Algorytmy i Struktury Danych

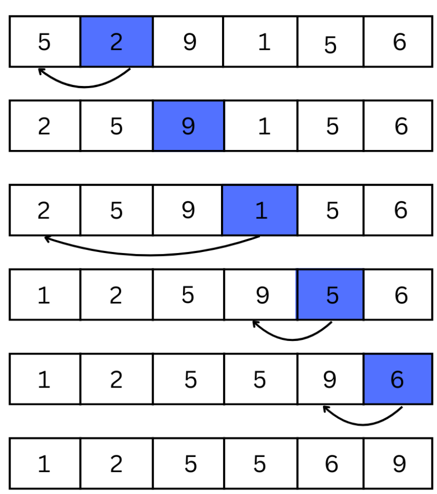
Wojciech Typer

Algorithm 1 Insertion Sort

---

procedure InsertionSort(A, n)
 for $i = 1$ to $n - 1$ **do**
  $key = A[i]$
  $j = i - 1$
  while $j \geq 0$ and $A[j] > key$ **do**
   $A[j + 1] = A[j]$
   $j = j - 1$
  end while
  $A[j + 1] = key$
 end for
end procedure

Złożoność czasowa: $O(n^2)$
Best case: w najlepszym przypadku złożoność czasowa będzie wynosić $O(n)$
Złożoność pamięciowa: $O(1)$



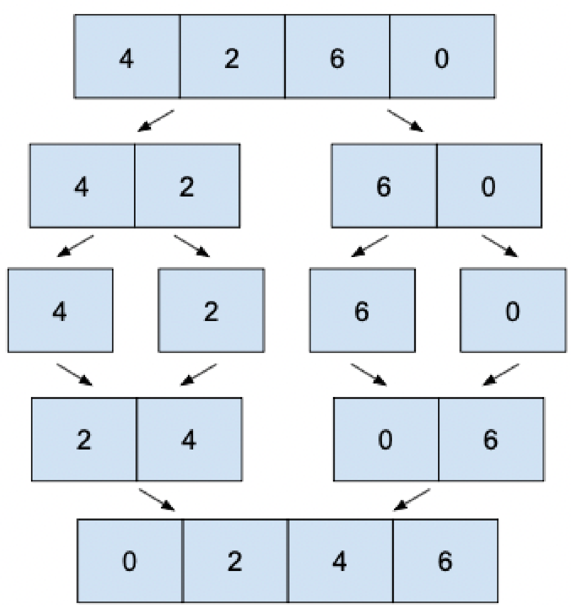
Algorithm 2 Merge Sort

procedure MergeSort(A, 1, n)
  if |A[1..n]| == 1 **then**
    return A[1..n]
  else
    $B = \text{MergeSort}(A, 1, \lfloor n/2 \rfloor)$
    $C = \text{MergeSort}(A, \lfloor n/2 \rfloor, n)$
    return Merge(B, C)
  end if
end procedure

Algorithm 3 Merge

procedure Merge(X[1..k], Y[1..n])
  if $X = \emptyset$ **then**
    return Y
  else if $Y = \emptyset$ **then**
    return X
  else if $X[1] \leq Y[1]$ **then**
    return $[X[1]] \times \text{Merge}(X[2..k], Y[1..n])$
  else
    return $[Y[1]] \times \text{Merge}(X[1..k], Y[2..n])$
  end if
end procedure

Złożoność czesowa 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.

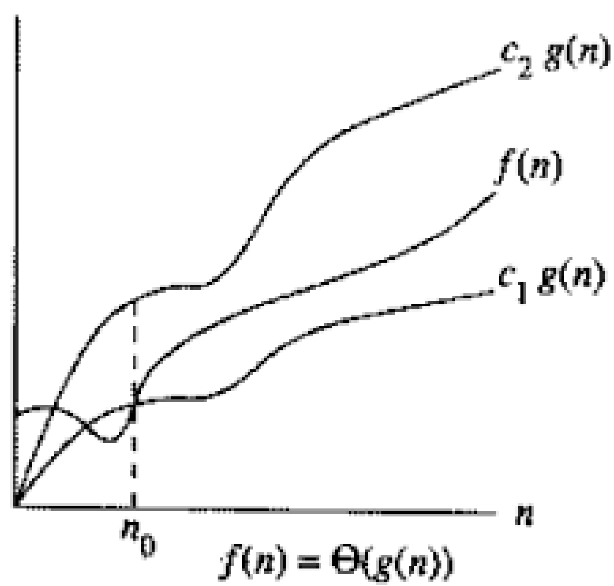
Algorithm 4 IterativeMergeSort

```
procedure ITERATIVEMERGESORT(A[1..n])
    for size = 1 to n − 1 by size × 2 do
        for left = 0 to n − 1 by 2 × size do
            mid ← min(left + size − 1, n − 1)
            right ← min(left + 2 × size − 1, n − 1)
            MERGE(A, left, mid, right)
        end for
    end for
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)) $\rightarrow (\exists c > 0)(\exists n_0 \in N) : \forall n \geq n_0 \rightarrow 0 \leq f(n) \leq c \cdot g(n)$



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

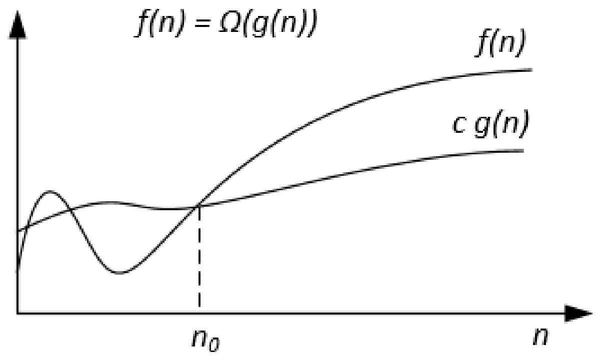
Notacja asymptotyczna - własności

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 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

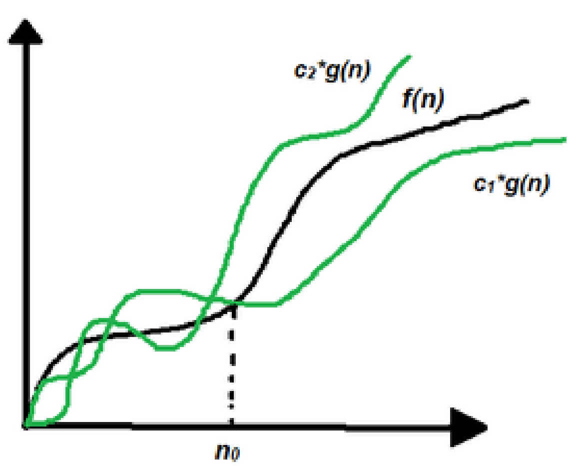
a) $n^3 = \Omega(2n^2)$

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

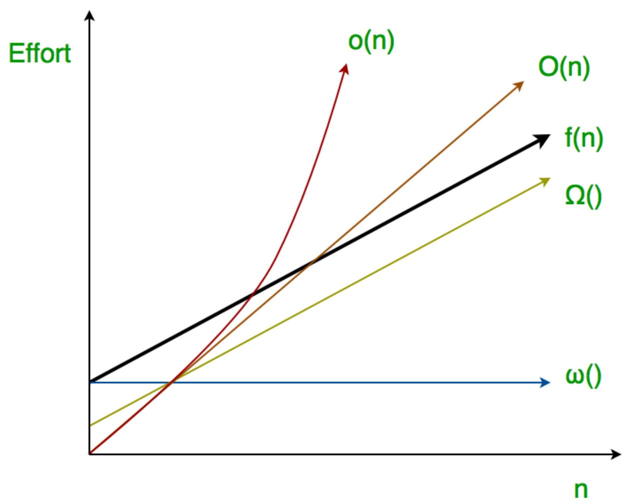
a) $117nlog(n) = o(n^2)$

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:

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

Pokazaliśmy, że $T(n) = O(n^2)$

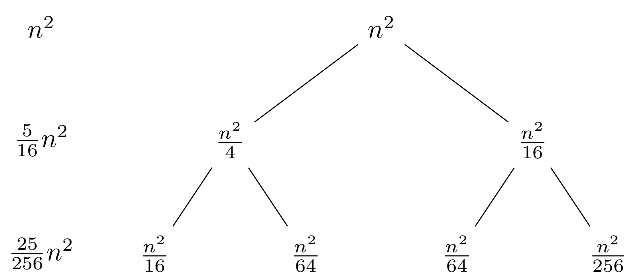
Spróbujmy wzmocnić zał. indukcyjne: $T(n) \leq c_1 n^2 - c_2 n$
$T(n) \leq 4T(\frac{n}{2}) + n \leq 4(c_1(\frac{n}{2})^2 - c_2\frac{n}{2}) + n =$
$= c_1 n^2 - 2c_2 n + n = c_1 n^2 - (2c_2 - 1)n \leq c_1 n^2 - c_2 n$
Musimy dobrać takie $c_1 i c_2$, aby $2c_1 \geq c_2$
Wówczas otrzymamy $T(1) = O(1) \leq c_1 1^2 - c_2 1$

Przykład 2:

$T(n) = 2T(\sqrt{n}) + log(n)$
Załóżmy, że n jest potęgą dwójki $n = 2^m \rightarrow m = log(n)$
$T(2^m) = 2T(2^{m/2}) + m$
oznaczmy $T(2^m) = S(m)$
$T(2^m) = 2T(2^{m/2}) + m \rightarrow 2S(m/2) + m$
$S(m) = O(mlog(m))$
$T(n) = O(log(n)log(log(n)))$ (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 urzywać do dowodzenia złożoności algorytmów. Jest to jedynie intuicyjne podejście do problemu. Formmalnie 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:

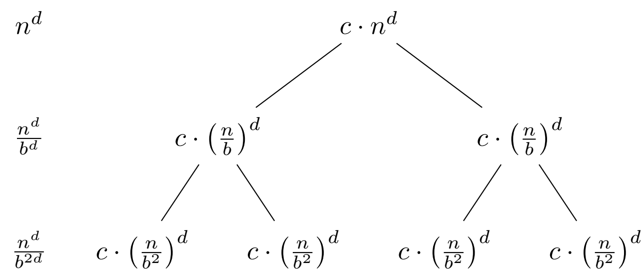
$$T(n) = aT(\frac{a}{b}) + \Theta(n^d)$$

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:

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



1. suma kosztoów w k–tym kroku
$$a^k c(\frac{n}{b^k})^d = c(\frac{a}{b^d})^k n^d$$
gdzie $c(\frac{n}{b^k})^d$ to koszt jednego podproblemu w k–tym kroku

2. obliczenie wysokości drzewa:
$$\frac{n}{b^h} = 1 \to h = \log_b n$$

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

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 kożystając 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. Rozwiazanie rekurencyjnie mniejsze podpoblemy
3. połącz rozwiązania podproblemów w celu rozwiązania problemu wejściowego

Algorytm – Binary Search

- **Input**: posortowania 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

```
procedure BinarySearch(A, x)
    l = 1
    r = |A|
    while l ≤ r do
        m = ⌊(l+r)/2⌋
        if A[m] = x then
            return m
        else if A[m] < x then
            l = m + 1
        else
            r = m − 1
        end if
    end while
    return 0
end procedure
```

- **Asypmtotyka** Algorytm spełnia następująca rekurencje:

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

Rozwiązując za pomocą **Master Theorem** otrzymujemy:

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

Divide & Conquer

Problem: Obliczenie x^n.

Rozwiązanie naiwną metodą iteracyjną:

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

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

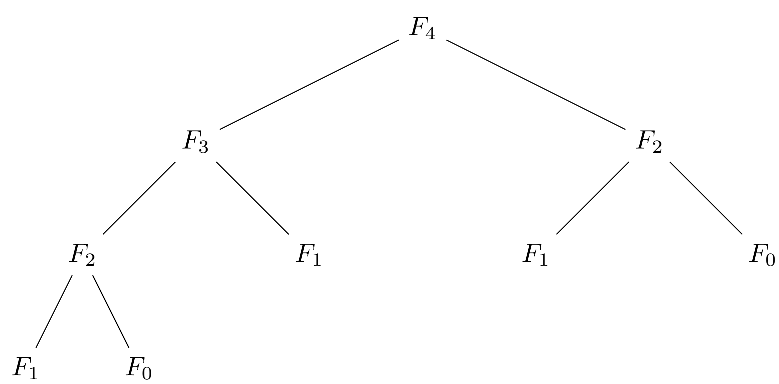
Metoda rekurencyjna:

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

Ma ona złożoność wykładniczą:

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

Drzewo rekurencyjne dla F_4:



Wzór jawny:

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

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:

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

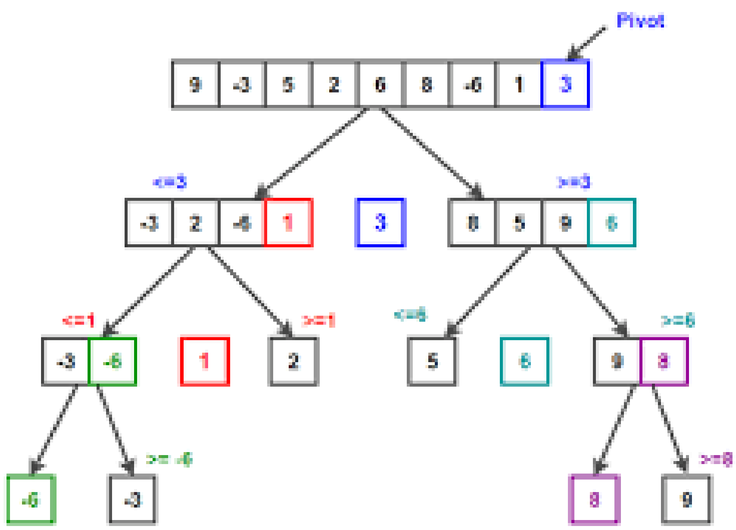
Algorithm 6 Multiply - Mnożenie dużych liczb binarnych metodą Gaussa

procedure MULTIPLY(x, y)
  $n \leftarrow \max(|x|, |y|)$
  if $n = 1$ **then**
    return $x \cdot y$
  end if
  $m \leftarrow \lceil n/2 \rceil$
  $x_L, x_R \leftarrow$
  $y_L, y_R \leftarrow$
  $p_1 \leftarrow$ MULTIPLY(x_L, y_L)
  $p_2 \leftarrow$ MULTIPLY(x_R, y_R)
  $p_3 \leftarrow$ MULTIPLY$((x_L + x_R), (y_L + y_R))$
  return $p_1 \cdot 2^{2m} + (p_3 - p_1 - p_2) \cdot 2^m + p_2$
end procedure

QuickSort

Algorithm 7 QuickSort - Sortowanie szybkie

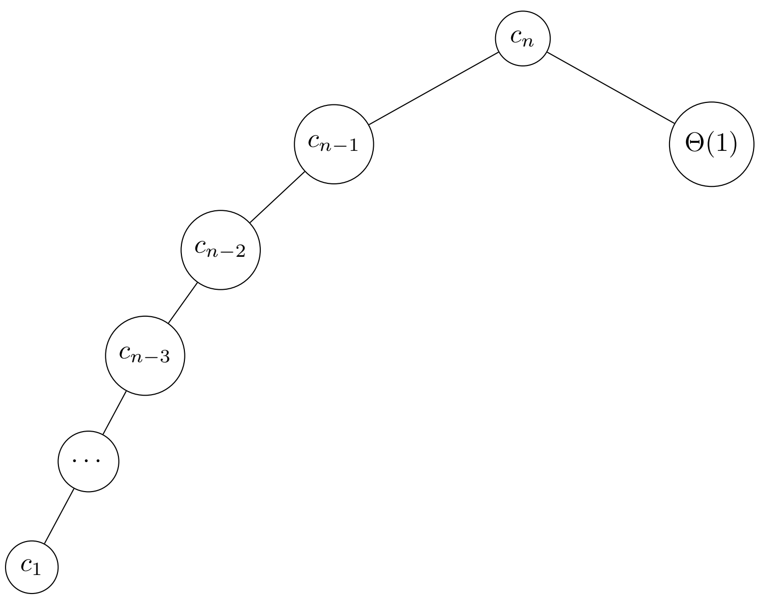
```
procedure QUICKSORT(A, low, high)
    if low < high then
        p ← PARTITION(A, low, high)
        QUICKSORT(A, low, p - 1)
        QUICKSORT(A, p + 1, high)
    end if
end procedure

procedure PARTITION(A, low, high)
    pivot ← A[high]
    i ← low − 1
    for j ← low to high − 1 do
        if A[j] ≤ pivot then
            i ← i + 1
            SWAP(A[i], A[j])
        end if
    end for
    SWAP(A[i + 1], A[high])
    return i + 1
end procedure
```



Algorithm 8 Hoare Partition

```
procedure HOARE_PARTITION(A, p, q)
    pivot ← A[⌊(p+q)/2⌋]
    i ← p − 1
    j ← q + 1
    while true do
        repeat
            i ← i + 1
        until A[i] ≥ pivot
        repeat
            j ← j − 1
        until A[j] ≤ pivot
        if i ≥ j then
            return j
        end if
        swap(A[i], A[j])
    end while
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.

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

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

Liczymy wartość oczekiwaną:

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

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

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

$$= \frac{2}{n} \sum_{k=0}^{n-1} E(T_k) + (n-1)$$

$$\Rightarrow E(T_n) = \Theta(n \log n)$$

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

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

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

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

liczymy wartość oczekiwaną:

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

Dual pivot quicksort

$< LP$	LP	$LP \leq \ \& \ \leq RP$	RP	$RP <$

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

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

Yaroslavsky dual pivot qs
E(liczba porównań w partition) $\approx \frac{19}{12}n$
E(liczba porównań w Dual Pivot qs Yaroslavsky) $\approx 1.9nlogn$

Strategia count
E(liczba porównań w Count Partition) $\approx \frac{3}{2}n$
E(liczba porównań w Dual Pivot qs z count) $\approx 1.8nlogn$

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

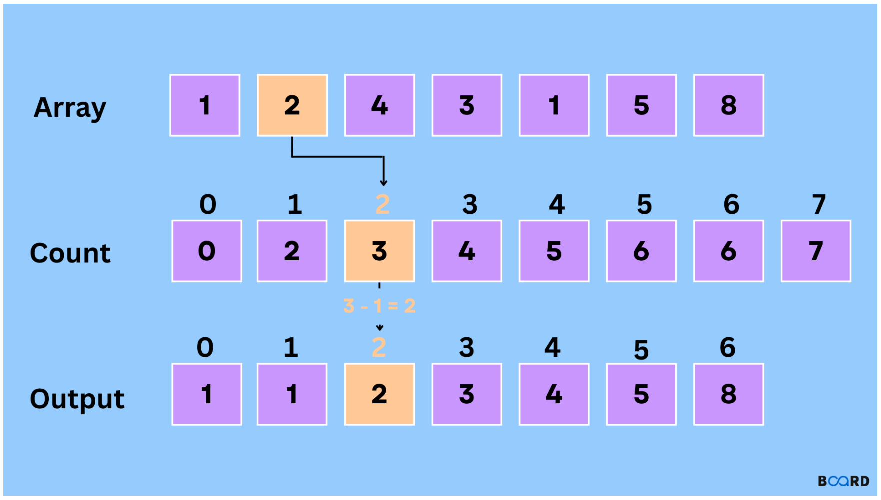
D-d:

- 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
  $h \leq \log_2 n!$
  $lgn! = lg(\sqrt{s\pi n}(\frac{n}{e})^n(1+o(1)))$
  $lg(\frac{n}{e})^n + lg(\sqrt{(}2\pi n)(1+o(1)))$
  $nlogn - nlge + lg(\sqrt{2\pi n}(2+o(1))) = \Omega(nlogn)$

Sortowanie:
Input: $|a| = n, \forall i \in \{1, ..., k\}$
Output: posortowana rosnąco tablica A

Algorithm 9 CountingSort

procedure COUNTING_SORT(A, n, k)
  for $i = 1$ **to** k **do**
    $C[i] \leftarrow 0$
  end for
  for $i = 1$ **to** n **do**
    $C[A[i]] \leftarrow C[A[i]] + 1$
  end for
  for $i = 2$ **to** k **do**
    $C[i] \leftarrow C[i] + C[i-1]$
  end for
  for $i = n$ **downto** 1 **do**
    $B[C[A[i]]] \leftarrow A[i]$
    $C[A[i]] \leftarrow C[A[i]] - 1$
  end for
  return B
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

```
procedure RADIX_SORT(A, n, d)
    for i = 1 to d do
        counting_sort(A, n, 9)
    end for
    return A
end procedure
```

Złożoność obliczeniowa RadixSorta

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

Zatem RadixSort będzie miał złożoność oblicznieiową:
$\Theta(\frac{b}{r} \cdot (n + 2^r))$
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:

- $k = 1 \rightarrow O(n)$
- $k = n \rightarrow O(n)$
- $k = \lfloor \rightarrow$ sortowanie $O(n \log n)$

Algorithm 11 RandomSelect

procedure RANDOM_SELECT(A, p, q, i)
    if $p = q$ **then**
        return $A[p]$
    end if
    $r \leftarrow$ RandPartition(A, p, q)
    $k \leftarrow r - p + 1$
    if $i = k$ **then**
        return $A[r]$
    else if $i < k$ **then**
        return RANDOM_SELECT(A, p, $r - 1$, i)
    else
        return RANDOM_SELECT(A, $r + 1$, q, $i - k$)
    end if
end procedure

Select algorithm

- dzielimy A[p..q] na $\frac{n}{\lfloor 5 \rfloor}$ pięcioelementowych częsci oraz ostanią część na ≤ 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 $\rightarrow T(\lceil \frac{n}{5} \lceil)$
- 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: $t(n) = T(\lceil \frac{n}{5} \lceil) + \Theta(?)$

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 - zwróc 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

```
procedure InorderTreeWalk(x ∈ T)
    if x ≠ null then
        InorderTreeWalk(x.left)
        print(x)
        InorderTreeWalk(x.right)
    end if
end procedure
```

Tree Search

Algorithm 13 TreeSearch

```
procedure TreeSearch(x ∈ T, k)
    if x = null ∨ k = x.key then
        return x
    else if k < x.key then
        return TreeSearch(x.left, k)
    else
        return TreeSearch(x.right, k)
    end if
end procedure
```

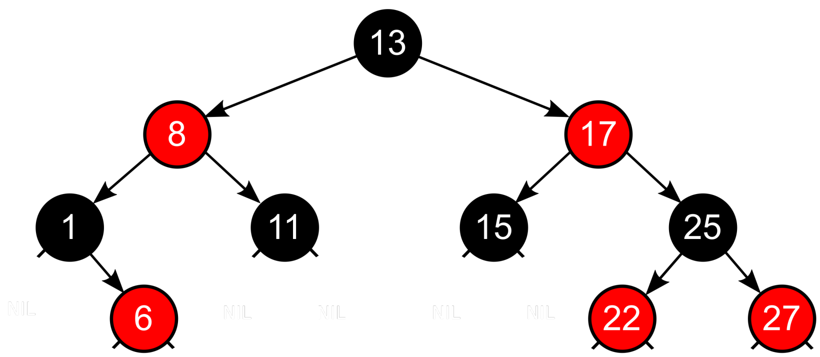
BST - Delete

- x jest liścciem - 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 -> y
  - zastąp dane x danymi z y
  - skasuj y

Twierdzenie: Niech T będzie losowym drzewem BST o n-węzłach. wtedy:
$E(h(t)) \leq 3log_2 n = o(logn)$
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 $E(Y_n) = O(n^3)$
$2^{H_n} \leq E(2^{H_n}) = E(Y_n) = O(n^3) // log_2$
$E(H_n) = 3 \cdot log_2 n + o(lnn)$

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.kolor = czerwony
- FixUp (nie chodzi o zespół punkowy)

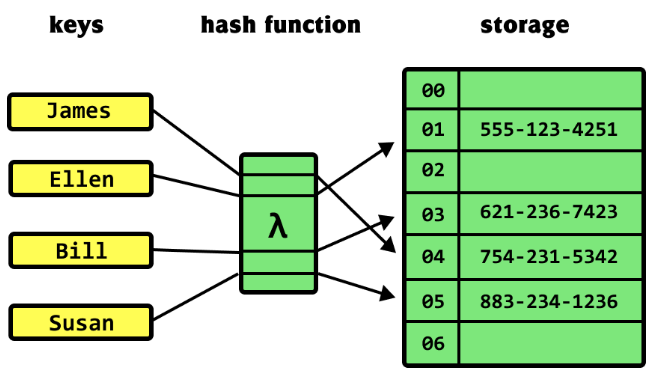
Więcej o drzewacz czerwono - czarnych można znaleźć pod linkiem:

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

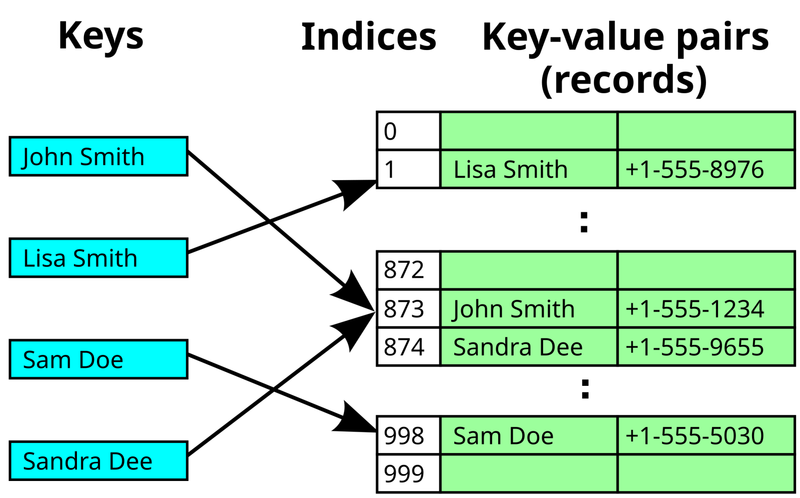


Ponieważ k » m (k - zbiór kluczy, m - wartości hash): funkcja hash nie będzie różnowartościowa:

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

Aby poradzić sobie z tym problemem:

- open adressing



Chcemy abu funkcja h(hash) zwracała wyniki z rozkładu jednostajnego na zbiorze 0, ..., m-1

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

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

Własności Universal hash function:

- Universal hash property:

$$P_{h \in H}(h(k_i) = h(k_j)) \leq \frac{1}{m}, \forall k_1, k_j \in \{0, \text{..., k-1}\}$$

  D - d:
  zakładamy, że $h_{a,b}$ ma Universal Hash Property.
  $a \cdot x + b \equiv a \cdot y + b + i \cdot m \text{mod p}$
  Skoro p > k, $x \neq y \rightarrow x - y \neq 0$ oraz ma odwrotność mod p.
  $a \equiv i \cdot m \cdot (x - y)^{-1} \text{mod p}$
  zatem otrzymujemy:
  $P_r(\text{kolizja}) \leq \frac{\lfloor \frac{p-1}{m} \rfloor}{p-1} \leq \frac{\frac{p-1}{m}}{p-1} = \frac{1}{m}$ Wartość oczekiwana kolizji jest rzędu $\Theta(1)$

Wzbogacanie strukur danych

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

Algorithm 14 OS-Select

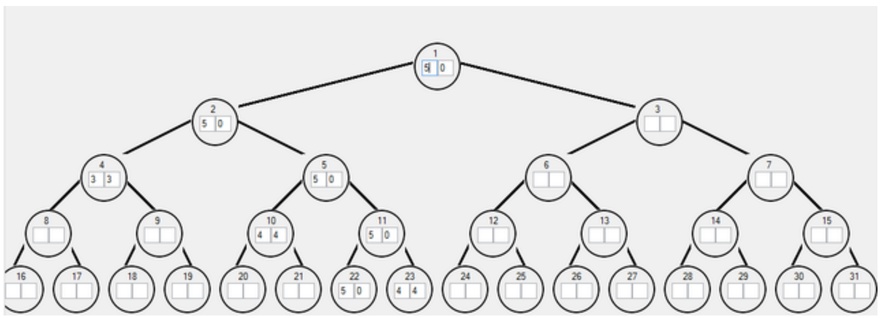
```
procedure OS-Select(x, i)
    k ← size(x.left) + 1
    if i = k then
        return x
    else if i < k then
        return OS-Select(x.left, i)
    else
        return OS-Select(x.right, i − k)
    end if
end procedure
```

Algorithm 15 OS-Rank

```
procedure OS-Rank(x)
    r ← size(x.left) + 1
    y ← x
    while y ≠ root do
        if y = y.parent.right then
            r ← r + size(y.parent.left) + 1
        end if
        y ← y.parent
    end while
    return r
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

```
procedure IntervalSearch(i, T)
    x ← root(T)
    while x ≠ null and (i.low > x.high or x.low > i.high) do
        if x.left ≠ null and i.low < x.left.m then
            x ← x.left
        else
            x ← x.right
        end if
    end while
    return x
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

```
procedure FibonacciMemo(n, memo)
    if n ≤ 1 then
        return n
    end if
    if memo[n] ≠ −1 then
        return memo[n]
    end if
    memo[n] ← FibonacciMemo(n − 1, memo) + FibonacciMemo(n − 2, memo)
    return memo[n]
end procedure
```

Problem wydawania reszty
Dane: $c_1 < c_2 < ... < c_k$ - 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: $\mathbb{I} \in \{1, .., n\}$ takie, że:

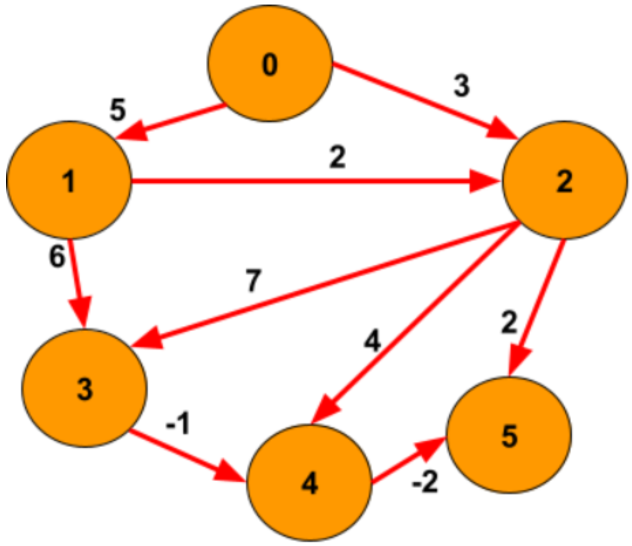
- $\sum_{i \in \mathbb{I}} w_i \leq W$
- $\sum_{i \in \mathbb{I}} v_i$ jest maksymalne

Algorithm 18 Algorytm plecakowy

```
procedure Knapsack(W, w, c, n)
    utwórz tablicę A[0 . . . W]
    for i ← 0 to W do
        A[i] ← 0
    end for
    for i ← 0 to W do
        for j ← 1 to n do
            if w[j] ≤ i then
                A[i] ← max (A[i], A[i − w[j]] + c[j])
            end if
        end for
    end for
    return A[W]
end procedure
```

Grafy jako struktura danych

Grafy w informatyce to nieliniowa struktura danych, składająca się z wierzchołków (węzłów) i krawędzi, które je łączą. Modelują relacje między obiektami, jak np. sieci społecznościowe, sieci komputerowe czy mapy drogowe.
Krawędzie mogą być skierowane (grafy skierowane) lub nieskierowane (grafy nieskierowane).



W grafach możemy wyróżnić pewne węzły na podstawie ich krawędzi.
Jeżeli do węzła u nie wchodzą żadne krawędzie, to nazywamy go źródłem.
Jeżeli z węzła u nie wychodzą żadne krawędzie, to nazywamy go ujściem.
Liczenie najkrótszej ścieżki w DAG (skierowany graf acykliczny)
można wykonać w czasie $O(V+E)$, gdzie V to liczba węzłów, a E to liczba krawędzi.

$$L(A) = \max \begin{cases} L(S) + w(S, A) \\ L(C) + w(C, A) \end{cases}$$

W pseudokodzie można to zaprezentować w następujący sposób:

Algorithm 19 Najkrótsza ścieżka w DAG

procedure ShortestPath(G, E)

$\triangleright$ **Inicjalizacja:** $\Theta(|V|)$

  for $v \in V$ **do**
    $L(v) \leftarrow \infty$
  end for
  $L(S) \leftarrow 0$

$\triangleright$ **Meat algorytmu:** $\Theta(|E|)$

  for $v \in V \setminus \{S\}$ **do**
    $L(v) \leftarrow \min_{(u,v)\in E}\{L(u) + w(u,v)\}$
  end for

  return L
end procedure

Całkowita złożoność algorytmu to $\Theta(|V| + |E|)$, ponieważ w najgorszym przypadku musimy przejść przez wszystkie węzły i krawędzie. **Edit Distance**

Przykładem wykorzystania tego problemu jest użycie go w spellcheckerach.
Sugerują one poprawki do błędnie napisanych słów.

- **Input**: w_1, w_2 – słowa, Σ – alfabet
- **Output**: $EditDistance(w_1, w_2)$ – minimalna liczba operacji potrzebnych do przekształcenia w_1 w w_2
- **Problem**: Znaleść minimalną liczbę operacji potrzebnych do przekształcenia w_1 w w_2

Zobaczmy to najpierw na **przykładzie**:

$$w_1 = \text{SNOWY}, \quad w_2 = \text{SUNNY}.$$

$d_{i,j}$		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2	3	4
N	2	1	1	1	2	3
O	3	2	2	2	2	3
W	4	3	3	3	3	3
Y	5	4	4	4	4	3

Stąd

$$\text{EditDistance}(\text{SNOWY}, \text{SUNNY}) = d_{5,5} = 3,$$

czyli potrzebne są trzy podstawienia (np. $N \to U$, $O \to N$, $W \to N$).

Niech $E(i,j)$ – edit distance $w_1[1 \dots i], w_2[1 \dots j]$. Mamy następujące możliwości

- dodanie litery do $w_1 \leftarrow E(i, j-1) + 1$
- usuniecie litery z $w_2 \leftarrow E(i-1, j) + 1$
- podmienienie litery w $w_2 \leftarrow E(i-1, j-1) + 1$
- bez zmian $w_1 \leftarrow E(i-1, j-1)$

Przy wykonywaniu tego algorytmu musimy brać minimum z tych czterech możliwości, a więc

$$E(i,j) = \min \begin{cases} E(i,j-1)+1 \\ E(i-1,j)+1 \\ E(i-1,j-1)+1 \\ E(i-1,j-1) \end{cases}$$

A jak wygląda graf?

$d_{i,j}$		0	1	2	3	4	...
	0	1	2	3	4	5	...
1	1	0	1	2	3	4	...

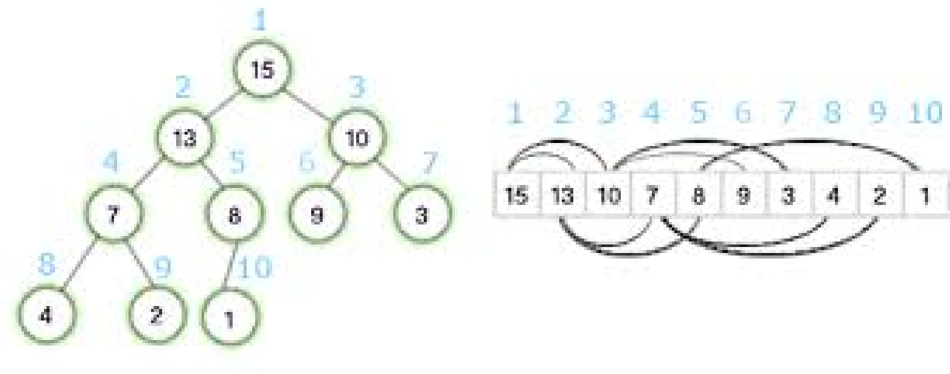
Pseudokod:

Algorithm 20 Edit Distance

procedure EditDistance(w_1, w_2)
  $n \leftarrow |w_1|$
  $m \leftarrow |w_2|$
  for $i = 0$ to n **do**
    for $j = 0$ to m **do**
      if $i = 0$ **then**
        $d(i,j) = j$
      else if $j = 0$ **then**
        $d(i,j) = i$
      else
        $$d(i,j) = \min \begin{cases} d(i-1,j)+1 \\ d(i,j-1)+1 \\ d(i-1,j-1)+1 \\ d(i-1,j-1) \end{cases}$$
      end if
    end for
  end for
  return $d(n,m)$
end procedure

Kopiec binarny

- Pełne drzewo binarne przetrzymywane w tablicy



Własność kopca (maksymalnego)

- $\forall i$ A[parent[i]] > A[i]
- Wysokość węzła = długość najdłuższej prostej ścieżki od tego węzła do liścia

Algorithm 21 Heapify

procedure HEAPIFY(a)
    $largest \leftarrow a$
    if $2a \leq size$ **and** $H[2a] > H[largest]$ **then**
        $largest \leftarrow 2a$
    end if
    if $2a + 1 \leq size$ **and** $H[2a+1] > H[largest]$ **then**
        $largest \leftarrow 2a + 1$
    end if
    if $largest \neq a$ **then**
        Zamień $H[largest]$ i $H[a]$
        HEAPIFY($largest$)
    end if
end procedure

Złożoność obliczeniowa procedury Heapify wynosi: $O(logn)$

Build Heap

```
procedure BUILD-HEAP
    for i ← size down to 1 do
        HEAPIFY(i)
    end for
end procedure
```

Fakt: W n- elementowym kopcu binarnym mamy co najwyżej
$\lceil \frac{n}{2^{h-1}} \rceil$ węzłów o wysokości h

Kolejka priorytetowa

- Insert (Q, x)
- Maximum (Q)
- ExtractMax (Q) - zwraca element o największym priorytecie i usuwa go z Q
- Increase / Decrease Key (Q, x ,y)
- Delete (Q, x)
- Union (Q1, Q2) - łączy dwie kolejki priorytetowe w jedną

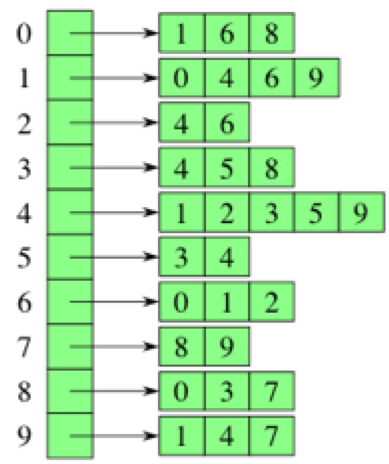
Procedure	**Binary Heap**	**Binomial Heap**	**Fibonacci Heap**
Insert	$O(\log n)$	$O(\log n)$	$O(1)$
Maximum	$O(1)$	$O(\log n)$	$O(1)$
ExtractMax	$O(\log n)$	$O(\log n)$	$O(\log n)$
IncreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$
DecreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
Union	$O(n)$	$O(\log n)$	$O(1)$

Złożoności obliczeniowe w przypadku kopca Fibonacciego zostały podane jako wartości zamortyzowane - koszt operacji jest średnią z wielu kosztów, pojedyncze wartości mogą być czasem wolniejsze

Graf

$$G = (V, E)$$
$$V \rightarrow \text{zbiór wierzchołków } 1..n$$
$$E \in \{\{i, j\} : i, j \in V, i \neq j\}$$

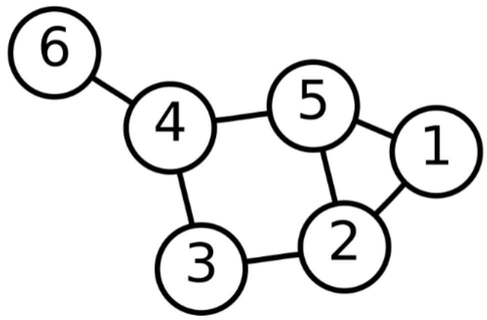
W grafie nieskierowanym nie rozróżniamy i i j, traktujemy to jako zbiór.
W grafie skierowanym i i j traktujemy jako parę, więc kolejność jest istotna.
W oznaczeniach będziemy używać: $|V| = n$ i $|E| = m$



Powyższa grafika to lista sąsiedztwa, jest to reprezentacja grafu,
umożliwiająca jego obróbkę z użyciem programów komputerowych.

Złożoność pamięciowa: O(n + m) = O(|V| + |E|) - wielkość grafu

Następnym sposobem reprezentacji grafu jest macierz sąsiedztwa:



macierz sąsiedztwa jest następująca:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Algorithm 22 Explore

```
procedure EXPLORE(G, v)
    visited(v) ← true
    PREVISIT(v)
    for each (v, u) ∈ E do
        if not visited(u) then
            EXPLORE(G, u)
        end if
    end for
    POSTVISIT(v)
end procedure
```

Powyższa procedura przeszukuje graf w głąb, dzięki niej
Możemy dotrzec do wszystkich wierzchołków z danego wierzchołka

Algorithm 23 DFS (Depth First Search)

```
procedure DFS(G)
    for each v ∈ V do
        visited(v) ← false
    end for
    for each v ∈ V do
        if not visited(v) then
            Explore(G, v)
        end if
    end for
end procedure
```

Przeszukanie grafu w głąb (DFS)

Własność 1: $u, v \in V$: pre(v) < pre(u) < post(u) < post(v)
$u, v \in V$: pre(v) < post(v) < pre(u) < post(u)
Własność 2: $(u, v) \in E$ Tree / forward edge: pre(u) < pre(v) < post(v) < post(u)
Backedge: pre(v) < pre(u) < post(u) < post(v)
Cross edge: pre(v) < post(v) < pre(u) < post(u)

Własność 3: W grafie skierowanym istenie cykl wtedy, gdy DFS wykryje back edge

Sortowanie topologiczne skierowanych grafów acykicznych (dags)
Input: $G = (V, E) \rightarrow$ acykiczny, skierowany
Output: $(V, \prec) \rightarrow$ porządek topologiczny

- Wykonujemy DFS, zapisując pre i post
- Zwróć wierzchołki w malejącym porządku po post()

Złożoność obliczeniowa: DFS: O(|V| + |E|); punkt drugi: O(|V| log|V|)

W grafie skierowanym g+ (V, E) powiemy, że wierzchołki $u, v \in V$
są połączone, jeśli istnieje śnieżka z u do v i z v do u