

Niech  $f(n)$  oraz  $g(n)$  będą funkcjami ze zbioru  $\mathbb{N}$  w zbior  $\mathbb{R}$ . Powiemy, że  $f = O(g)$  (co oznacza "f rośnie nie szybciej niż g"), jeśli istnieje stała  $c > 0$  taka, że  $f(n) \leq cg(n)$

$$1, \log \log n, \log n, (\log n)^c, n^c < 1, n, n \log n, n^2, n^c, c^n, \log_a b = c \Leftrightarrow a^c = b, \log_a a = 1, \log_a * b = \log_a a + \log_b b, \log_b a^n = n \log_b a, \log_b a = \frac{\log_a a}{\log_a b}, \log_b(\frac{1}{a}) = -\log_b a, \log_b a = \frac{1}{\log_a b}, a^{\log_b n} = n^{\log_b a}, \log_a \frac{x}{y} = \log_a x - \log_a y, n! = o(n^n), \omega(2^n) = \log(n!) = \Theta(n \lg n), \sum_{j=i}^n j = \frac{n(n+1)}{2} - \frac{i(i-1)}{2}, \sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}, \sum_{j=k_0}^{k_1} \sum_{j=l_0}^{l_1} a_i b_j = (\sum_{i=k_0}^{k_1} a_i)(\sum_{j=l_0}^{l_1} b_j), \sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k, \sum_{k=1}^n k = \frac{1}{2}n(n+1), \sum_{i=1}^n i = \frac{n(n+1)}{2}, \sum_{k=0}^n x^k = \frac{1-x^{n+1}}{1-x}, \sum_{i=1}^n \frac{1}{i} = \Theta(\log n), \sum_{j=i}^n 1 = n - i, \sum_{i=1}^n 1 = n^2, \sum_{i=0}^n \log \frac{n}{i} = \sum_{i=0}^n \log n - \sum_{i=0}^n \log i, T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow \Theta(n \lg n), T(n) = T(\lceil n/2 \rceil) + 1 \rightarrow O(\lg n), \sum_{i=0}^n \frac{\lg n}{2^i}, T(n) = T(\lceil n/2 \rceil) + n \rightarrow O(n), \sum_{i=0}^n \frac{\lg n}{2^i}, T(n) = 2T(n/2) + n^2 \rightarrow \Theta(n^2), T(n) = T(n/2) + T(2n/3) + n \rightarrow O(n \lg n), T(n) = 9T(n/3) + n \rightarrow \Theta(n^2), T(n) = T(2n/3) + 1 \rightarrow \Theta(\lg n), T(n) = 3T(n/4) + nlgn \rightarrow \Theta(n \lg n), T(n) = T(n-1) + n \rightarrow \Theta(n^2), T(n) = T(n-1) + 1 \rightarrow \Theta(n), T(n) = 3T(n/2) + n \rightarrow O(n \lg^3 n), \sum_{i=0}^n \frac{\lg n}{2^i} \cdot \frac{n}{2^i} \cdot 3^k$$

## ROZWIAZYWANIE REKURENCJI

1\* Metoda podstawiania

$$T(n) = 4T(\frac{n}{2}) + n, \quad T(1) = \Theta(1)$$

•  $T(n) = O(n^3) \leftarrow$  zgadujemy

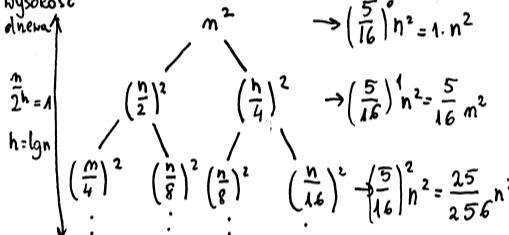
Zał. ind.:  $\forall_{k < n} T(k) \leq ck^3$

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

Kiedy  $n - c\frac{n^3}{2} \leq 0? \rightarrow c > 2, n > 2 \rightarrow$  Istnieją wartości dla których krok indukcyjny jest prawdziwy. Aczkolwiek ten przykład lepiej zrobić tak: zał. ind.:  $\forall_{k < n} T(k) \leq c_1 k^2 - c_2 k$

2\* Drzewo rekurencji

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



liczba operacji  $\rightarrow$  suma operacji w każdym wierszu

$$\sum_{k=0}^{\lg n} (\frac{5}{16})^k n^2 \rightarrow n^2 * \frac{16}{11} - \frac{5}{11} n^{0.4}$$

3\* Przez zamianę zmiennych

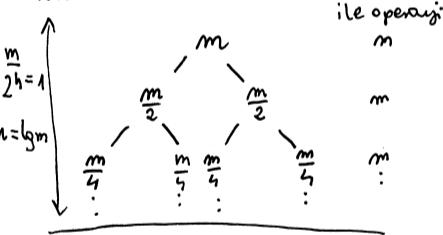
$$T(n) = 2T(\sqrt{n}) + \lg n$$

Założenie:  $n = 2^m, T(2^m) = 2T(2^{\frac{m}{2}})$

$$S(m) = T(2^m) \text{ drzewo } S(m) = 2S(\frac{m}{2}) + m \quad S(m) = m \lg m$$

$$T(n) = T(2^m) = S(m) = m \lg m = (\lg n)(\lg \lg n)$$

$$n = 2^m, m = \lg n$$



4\* Metoda iteracyjna

$$T(1) = \Theta(1); \quad T(n) = 3T(\frac{n}{4}) + n = 3(3T(\frac{n}{16}) + \frac{n}{4}) = n + \frac{3}{4}n + 3^2 T(\frac{n}{16}) = n + \frac{3}{4}n + 3^2(3T(\frac{n}{64}) + \frac{n}{16}) = n + \frac{3}{4}n + (\frac{3}{4})^2 n + 3^3 T(\frac{n}{64}) = n + \frac{3}{4}n + (\frac{3}{4})^2 n + \dots + (3)^{\log_4 n} * \Theta(1) = n \sum_{k \leq 0} (\frac{3}{4})^k = \Theta(n)$$

## MASTER THEOREM

$$T(n) = aT(\frac{n}{b}) + O(n^d), \quad a > 0, b > 1, d \geq 0$$

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

## INSERTION SORT( $a, n$ )

for  $j = 2$  to  $n$ :

- . key =  $a_j, i = j - 1$
- . while  $i > 0$  and  $a_i > key$ :
- . .  $a_{i+1} = a_i, i --$
- .  $a_{i+1} = key$

$$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(\sum_{j=2}^n j) = \Theta(n^2)$$

## MERGE SORT( $A[1..n], n$ )

if  $n == 1$ : return  $A[1]$

else:  $B = A[1, \dots, \lfloor \frac{n}{2} \rfloor]$

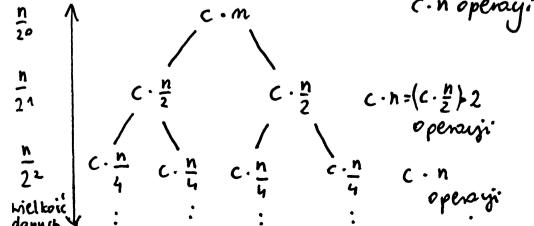
.  $C = A[\lfloor \frac{n}{2} \rfloor + 1, \dots, n] \text{ B, C} \rightarrow \Theta(n)$

. MergeSort( $B, \frac{n}{2}$ )  $\rightarrow T(\frac{n}{2})$

. MergeSort( $C, \frac{n}{2}$ )  $\rightarrow T(\frac{n}{2})$

.  $A = \text{Merge}(B, C) \rightarrow \Theta(n)$

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



Ile mamy poziomów w tym drzewie? (wysokość drzewa) poziom  $k + 1$ :  $\frac{n}{2^k} = 1 \rightarrow k = \log_2 n$

$$T(n) = (c * n) * \lg n = \Theta(n \lg n)$$

**DIVIDE & CONQUER**  $T(n) = aT(\frac{n}{b} + O(n^d))$

$a \rightarrow$  liczba podproblemów, których jest  $\frac{n}{b}$ ,  $O(n^d) \rightarrow$  scalanie podproblemów

**BINARY SEARCH(x, A, p, q)**

input: posortowana tablica  $A$ , output: znaleźć element  $x$

$$y = a \lfloor \frac{p+q}{2} \rfloor$$

if  $y == x$ : return  $\lfloor \frac{p+q}{2} \rfloor$

else if  $y < x$ : BinarySearch(x, A,  $\lceil \frac{p+q}{2} \rceil, q$ )

else: BinarySearch(x, A, p,  $\lfloor \frac{p+q}{2} \rfloor - 1$ )

$$T(n) = T(\frac{n}{2}) + O(1) \rightarrow O(\lg n)$$

**QUICK SORT(A, p, q)**

if  $p < q$ :

. pivot = Partition(A, p, q)

. QuickSort(A, p, pivot - 1)

. QuickSort(A, pivot + 1, q)

Best Case  $\rightarrow$  każdy Partition będzie zwracał pivota na środku  $\rightarrow \Theta(n \lg n)$

Worst Case  $\rightarrow$  pivot albo najmniejszym albo największym elementem tablicy (zwraca niezmienioną tablicę)  $\rightarrow T(n) = T(n-1) + \Theta(n) \rightarrow \Theta(n^2)$

Average Case  $\rightarrow \Theta(n \log n)$

**PARTITION(A, p, q)  $\rightarrow \Theta(n)$**

$x = \text{SelectPivot}(A, p, q)$

$pivotValue = A[x]$

$swap(x, q)$

$index = p$  for  $j = p$  to  $q - 1$ :

. if  $A[j] \leq pivotValue$ : swap(j, index), index ++

swap(index, q)

return index

**COUNTING SORT(A, n, k)  $\rightarrow$  sortowanie liniowe**

for  $i = 1$  to  $k$ :  $\rightarrow \Theta(k)$

.  $C[i] = 0$

for  $j = 1$  to  $n$ :  $\rightarrow \Theta(n)$

.  $C[A[j]] ++$

for  $i = 2$  to  $k$ :  $\rightarrow \Theta(k)$

.  $C[i] = C[i] + C[i-1]$

for  $j = n$  doント 1:  $\rightarrow \Theta(n)$

.  $B[C[A[j]]] = A[j]$

.  $C[A[j]] --$

Złożoność:  $\Theta(n+k)$ , jeśli  $k = \Theta(n)$  to  $\Theta(n+k) = \Theta(n)$   $k -$  rozpiętość danych, różnica między maksymalną a minimalną wartością.

**STABLE SORTING PROPERTY**  $\rightarrow$  jeśli dla równych sobie elementów po posortowaniu zachowana jest kolejność (zachodzi dla Counting Sort i Radix Sort). **RADIX SORT(a, d)**

for  $i = 0$  to  $d$ : posortuj stabilnie ciągi według  $i$ -tej pozycji b - liczba bitów w słowie, r - liczba bitów w "cyfrze" k - ilość wartości jakie może przyjmować cyfra, rozmiar alfabetu d - ilość cyfr w słowie, n - ilość danych do posortowania  $\Theta(d(n+k))$ . Jeśli sortowanie jest stabilne to  $\Theta(n+k)$ . Jeśli  $d$  jest stała, a  $k = O(n)$  to sortowanie działa w czasie liniowym.

$2^b = \text{scope}$

if  $b < \lfloor \lg n \rfloor r = b$

else:  $r = \lfloor \lg n \rfloor$

$k = 2^r, d = \lceil \frac{b}{r} \rceil$

Mamy  $n$  liczb  $b$ -bitowych i dodatkową liczbę całkowitą  $r \leq b$ . Za pomocą RS możemy posortować te liczby w czasie  $\Theta((\frac{b}{r})(n+r))$ , jeśli stosowane w niej stabilne sortowanie zajmuje czas  $\Theta(n+k)$  dla danych wejściowych należących do przedziału  $0 - k$ .

Sortowanie przez zliczania zastosowane z  $k = 2^2$ .

Minimalizacja wartości wyrażenia  $\Theta((\frac{b}{r})(n+2^r))$ :

Jeśli  $b \geq \lfloor \lg n \rfloor$  to biorąc  $r = \lfloor \lg n \rfloor$ , dostajemy najlepszy czas. Jeśli  $b < \lfloor \lg n \rfloor$ , to dla dowolnego  $r \leq b$  mamy  $(n+2^r) = \Theta(n)$ , biorąc  $r = b$  dostajemy czas  $((\frac{b}{r})(n+2^b)) = \Theta(n)$ , który jest asymptotycznie optymalny.

**STATYSTYKI POZYCYJNE**  $\rightarrow$  k-ty największy el. tablicy

**RAND SELECT (A, p, q, i):**

if  $p == q$ : return  $A[p]$

$r = \text{Rand-Partition}(A, p, q)$

//  $r - index$  pivota po zrobieniu partition całego A

$k = r - p + 1 // k - index$  pivota, ale  $A[p..q]$

if  $i == k$ : return  $A[r]$

if  $i < k$ : return RandSelect(A, p, r-1, i)

else: return RandSelect(A, r+1, q, i-k)

Worst Case  $\rightarrow O(n^2)$  Best Case  $\rightarrow O(n)$

**SELECT (A, n, k):**

j.w. tylko inny Partition:

1. podziel zbiór n elementowy na  $\lceil \frac{n}{5} \rceil$  zbiorów po 5 elementów

2. wyznacz medianę każdej grupy (np. Insertion Sortem)

3. użyj rekurencyjnie Select(), aby wyznaczyć medianę median

4. użyj mediany median jako pivota do Partition() dzielącego zbiór na  $k$  w dolnej i  $n - k$  elementów w górnej części

5. rekurencyjnie Select() na odpowiedniej części podziału, aby wyznaczyć  $i$ -ty element w dolnej lub  $(i-k)$ -ty w górnej części.

**BINARY SEARCH TREE**

for  $i = 1$  to  $n$ :

. . .  $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

return  $\max L(j)$

//  $L(i)$  - długość LIS do  $a_i$

## NAJDŁUŻSZY WSPÓŁNY PODCIĄG

= LCS-longest common subsequence  $\rightarrow O(m * n)$

znaki nie muszą być obok siebie

// wypełnienie stanów początkowych

for  $i = 0$  to  $n$ :  $C[i][0] = 0$

for  $j = 0$  to  $m$ :  $C[0][j] = 0$

for  $i = 1$  to  $n$ :

. for  $j = 1$  to  $m$ :

. . if  $A[i] == B[j]$ :  $C[i][j] = C[i-1][j-1] + 1$

. . else:  $C[i][j] = \max(C[i-1][j], C[i][j-1])$

return  $C[i][j]$

## NAJDŁUŻSZE WSPÓŁNE PODSŁOWO

= Longest common substring  $\rightarrow O(mn)$

$\rightarrow$  największe takie  $k$ , dla którego istnieją indeksy  $i, j$ , takie

że  $x_i x_{i+1} \dots x_{i+k-1} = y_j y_{j+1} \dots y_{j+k-1}$  w rozwiąż. S, T-słowa

$LCSubst(S_{1..p}, T_{1..q}) = \{LCSubst(S_{1..p-1}, T_{1..q-1}) + 1 \text{ if } S_p = T_q$

$\rightarrow$  Znaleźć najdłuższy wspólny sufiks dla wszystkich par pre-

fiksów słów. z- trzyma najdl. wspólne podsłowo znalezione

do tej pory, ret - trzyma zestaw stringów długości  $z$ , i-ostatni

znak(w LCS rozmiaru)

$LCSubst(S, T) = \max_{1 \leq j \leq n, 1 \leq i \leq n} LCSubst(S_{1..i}, T_{1..j})$

$LFCSubst(S[1..m], T[1..n]): L = A[1..m][1..n], z = 0, ret = \{\}$

{ for  $i = 1$  to  $m$ : for  $j = 1$  to  $n$ :

. if  $S[i] == S[j]$ :

. . if  $i == 1$  or  $j == 1$ :  $L[i, j] = 1$

. . else:  $L[i, j] = L[i-1, j-1] + 1$

. . if  $L[i, j] > z$ :  $z = L[i, j], ret = S[i-z+1..i]$

. . else if  $L[i, j] = z$ :  $ret.addS[i-z+1..i]$

## EDIT DISTANCE $\rightarrow O(m * n)$

for  $i = 0$  to  $m$ :  $E[i][0] = i$

for  $j = 0$  to  $n$ :  $E[0][j] = j$

for  $i = 1$  to  $m$ :

. for  $j = 1$  to  $n$ :

. .  $E[i][j] = \min(E[i-1][j] + 1, E[i][j-1] + 1, E[i-1][j-1] + \mathbb{I}_{x[i]=y[j]})$

$\mathbb{I} = 0 : x[i] = y[j], 1 : x[i] \neq y[j]$

## KOPIEC BINARNY

Drzewo pełne z wyłączeniem ostatniego wiersza  $\rightarrow h$ :  $\Theta(\log n)$

$parent(i) = \lfloor \frac{i}{2} \rfloor$ ,  $left(i) = 2i$ ,  $right(i) = 2i + 1$

HEAPIFY( $A, i$ )  $\rightarrow \Theta(\log n)$  naprawianie struktury kopca

BUILD HEAP( $A$ )  $\rightarrow O(n)$

for  $i = \lfloor \frac{\text{size}(A)}{2} \rfloor$  doント 1: Heapify( $A, i$ )

HEAP SORT( $A, n$ )  $\rightarrow O(n \log n)$

Build-Heap( $A$ )

for  $i = 1$  doント 2:

. swap( $A[1], A[i]$ )

. heap.size --

. Heapify( $A, i$ )

## KOLEJKA PRIORYTETOWA Q

INSERT( $Q, x$ )  $\rightarrow O(\log n)$

$Q.size ++$

$i = Q.size$

while  $i > 1$  and  $Q[Parent[i]] < x$ :

.  $Q[i] = Q[Parent[i]]$

.  $i = Parent[i]$

$Q[i] = x$

MAXIMUM( $Q$ )  $\rightarrow O(\log n)$

if  $Q.size = 0$ : return "error"

Extract-Max( $Q$ )

if  $Q.size = 0$ : return "emptyqueue"

else:

.  $max = Q[1]$

.  $Q[1] = Q[Q.size]$

.  $Q.size --$

. Heapify( $Q, 1$ )

. return  $max$

DECREASE-KEY( $Q, x, new-key$ ):

$x.key = new-key$

Heapify( $Q, x$ )

DELETE( $Q, x$ )  $\rightarrow O(\log n)$ :

$Q[x] = Q[Q.size]$

$Q.size --$

Heapify( $Q, x$ )

## GRAFY:

EXPLORE( $G, v$ )

$visited(v) = true$

$previsit(v)$

for each  $(u, v) \in E$ :

. if not  $visited(u)$ : Explore( $u$ )

$postvisit(v)$

DFS - Depth First Search( $G$ )  $\rightarrow O(|V| + |E|)$

for all  $v \in V$ :  $visited(v) = false$

for all  $v \in V$ :

. if not  $visited(v)$ : Explore( $G, v$ )

BFS - Breadth First Search( $G, s$ )  $\rightarrow O(|V| + |E|)$

for all  $v \in E$ :  $dist(v) = \infty$

$dist(s) = 0$

$Q = [s]$

while  $Q$  is not empty:

.  $u = \text{Eject}(Q)$

. for all  $edges(u, v) \in E$ :

. . if  $dist(v) = \infty$ :

. . . Inject( $Q$ ) =  $v$

. . .  $dist(v) = dist(u) + 1$

**DIJKSTRA**( $G, s$ )  $\rightarrow O((|E| + |V|) \log |V|)$ , kopiec binarny

for all  $v \in E$ :  $dist(v) = \infty$ ,  $prev(v) = null$

$dist(s) = 0$

$H = \text{Build-Heap}(v)$  // dystanse jako priorytet

while  $H$  is not empty:

.  $u = \text{Extract-Min}(H) \rightarrow O(|V|)$

. for all  $(u, v) \in E$ :

. . if  $dist(v) > dist(u) + l(u, v)$ :

. . .  $dist(v) = dist(u) + l(u, v)$

. . .  $prev(v) = u$

. . . Decrease-Key( $H, v$ )  $\rightarrow O(|V|)$

**ALG BELLMANA-FORDA**  $\rightarrow O(|V| * |E|)$

Gdy ścieżki mogą posiadać wagę ujemną musimy użyć mniej efektywnego, lecz bardziej wszechstronnego algorytmu Bellmana-Forda. Algorytm tworzy poprawny wynik tylko wtedy, gdy graf nie zawiera ujemnego cyklu, czyli cyklu, w którym suma wag krawędzi jest ujemna.

for all  $v \in V$ :  $dist(v) = \infty$ ,  $prev(v) = null$

$dist(s) = 0$

repeat  $|V| - 1$  times:

. for all  $e \in E$ : Update( $e$ )

UPDATE( $((u, v) \in E)$ )

$dist(v) = \min(dist(v), dist(u) + weight(u, v))$

**MINIMALNE DRZEWO ROZPINAJĄCE - MST:**

$\rightarrow$  drzewo rozpinające T, dla którego suma wag  $\sum_{e \in T} w(e)$  jest najmniejsza z możliwych. Dla niektórych grafów można wskazać wiele drzew rozpinających spełniających tę własność.

**KRUSKAL**( $G$ ) znajduje MST  $\rightarrow O(|E| * \lg |V|)$

for all  $v \in V$ : MakeSet( $v$ )

posortuj krawędzie  $E$  wg ich wag w porządku niemalejącym  $x = []$  for all  $u, v \in E$ :

. if Find( $u$ )  $\neq$  Find( $v$ ):  $\rightarrow O(2 * |E|)$

. .  $x.add(u, v)$

. . Union( $u, v$ )  $\rightarrow O(|V| - 1)$

MAKE-SET( $x$ )  $\rightarrow O(|V|)$

$\pi(x) = x // \pi(x) \rightarrow$  wskaźnik na ojca  $x$

$r(x) = 0 // r(x) \rightarrow$  h drzewa zawieszonego w wierzchołku  $x$

FIND( $x$ )

while  $x \neq \pi(x)$ :  $x = \pi(x)$

return  $x$

UNION( $x, y$ )

$A = \text{Find}(x)$

$B = \text{Find}(y)$

if  $A == B$ : return

if  $r(A) > r(B)$ :  $\pi(B) = A$

else:  $\pi(A) = B$

. if  $r(A) == r(B)$ :  $r(B) ++$

**PRIM** znajduje MST  $\rightarrow$  złożoność jak w Dijkstrze

for all  $v \in V$ :  $cost(v) = \infty$ ,  $prev(v) = null$

$v_0$  - początkowy wierzchołek

$cost(v_0) = 0$

$H = \text{Build-Heap}(v)$

while  $H$  is not empty:

.  $v = \text{Delete-Min}(H)$

. for each  $(v, z) \in E$ :

. . if  $cost(z) > w(v, z)$ :

