

Logarytmy

$\log_a b + \log_a c = \log_a (b \cdot c); \log_a b - \log_a c = \log_a \left(\frac{b}{c}\right); a^{\log_a b} = b; n \cdot \log_a b =$

$\log_a b^n; \log_a b = \frac{\log_c b}{\log_c a}; a^{\log_b n} = n^{\log_b a}$

Notacja Asymptotyczna

$f(n) = O(g(n)) \equiv \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq \infty$

$f(n) = \Omega(g(n)) \equiv \limsup_{n \rightarrow \infty} \left| \frac{g(n)}{f(n)} \right| \leq \infty$

$f(n) = o(g(n)) \equiv \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0$

$f(n) = \omega(g(n)) \equiv \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = \infty$

Metoda podstawiania -
Metoda dowodu indukcyjnego

Przykład 1. Rozwiążmy równanie rekurencyjne:
 $T(n) = 4T\left(\frac{n}{2}\right) + n \quad T(1) = \Theta(1)$
Wzmocnijmy zatem założenie indukcyjne:

- 1. $T(n) \leq c_1 n^2 - c_2 n$ (zał. indukcyjne)
- 2. $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4(c_1 \frac{n^2}{2} - c_2 \frac{n}{2}) + n$
- 3. $= c_1 n^2 - 2c_2 n + n = c_1 n^2 - (2c_2 - 1)n \leq$
- 4. $\leq c_1 n^2 - c_2 n$
- 5. Weźmy $c_1 = 1, c_2 = 2$, wtedy $T(n) \leq n^2 - 2n = O(n^2)$

Przykład 2. Weźmy paskudną rekursję $T(n) = 2T(\sqrt{n}) + \log n$.
Założmy, że n jest potęgą 2 oraz oznaczmy $n = 2^m, m = \log_2 n$.
 $T(2^m) = 2T((2^m)^{\frac{1}{2}}) + m$ Oznaczmy $T(2^m) = S(m)$. Wtedy:
 $S(m) = 2S\left(\frac{m}{2}\right) + m$ (dobrze znana rekurencja - $\tilde{S}(n) = O(m \log m)$) - patrz Lecture 2. Przejdźmy z powrotem na T, n : $T(2^m) = S(m)$
 $T(2^m) = O(m \log m)$
 $T(n) = O(\log n \log \log n)$

Master Theorem

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

$$T(n) = \begin{cases} \Theta(n^d) & \text{jeśli } d > \log_b a \\ \Theta(n^d \log n) & \text{jeśli } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{jeśli } d < \log_b a \end{cases}$$

Fibonacci

Istnieje macierz, która mnożona pozwala na policzenie n -tej liczby Fibonacciego.

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

używający tego wzoru - połączony z szybkim potęgowaniem, ma złożoność $\Theta(\log n)$.

Mnożenie liczb

$(a + ib)(c + id) = ac - bd + i(bc + ad)$
 $bc + ad = (a + b)(c + d) - ac - bd$
Zobaczmy, że ac, bd są już policzone wyżej - zamiast 4 mnożeń, mamy 3 mnożenia. $x \cdot y = x_L y_L 2^n + ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R$ $T(n) = \Theta(n^{\log_2 3})$

```
multiply(x, y)
n = max {|x|, |y|}
if n == 1 return x * y
x_L, x_R = leftmost(ceil(n/2),x), rightmost(floor(n/2),x)
y_L, y_R = leftmost(ceil(n/2),y), rightmost(floor(n/2),y)
RandomSelect(A, p, q, i)
if p == q return A[p]
r = Rand_Partition(A,p,q) # Hoare
k = r - p + 1
if i == k return A[r]
else return RandomSelect(A, p, r-1, i)
else return RandomSelect(A, r+1, q, i+k-p)

return p1 << n + (p3 - p1 - p2) << ceil(n/2) + k - p
```

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0..k
  C[i] = 0
for j = 1..length[A]
  C[A[j]] = C[A[j]] + 1
for i = 1..k
  C[i] = C[i] + C[i - 1]
for j = length[A]..1
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

```
BS(A[1..n], key) // BinarySearch 0(log n)
if (key == BS[mid]) ret mid
if (key >= BS[mid]) ret BS(A[mid..n], key)
else ret BS(A[1..mid-1], key)
ret -1
```

```
LomutoPartition(A[1..n]) // O(n)
pivot = A[1], i = 1 // any
for j = 1 to n
  if A[j] <= pivot
    i = i + 1
  swap (A[i], A[j])
swap (A[i+1], A[n])
ret i + 1
```

```
InsertionSort(A[1..n])
```

```
for j = 2..n
  key = A[j]
  i=j-1
  while(i>0 && A[i]>key)
    A[i+1] = A[i]
    i = i - 1
  A[i+1] = key
```

```
MergeSort(A[1..n]) // O(nlogn)
if (n==1) return A[1]
return Merge(
  MergeSort(A[1, mid]),
  MergeSort(A[mid+1,n])
)
```

```
Merge(A[1..k], B[1..l]) // O(k+l)
i = 1, j = 1
RES = []
while (i <= k) AND (j <= l)
  if (A[i] <= B[j]) RES+=A[i], i++
  else RES+=B[j], j++
RES+=A[i..k] OR RES+=B[j..l]
return RES

RandomSelect(A, p, q, i)
if p == q return A[p]
r = Rand_Partition(A,p,q) # Hoare
k = r - p + 1
if i == k return A[r]
else return RandomSelect(A, p, r-1, i)
else return RandomSelect(A, r+1, q, i+k-p)
```

```
Select5(A, p, q, i)
divide A[p..q] into groups of 5 elements
for each group sort it and find its median
store these medians in a new array M
mms = Select5(M, 0, len(M) - 1, len(M)/2)
r = Partition A[p..q] around mms
k = r - p + 1
if (i == k) return A[r]
if (i < k) return Select5(A, p, r - 1, i)
else return Select5(A, r + 1, q, i - k)

Insert(Q, key) // O(log n)
size(Q)++, i = size(Q)
while(i>1 AND Q[parent(i)]<key)
  Q[i] = Q[arent(i)]
  i = parent(i)
Q[i]=key
```

Struktury Danych

Drzewa

Zbalansowane drzewo $h = O(\log_2 n)$

Drzewo statystyk pozycyjnych - RBT z rozmiarem poddrzewa.

```
OS_Select(x, i) // O(log n), wykonaj x=root
r = size(x.left) + 1
if i == r return x
if i < r return OS_Select(x.left, i)
else return OS_Select(x.right, i - r)

Increase/DecreaseKey(A, i, newK)
if A[i] > newK
  swap (A[i], A[i+1])
  i = parent(i)
while(i<1 AND Q[parent(i)]<Q[i])
  swap (Q[parent(i)],Q[i])
  i = parent(i)
```

```
OS_Rank(x) // O(log n), statystyka wezla x
r = x.left.size + 1
y = x
while(y != root)
  if (y == y.p.right) // y jest prawym synem
    r += size(y.parent.left) + 1
  y = y.p
return r
```

```
while (y != root)
  if (y == y.p.right) // y jest prawym synem
    r += size(y.parent.left) + 1
  y = y.p
return r
```

Kopce

```
left(i) = 2i (LSH)
right(i) = 2i + 1 (LSH + 1)
parent(i) = i // 2 (RSH)
size(A) = rozmiar listy
```

```
HEAPIFY(A, i) // O(h)
l = left(i), r = right(i)
if (l <= size(A) AND A[l] > A[i])
  largest = l
else largest = i
if (r <= size(A) AND A[r] > A[largest])
  largest = r
if (largest != i) swap(A[i], A[largest])
HEAPIFY(A, largest)
```

PQ

```
Maximum(Q) return Q[1]
```

```
Union(Q1,Q2) // O(|Q1|+|Q2|)
BuiltHeap([Q1,Q2])
```

```
Insert(Q, key) // O(log n)
size(Q)++, i = size(Q)
while(i>1 AND Q[parent(i)]<key)
  Q[i] = Q[arent(i)]
  i = parent(i)
Q[i]=key

ExtractMax(Q) // O(log n)
max = Q[1]
Q[1] = Q[size(Q)]
size(Q)--
HEAPIFY(Q,1)
return max
```

```
Delete(Q, i) // O(log n)
Q[i] = Q[size(Q)]
size(Q)--
if (Q[i]<Q[parent(i)]) HEAPIFY(Q,i)
else
  while(i<1 AND Q[parent(i)]<Q[i])
    swap (Q[parent(i)],Q[i])
    i = parent(i)
```

```
Increase/DecreaseKey(A, i, newK)
if A[i] > newK
  swap (A[i], A[i+1])
  HEAPIFY(A,i)
else
```

```
while i>1 AND A[parent(i)] < newK
  A[i] = A[parent(i)]
  i = parent(i)
A[i] = newK
```

Grafy

Cut Property

Niech X będzie podzbiorem krawędzi minimalnego drzewa rozpinającego grafu $G = (V, E)$. Wybierzmy podzbiór wierzchołków $S \subset V$, takich, że żadna krawędź z X nie przechodzi pomiędzy wierzchołkami z S i $V \setminus S$. Niech $e \in E$ będzie krawędzią o najmniejszej wadze, która przechodzi pomiędzy S i $V \setminus S$. Wtedy $X \cup \{e\}$ należy do minimalnego drzewa rozpinającego grafu G .

Min-Cut Problem

Możemy zbudować minimalne drzewo rozpinające, w $S \setminus V - S$:

$$\Pr(A \in \text{MinCut}) \geq \frac{1}{n(n-1)} \quad (1)$$

Możemy stworzyć algorytm Kruskala (nieskierowany, więc nie musimy sortować krawędzi) do ostatniego jego kroku, w którym miałby on znaleźć ostatnią krawędź, która przechodzi pomiędzy S i $V - S$. Wtedy ta krawędź rozspójniłaby graf.

Powtórzymy $\Theta(n^2)$ razy algorytm Kruskala, aby znaleźć minimalne przecięcie - wraz z n dążącym do nieskończoności porafimy wyznaczyć najmniejszy zbiór rozcinający.

Sortowanie Topologiczne Wykonujemy DFS, zapisując czasy pre i post. Sortujemy wierzchołki według czasu post w porządku malejącym.

Mamy krawędź między składowymi silnie spójnymi (SCC), jeśli istnieje krawędź między wierzchołkami tych składowych w oryginalnym grafie.

Odwracamy kierunek wszystkich krawędzi, otrzymując graf G^T . Wykonujemy DFS na grafie G^T .

Ujście G ma największy post w G^T . DFS na G zaczyna się od ujścia G .

Komponenty:

Silnie Spójne Składowe — w obrębie jednej komponenty można dojść do każdego węzła.

Źródło - Wierzchołek grafu skierowanego, nie będący końcem żadnej krawędzi.
Ujście - Wierzchołek grafu skierowanego, nie będący początkiem żadnej krawędzi.

```
EXPLORE(G,v) # G - Graph, v - start vertex
    visited(v) = true
    previsit(v)
    for each edge (v,u) in E
        if not visited(u) EXPLORE(G,u)
    postvisit(v)
```

```
DFS(G) // O(|V|+|E|)
    for each vertex v in G
        visited(v) = false
    for each vertex v in G
        if not visited(v) EXPLORE(G,v)
```

```
BFS(G, s) // O(|V|+|E|)
    for each vertex v in G
        visited(v) = false
    visited(s) = true
    enqueue(Q, s)
    while Q not empty
        v = dequeue(Q)
        for each neighbor u of v
            if not visited(u)
                visited(u) = true
                enqueue(Q, u)
```

```
Bellman-Ford(G, s) // O(|V|*|E|)
    for all v in V
        dist(v) = infinity
    prev(v) = null
    dist(s) = 0
    repeat |V|-1 times
        for all e in E
            if dist(u) + w(u,v) < dist(v):
                dist(v) = dist(u) + w(u,v)
                prev(v) = u
```

```
Dijkstra(G, s) // O((|V|+|E|) log |V|)
    for each vertex v in G
```

```
dist(v) = infinity
visited(v) = false
dist(s) = 0
Q = makePQ(vertices) // by dist
while Q not empty
    u = extract-min(Q)
    if visited(u) continue
    visited(u) = true
    for each neighbor v of u
        if dist(v) > dist(u) + w(u,v)
            dist(v) = dist(u) + w(u,v)
            decrease-key(Q, v, dist(v))
```

```
Prim(G=(V,E), (w_i)i=1,...|E|) -> MST dla grafu G
    for v in V
        cost(v) = infinity
        prev(v) = null
    cost(u) = 0
    H = MakePQ(V) // priorytetem jest cost(v)
    while H is not empty
        v = ExtractMin(H)
        for each {v,z} in E
            if cost(z) > w(v,z)
                cost(z) = w(v,z)
                prev(z) = v
                decreaseKey(H,v,cost(z))
```

```
Kruskal(G=(V,E), w) // O(|E| log |E|), O(|E| log |V|)
    for all v in V
        makeSet(v)
    x = {}
    for all {u,v} in sorted E
        if find(u) != find(v)
            x = x U {u,v}
            union(u,v)
    ret x
```

House Robber

```
rob(values: list):
    prev, curr = 0, 0
    for value in values:
```

```
prev, curr = curr,
max(curr, prev + value)
return curr
```

Edit Distance

$|W_1| = i, |W_2| = j$. Mimumum z możliwości do (i, j) :

- insert $(i, j - 1) \rightarrow (i, j)$ (+1)
- delete $(i - 1, j) \rightarrow (i, j)$ (+1)
- replace $(i - 1, j - 1) \rightarrow (i, j)$ (+1)
- keep $(i - 1, j - 1) \rightarrow (i, j)$ (+0)

```
for (size_t n = 1; n <= s1; ++n) {
    for(size_t m = 1; m <= s2; ++m) {
        dp[n][m] = std::min({
            dp[n-1][m] + 1,
            dp[n][m-1] + 1,
            dp[n-1][m-1] +
                diff(word1[n-1], word2[m-1])});
    }
}
```

0/1 Knapsack

$max(v + arr[i - 1][j - w], arr[i - 1][j])$

Coin change 1

Minimalna liczba monet potrzebna do wydania reszty

```
for(int i = 1; i <= amount; ++i) {
    for(int j = 0; j < coins.size(); ++j) {
        if (coins[j] <= i) {
            L[i] = std::min(L[i], 1 + L[i - coins[j]])
        }
    }
}
```

Coin change 2

Unbounded Knapsack. Na ile sposobów możemy wydać resztę amount.

```
def change(amount: int, coins: List[int]) -> int:
    dp = [0] * (amount + 1)
    dp[0] = 1
```

```
for c in coins:
    for a in range(c, amount + 1):
        dp[a] += dp[a-c]
```

```
return dp[amount]
```

Longest Common Subsequence

Największy wspólny podciąg. "ace" is a subsequence of "abcde".

```
int longestCommonSubsequence(string &a, string &b) {
    short m[1001][1001] = {0};
    for (auto i = 0; i < a.size(); ++i)
        for (auto j = 0; j < b.size(); ++j)
            m[i + 1][j + 1] = a[i] == b[j]
                ? m[i][j] + 1
                : max(m[i + 1][j], m[i][j + 1]);
    return m[a.size()][b.size()];
}
```

Longest Increasing Subsequence

```
def lengthOfLIS(self, nums: List[int]) -> int:
    if not nums:
        return 0
    n = len(nums)
    dp = [1] * n
    for i in range(1, n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

Perlynt

Potrzeba (a_i, p_i) ale możemy zastępować gorsze perły lepszymi.

```
REP(i, MAX)
    REP(j, MAX)
        t[i][j] = INF;
t[n - 1][n - 1] = (a[n - 1] + 10) * p[n]
FORD(i, n - 2, 0)
{
    t[i][i] = (a[i] + 10) * p[i] +
        *min_element(t[i + 1], t[i + 1]
            FOR(j, i + 1, n - 1)
                t[i][j] = a[i] * p[j] + t[i + 1][j])
}
```

Red Black Trees

78. Guibas, Sedgewick - Red Black (RB) Trees

- Własność 0 - Drzewa RB są drzewami BST - mają BST Property - po lewej stronie węzła występują wartości mniejsze, a po prawej większe
- Własność 1 - Każdy węzeł ma kolor czerwony albo czarny (to może być bit)
- Własność 2 - Korzeń oraz *liście* są czarne
- Własność 3 - Jeśli węzeł jest czerwony, to jego bezpośrednie dzieci są czarne
- Własność 4 - $\forall X$ Każda prosta ścieżka od węzła X do liści ma tyle samo czarnych węzłów. (black_height(x), inaczej bh(x)). Prosta ścieżka oznacza, że nie zawracamy, zawsze idziemy w dół.

Struktura	Build	Find	Insert/Delete	Find mM	Find pn	List_ordered
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
Sorted Array	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
BST	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
RBT	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
MinHeap	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$ (min) / $\Theta(n)$ (max)	$\Theta(n)$	$\Theta(n \log n)$

Table 1: Porównanie różnych struktur danych (złożoność w najgorszym przypadku)