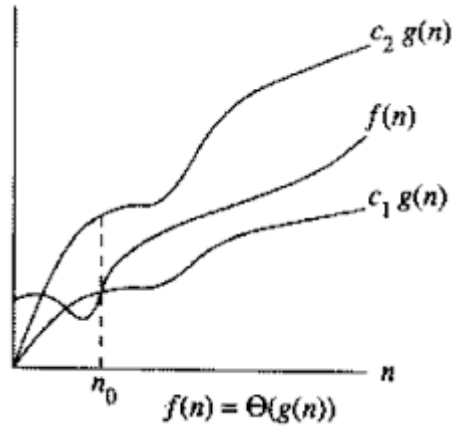
1: procedure ITERATIVEMERGESORT( $A[1..n]$ )
2:   for  $size = 1$  to  $n - 1$  by  $size \times 2$  do
3:     for  $left = 0$  to  $n - 1$  by  $2 \times size$  do
4:        $mid \leftarrow \min(left + size - 1, n - 1)$ 
5:        $right \leftarrow \min(left + 2 \times size - 1, n - 1)$ 
6:       MERGE( $A$ ,  $left$ ,  $mid$ ,  $right$ )
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 $O: f(n) = O(g(n)) \rightarrow (\exists c > 0)(\exists n_0 \in N) : \forall n \geq n_0 \rightarrow 0 \leq f(n) \leq c \cdot g(n)$



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

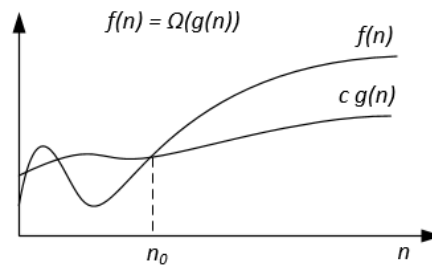
Notacja asymptotyczna - własności

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

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

Notacja Ω

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



Notacja Ω - własności

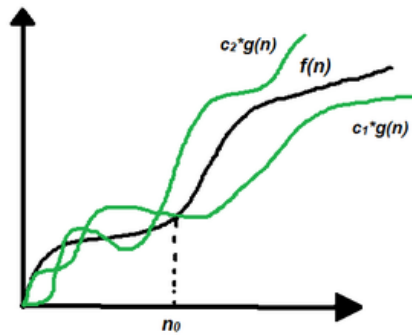
a) $n^3 = \Omega(2n^2)$

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

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

Notacja Θ

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



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

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

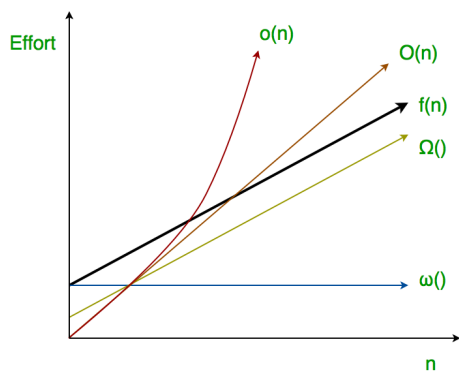
a) $117n \log(n) = o(n^2)$

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

Notacja ω

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

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



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, $(\forall (k < n))(T(k) \leq c * k^3) =$

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

$$= cn^3 - \frac{1}{2}cn^3 + n = cn^3 - \left(\frac{1}{2}cn^3 - n\right) \leq cn^3$$

Pokazaliśmy, że $T(n) = O(n^2)$

Spróbujmy wzmocnić zał. indukcyjne: $T(n) \leq c_1n^2 - c_2n$

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

$$= c_1n^2 - 2c_2n + n = c_1n^2 - (2c_2 - 1)n \leq c_1n^2 - c_2n$$

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

Wówczas otrzymamy $T(1) = O(1) \leq c_11^2 - c_21$

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

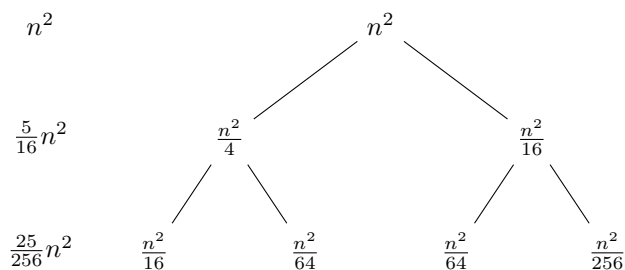
$$T(2^m) = 2T(2^{m/2}) + m \rightarrow 2S(m/2) + m$$

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

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

Drzewo rekursji

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



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

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

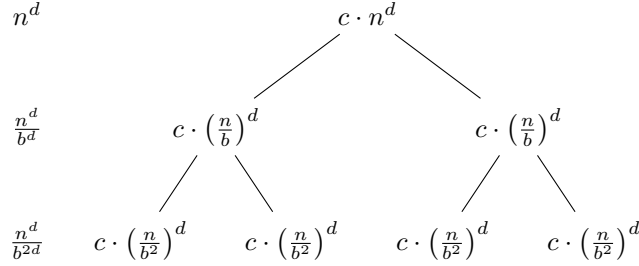
Wówczas:

- $\Theta(n^d)$, 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

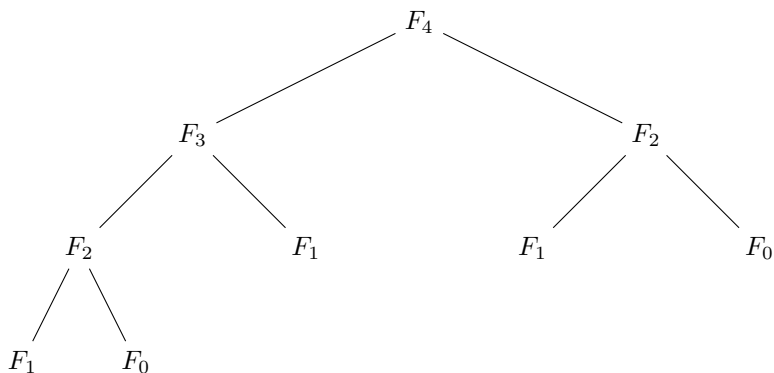
Metoda rekurencyjna:

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

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
```

