

AiSD

Rafał Włodarczyk

INA 4, 2025

Contents

1	Lecture I - Sortowanie	5
1.1	Worst-case analysis	5
1.2	Average-case analysis	5
1.3	Analiza losowego sortowania	5
1.4	Insertion Sort (A, n)	5
1.4.1	Worst-case analysis - Insertion Sort (A, n)	6
1.4.2	Average-case analysis - Insertion Sort (A, n)	6
1.5	Przykład złożoności	6
2	Lecture II - Merge Sort	7
2.1	Merge sort $(A, 1, n)$	7
3	Lecture III - Narzędzia do analizy algorytmów	9
3.1	Notacja asymptotyczna	9
3.2	Notacja Big- O	9
3.3	Notacja Big- Ω	10
3.4	Notacja Big- Θ	11
3.5	Notacja small- o	11
3.6	Notacja small- ω	11
3.7	Metody rozwiązywania rekurencji	12
3.8	Rozwiązywanie rekurencji	12
3.9	Metoda podstawiania - Metoda dowodu indukcyjnego	12
4	Lecture IV - Metoda drzewa rekursji	13
4.1	Metoda drzewa rekursji	13
4.2	Metoda iteracyjna	15
4.3	Master Theorem	16
4.4	Divide and Conquer	18
4.5	Wyszukiwanie elementów w portowanej tablicy	18
4.6	Binary search	18

5	Lecture V - Divide and Conquer	18
5.1	Potęgowanie liczby	18
5.2	Wyliczenie n -tej liczby Fibonacciego	19
5.3	Mnożenie Liczb	19
5.4	Mnożenie macierzy	21
5.5	Quick Sort	21
6	Lecture VI - Quicksort	22
6.1	Lomuto Partition	22
6.2	Hoare Partition	23
6.3	Worst Case Analysis for QS	24
6.4	Best case Analysis for QS	25
6.5	Specific case analysis for QS	25
6.6	Best/Worst case analysis for QS - Intuition	26
6.7	Average case analysis for QS	26
7	Lecture VII - Quicksort - further analysis	28
7.1	Strategia Count	29
7.2	Counting Sort	29
7.3	Radix Sort	30
8	Lecture VIII	30
8.1	Poprawność Radix Sort	30
8.2	Złożoność obliczeniowa Radix Sort	30
8.3	Statystyki pozycyjne	31
8.4	RandomSelect(A,p,q,i)	31
8.5	Best Case dla RandomSelect	32
8.6	Worst Case dla RandomSelect	32
8.7	Average Case dla RandomSelect	32
8.8	Select(A,p,q,i)	33
9	Lecture IX - Select	34
9.1	Struktury Danych	36
9.2	Binary Search Tree	36
9.3	Operacje na BST	37
10	Lecture X	38
10.1	Wysokość Drzewa BST	38
10.2	BST_Sort	39
11	Lecture XI	42
11.1	Red Black Trees	42
11.2	Red Black Tree Example	42
11.3	Insert w Red Black Trees	43
12	Lecture XII	45
12.1	Wzbogacanie struktur danych	45

13 Lecture XII	45
13.1 Funkcje Hashujące	45
14 Lecture XIII	46
14.1 Programowanie Dynamiczne - Wstęp	46
14.2 Przykład programowania dynamicznego - Ciąg Fibonacciego	47
14.3 Najdłuższy rosnący podciąg	47
14.4 Problem wyznaczania reszty	47
14.5 Rozkład liczby pierwszej	48
14.6 Knapsack - Problem Plecakowy	49
14.7 Optymalne Mnożenie Macierzy	50
15 Lecture XIV	50
15.1 Programowanie Dynamiczne - Kontynuacja	50
15.2 Grafy Skierowane	50
15.3 Najkrótsze ścieżki w DAG'ach - Directed Acyclic Graph	50
15.4 Edit Distance Problem	51
16 Lecture XV	52
16.1 Kopiec binarny (Binary Heap)	52
16.2 Własność kopca (maksymalnego)	52
16.3 Kolejka Priorytetowa (PQ)	56
17 Lecture XVI	56
17.1 Kolejka priorytetowa - Priority Queue	56
17.2 Inne struktury danych	57
17.3 Grafy	57
17.4 Listy sąsiedztwa	57
17.5 Macierz sąsiedztwa	58
17.6 DFS - Depth First Search	59
17.7 Zliczanie komponentów spójnych	59
17.8 Globalny zegar	59
18 Lecture XVII	59
18.1 Drzewo przejścia w DFS	59
18.2 Sortowanie topologiczne	60
19 Lecture XVIII	61
20 Lecture XIX	61
20.1 Dowód dla Dijkstra Algorithm	61
20.2 Algorytm Bellmana-Forda	62
20.3 Algorytmy Zachłanne	62
20.4 Definicja Drzewa	62
20.5 Minimalne drzewo rozpinające, MST - Minimum Spanning Tree	62
20.6 Własności minimalnego drzewa rozpinającego	63
20.7 Cut Property	63

I welcome you on the path to insanity.

Good luck :)

1 Lecture I - Sortowanie

Definiujemy problem:

1. Input: $A = (a_1, \dots, a_n), |A| = n$
2. Output: Permutacja tablicy wyjściowej $(a'_1, a'_2, \dots, a'_n)$, takie że: $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

1.1 Worst-case analysis

$$T(n) = \max_{\text{wszystkie wejścia}} \{\text{\#operacji po wszystkich } |n|\text{-wejściach}\} \quad (1.1.1)$$

1.2 Average-case analysis

Zakładamy pewien rozkład prawdopodobieństwa na danych wyjściowych. Z reguły myślimy o rozkładzie jednostajnym. Niech T - zmienna losowa liczby operacji wykonanych przez badany algorytm.

$$\mathbf{E}(T) - \text{wartość oczekiwana } T \quad (1.2.1)$$

Później możemy badać wariancję, oraz koncentrację.

1.3 Analiza losowego sortowania

Dla poprzedniego algorytmu zobaczmy, że: $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ [czyli $f(n) \sim g(n) \equiv \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$].
To jest tragiczna złożoność.

1.4 Insertion Sort (A, n)

$(A, n) = ((a_1, a_2, \dots, a_n), n)$

```
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
}
```

Przykład: $A = (8, 2, 4, 9, 3, 6), n = 6$

- $8_i, 2_j, 4, 9, 3, 6 \quad j = 2, i = 1, key = 2$ while
- $2, 8_j, 4, 9, 3, 6$
- $2, 8_i, 4_j, 9, 3, 6 \quad j = 3, i = 2, key = 4$ while

- 2, 4, 8, 9, 3, 6
- 2, 4, 8_i, 9_j, 3, 6 $j = 4, i = 3, key = 9$ no while
- 2, 4, 8, 9_i, 3_j, 6 $j = 5, i = 4, key = 3$ while
- 2, 3, 4, 8, 9, 6
- 2, 3, 4, 8, 9_i, 6_j $j = 6, i = 5, key = 6$ while
- 2, 3, 4, 6, 8, 9

```
| <= x | > x | x | ... |
| <= x | x | > x | ... |
```

Porównujemy element ze wszystkim co jest przed nim - wszystko przed j -tym elementem będzie posortowane. Insertion sort nie swapuje par elementów w tablicy, a przenosi tam gdzie jest jego miejsce.

1.4.1 Worst-case analysis - Insertion Sort (A, n)

Odwrotnie posortowana tablica powoduje najwięcej przesunięć. Ponieważ ustaliliśmy że liczba operacji w while zależy od j , wtedy:

$$T(n) = \sum_{j=2}^n O(j-1) = \sum_{j=1}^{n-1} O(j) = O\left(\sum_{j=1}^{n-1} j\right) = \quad (1.4.1)$$

$$= O\left(\frac{1+n-1}{2} \cdot (n-1)\right) = O\left(\frac{(n-1) \cdot (n)}{2}\right) = O\left(\frac{n^2}{2}\right) = O(n^2) \quad (1.4.2)$$

1.4.2 Average-case analysis - Insertion Sort (A, n)

Policzmy dla uproszczenia, że na wejściu mamy n -elementowe permutacje, z których każda jest jednakowo prawdopodobna $p = \frac{1}{n!}$. Spróbujmy wyznaczyć \mathbf{E} , korzystając z inwersji permutacji. Wartość oczekiwana liczby inwersji w losowej permutacji wynosi:

$$\mathbf{E} \sim \frac{n^2}{4} \quad (1.4.3)$$

Pominęliśmy stałe wynikające z innych operacji niż porównywanie. W average-case będziemy około połowę szybciej niż w worst-case.

Pseudokod bez przykładu jest słaby.

1.5 Przykład złożoności

Patrzmy na wiodący czynnik.

$$13n^2 + 91n \log n + 4n + 13^{10} = O(n^2) \quad (1.5.1)$$

$$= 13n^2 + O(n \log n) \quad (1.5.2)$$

Chcielibyśmy gdzie to konieczne, zapisać *lower order terms*.

Pytanie o dzielenie liczb - istnieją algorytmy, które ze względu na arytmetyczne właściwości liczb sprawiają, że mniejsze liczby mogą dzielić się dłużej niż większe. Podczas tego kursu nie omawiamy złożoności dla takich algorytmów.

2 Lecture II - Merge Sort

2.1 Merge sort ($A, 1, n$)

Niech złożoność $T(n)$ - złożoność algorytmu.

Funkcja Merge Sort stanowi o strukturze algorytmu:

```
MERGE_SORT(A,1,n)
if |A[1...n]| == 1 return A[1...n]          | 0(1)
else
    B = MERGE_SORT(A,1,floor(n/2))          | T(floor(n/2))
    C = MERGE_SORT(A,floor(n/2)+1, n)       | T(ceil(n/2))
    return MERGE(B,C)                       | 0(n)
```

Funkcja Merge pozwala łączyć poszczególne wywołania rekurencyjne:

```
MERGE(X[1...k], Y[1...l])
if k = 0 return Y[1...l]
if l = 0 return X[1...k]
if X[1] <= Y[1]
    return X[1] o MERGE(X[2...k], Y[1...l])
else
    return Y[1] o MERGE(X[1...k], Y[2...l])
```

```
MERGE(A,B)
2 1 ---> [1] + MERGE(A,B (bez 1))
7 9
13 10
19 11
20 14

2 9 ---> [1,2] + MERGE(A (bez 2),B)
7 10
13 11
19 14
20 .

... ---> [1,2,7,9,10,11,13,14]
19 .
20 .

... ---> [1,2,7,9,10,11,13,14,19,20]
```

[10], [2], [5], [3], [7], [13], [1], [6]

[2, 10], [3, 5], [7, 13], [1, 6]

[2, 3, 5, 10], [1, 6, 7, 13]

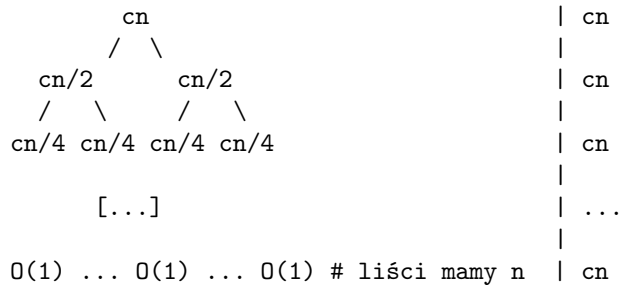
[1, 2, 3, 5, 6, 7, 10, 13]

Złożoność obliczeniowa merge-a wynosi $O(k + l)$ - w najgorszym przypadku bierzemy najpierw z jednej strony, potem z drugiej i na zmianę.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) \quad (2.1.1)$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \quad (2.1.2)$$

Rozpiszmy tzw drzewo rekursji:



Musimy dodać wszystkie koszty, które pojawiły się w drzewie. Dodajmy piętra, a następnie zsumujmy. Żeby znać wysokość drzewa interesuje nas dla jakiego h znajdzie $\frac{n}{2^h} = 1$

$$\frac{n}{2^h} = 1 \implies 2^h = n \implies h = \log_2 n \quad (2.1.3)$$

Zatem złożoność:

$$\sum_{i=1}^{\log n} cn = cn \log n \sim O(n \log n) \quad (2.1.4)$$

3 Lecture III - Narzędzia do analizy algorytmów

Dzisiejszy wykład prowadzi GODfryd

3.1 Notacja asymptotyczna

- Big- O (O -duże) $f : \mathbb{N} \rightarrow \mathbb{R}$
- Big- Ω (Ω -duże) $f : \mathbb{N} \rightarrow \mathbb{R}$
- Big- Θ (Θ -duże) $f : \mathbb{N} \rightarrow \mathbb{R}$
- Small- o (o -małe) $f : \mathbb{N} \rightarrow \mathbb{R}$

3.2 Notacja Big- O

Definition. Notacja Big- O . Funkcja $f(n) \in O(g(n))$, gdy:

$$f(n) = O(g(n)) \equiv (\exists c > 0) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (|f(n)| \leq c \cdot |g(n)|)$$

Przykład: $2n^2 = O(n^3)$, dla $n_0 = 2, c = 1$ definicja jest spełniona.

Pomijamy tutaj stałe - interesuje nas rząd wielkości

$$O(g(n)) = \{f \in \mathbb{N}^{\mathbb{R}} : f \text{ spełnia definicję}\}$$

$O(g(n))$ jest klasą funkcji, ale jako informatycy możemy zapisywać $f = O(g)$, zamiast $f \in O(g)$. Notacja nie ma symetrii, to znaczy $f = O(g) \nrightarrow g = O(f)$

Fact. Definicja Big-O za pomocą granicy. Możemy zapisać alternatywnie:

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

Uwaga. Jeśli $\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$ (istnieje), to:

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

Przykłady:

$$\begin{cases} f(n) = n^2 \\ g(n) = (-1)^n n^2 \end{cases} \quad (3.2.1)$$

Granica nie istnieje, ale $\limsup = 1$

$$\frac{f(n)}{g(n)} = \begin{cases} 1, & 2 \mid n \\ \frac{1}{n}, & 2 \nmid n \end{cases} \quad (3.2.2)$$

Granica nie istnieje.

Fact. Dokładność zapisu Big-O. Pomijamy składniki niższego rzędu jako mniej istotne, ale podkreślamy że istnieją:

$$f(n) = n^3 + O(n^2) \equiv (\exists h(n) = O(n^2)) (f(n) = n^3 + h(n)) \quad (3.2.3)$$

Rozważmy następnie stwierdzenie:

$$n^2 + O(n) = O(n^2) \equiv (\forall f(n) = O(n)) (\exists h(n) = O(n^2)) (n^2 + f(n) = h(n)) \quad (3.2.4)$$

Rozumiemy to następująco - dodając dowolną funkcję z klasy funkcji liniowych do n^2 otrzymamy funkcję z klasy funkcji kwadratowych.

3.3 Notacja Big-Ω

Definition. Notacja Big-Ω. Funkcja $f(n) \in \Omega(g(n))$, gdy:

$$f(n) = \Omega(g(n)) \equiv (\exists c > 0) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (|f(n)| \geq c \cdot |g(n)|) \quad (3.3.1)$$

biorąc $c' = \frac{1}{c} > 0$ mamy: $(|g(n)| \leq c' \cdot |f(n)|)$, czyli $g(n) = O(f(n))$.

Przykład:

$$2n^2 = O(n^3) \quad (3.3.2)$$

$$n^3 = \Omega(2n^2) \quad (3.3.3)$$

$$n = \Omega(\log n) \quad (3.3.4)$$

Każda funkcja jest Omega od siebie samej.

3.4 Notacja Big- Θ

Definition. Notacja Big- Θ . Funkcja $f(n) \in \Theta(g(n))$, gdy:

$$f(n) = \Theta(g(n)) \equiv (\exists c_1, c_2 > 0) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|) \quad (3.4.1)$$

Przykład:

$$n^2 = \Theta(2n^2) \quad (3.4.2)$$

$$n^3 = \Theta(n^3) \quad (3.4.3)$$

$$n^4 + 3n^2 + \log n = \Theta(n^4) \quad (3.4.4)$$

Fact. Dokładność zapisu Theta.

$$f(n) = \Theta(g(n)) \equiv f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)) \quad (3.4.5)$$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)) \quad (3.4.6)$$

Rozważmy przypadek patologiczny

$$f(n) = n^{1+\sin \frac{\pi \cdot n}{2}} \quad g(n) = n \quad (3.4.7)$$

$$f \neq O(g), g \neq O(f) \quad (3.4.8)$$

3.5 Notacja small- o

Definition. Notacja small- o . Funkcja $f(n) \in o(g(n))$, gdy:

$$f(n) = o(g(n)) \equiv (\forall c > 0) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (|f(n)| < c \cdot |g(n)|) \quad (3.5.1)$$

Równoważnie:

$$f(n) = o(g(n)) \equiv \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0 \quad (3.5.2)$$

Przykład:

$$n = o(n^2) \quad (3.5.3)$$

$$n^2 = o(n^3) \quad (3.5.4)$$

$$n^3 = o(2^n) \quad (3.5.5)$$

3.6 Notacja small- ω

Definition. Notacja small- ω . Funkcja $f(n) \in \omega(g(n))$, gdy:

$$f(n) = \omega(g(n)) \equiv (\forall c > 0) (\exists n_0 \in \mathbb{N}) (\forall n \geq n_0) (|f(n)| > c \cdot |g(n)|) \quad (3.6.1)$$

Równoważnie:

$$f(n) = \omega(g(n)) \equiv \lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = \infty \quad (3.6.2)$$

Przykład:

$$3.14n^2 + n = O(n^3) = \omega(n) \quad (3.6.3)$$

3.7 Metody rozwiązywania rekurencji

- Metoda podstawienia (indukcji) - Cormen
- Metoda drzewa rekursji
- Metoda master theorem

3.8 Rozwiązywanie rekurencji

1. Zgadnij odpowiedź (wiodący składnik)
2. Sprawdź przez indukcję, czy dobrze zgadliśmy
3. Wylicz stałe

Information. Historyjka. Dwóch przyjaciół zgubiło się podczas podróży balonem.

- "Gdzie jesteśmy?"
- "W balonie."

Osoba, którą spotkali, była matematykiem.

Odpowiedź była precyzyjna, dokładna i całkowicie bezużyteczna.

3.9 Metoda podstawiania - Metoda dowodu indukcyjnego

Przykład 1. Rozwiążmy równanie rekurencyjne:

$$T(n) = 4T\left(\frac{n}{2}\right) + n \quad T(1) = \Theta(1) \quad (3.9.1)$$

Założmy, że $T(n) = O(n^3)$ - pokazać, że $T(n) \leq c \cdot n^3$ dla dużych n .

1. Krok początkowy $T(1) = \Theta(1) \leq c \cdot 1^3 = c$ ok.
2. Założmy, że $\forall_{k < n} T(k) \leq c \cdot k^3$ (zał. indukcyjne, nie $\Theta(k^3)$ - chcemy konkretną stałą c)
3. $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4c\left(\frac{n}{2}\right)^3 + n = \frac{1}{2}cn^3 + n = cn^3 - \frac{1}{2}cn^3 + n \leq cn^3$.
4. Wystarczy wskazać c , takie że $\frac{1}{2}cn^3 - n \geq 0$, np $c \geq 2$
5. Pokazaliśmy, że $T(n) = O(n^3)$

Założmy, że $T(n) = O(n^2)$ - pokazać, że $T(n) \leq c \cdot n^2$ dla dużych n .

1. Krok początkowy $T(1) = \Theta(1) \leq c \cdot 1^2 = c$ ok.
2. Założmy, że $\forall_{k < n} T(k) \leq c \cdot k^2$ (zał. indukcyjne)
3. $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4c\left(\frac{n}{2}\right)^2 + n = cn^2 + n = cn^2 - cn^2 + n \leq cn^2$.
4. Tego się nie da pokazać - nie jest prawdą, że $T(n) = O(n^2)$

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 $T(n) = 2T(\sqrt{n}) + \log n$.
 Załóżmy, że n jest potęgą 2 oraz oznaczmy $n = 2^m, m = \log_2 n$.

$$T(2^m) = 2T((2^m)^{\frac{1}{2}}) + m \quad (3.9.2)$$

Oznaczmy $T(2^m) = S(m)$. Wtedy:

$$S(m) = 2S\left(\frac{m}{2}\right) + m \quad (3.9.3)$$

(dobrze znana rekurencja - $S(n) = O(m \log m)$) - patrz Lecture 2. Przejdźmy z powrotem na T, n :

$$T(2^m) = S(m) \quad (3.9.4)$$

$$T(2^m) = O(m \log m) \quad (3.9.5)$$

$$T(n) = O(\log n \log \log n) \quad (3.9.6)$$

Formalnie pokazaliśmy to tylko dla potęg 2 - musielibyśmy jeszcze indukcyjnie to udowodnić.

Kiedy podłogi i sufity mają znaczenie?

4 Lecture IV - Metoda drzewa rekursji

4.1 Metoda drzewa rekursji

W danym węźle wstawiamy koszt operacji. Sumujemy koszty węzłów na danym poziomie.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n^2, \quad T(1) = \Theta(1) \quad (4.1.1)$$

Chcemy sumować koszty na danym poziomie, a potem napisać pełną sumę.

$$\begin{array}{rcl}
 & n^2 & | \quad n^2 \\
 & / \quad \backslash & \\
 (n/2)^2 & & (n/4)^2 \quad | \quad 5/16 \quad n^2 \\
 / \quad \backslash & & / \quad \backslash \\
 (n/4)^2 \quad (n/8)^2 & (n/8)^2 \quad (n/16)^2 & | \quad 25/256 \quad n^2 = (5/16)^k \quad n^2 \\
 \dots & &
 \end{array}$$

$$T^*(n) = \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k n^2 = \quad (4.1.2)$$

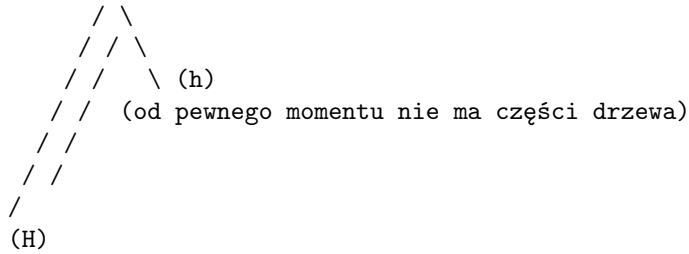
$$= n^2 \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k = \quad (4.1.3)$$

$$= n^2 \cdot \left(\frac{1}{1 - \frac{5}{16}}\right) = \quad (4.1.4)$$

$$= \frac{16}{11} n^2 \quad (4.1.5)$$

Nie mogłoby być mniej niż n^2 , bo już w pierwszym rzędzie jest n^2 .
Nie jest to dokładne, ale dostaliśmy górne ograniczenie.

$$T(n) = O(n^2) \quad (4.1.6)$$



Wysokości różnią się o stałą:

$$\frac{n}{2^H} = 1 \implies H = \log_2 n \quad (4.1.7)$$

$$\frac{n}{4^h} = 1 \implies h = \log_4 n \quad (4.1.8)$$

Za chwilę będę dodawał rzeczy, które nie istnieją

Pamiętajmy, że:

$$a^{\log_b n} = n^{\log_b a}$$

$$\hat{T}(n) = \sum_{k=0}^{H=\log_2(n)} \left(\frac{5}{16}\right)^k n^2 = \quad (4.1.9)$$

$$= n^2 \sum_{k=0}^H \left(\frac{5}{16}\right)^k = \quad (4.1.10)$$

$$= n^2 \cdot \frac{1}{11} \left(16 - 5 \left(\frac{5}{16}\right)^{\log_2 n} \right) = \quad (4.1.11)$$

$$= \frac{16}{11} n^2 - \frac{5}{11} n^{2-1.67} \quad (4.1.12)$$

Rozważmy ograniczenie dolne:

$$\check{T}(n) = \sum_{k=0}^{h=\log_4(n)} \left(\frac{5}{16}\right)^k n^2 = n^2 \frac{1}{11} \left(16 - C \cdot \left(\frac{5}{16}\right)^{\log_4 n} \right) \quad (4.1.13)$$

Zatem wiemy, że:

$$T(n) = O(\hat{T}(n)) = O(T^*(n)) \quad (4.1.14)$$

$$T(n) = \Omega(\check{T}(n)) \quad (4.1.15)$$

$$T(n) = \Theta(n^2) = \frac{16}{11} n^2 + o(n^2) \quad (4.1.16)$$

4.2 Metoda iteracyjna

$$T(n) = 3T\left(\frac{n}{4}\right) + n = \quad (4.2.1)$$

$$T(n) = 3 \left(3T\left(\frac{n}{16}\right) + \frac{n}{4} \right) + n = 9T\left(\frac{n}{16}\right) + \frac{3}{4}n + n = \quad (4.2.2)$$

$$T(n) = n + \frac{3}{4}n + 9 \left(3T\left(\frac{n}{64}\right) + \frac{n}{16} \right) = \quad (4.2.3)$$

$$T(n) = n + \frac{3}{4}n + \frac{9}{16}n + 27T\left(\frac{n}{64}\right) = \quad (4.2.4)$$

$$T(n) = n + \frac{3}{4}n + \left(\frac{3}{4}\right)^2 n + \left(\frac{3}{4}\right)^3 n + \dots + 3^j T\left(\frac{n}{4^j}\right) = \quad (4.2.5)$$

$$(4.2.6)$$

Wyznaczmy koniec iteracji:

$$\frac{n}{4^j} = 1 \implies j = \log_4 n \quad (4.2.7)$$

To jest nic innego jak:

$$\sum_{j=0}^{\log_4 n} \left(\frac{3}{4}\right)^j = O(n) \quad (4.2.8)$$

4.3 Master Theorem

Theorem. Master Theorem. Jeśli $T(n) = a \cdot T(\lceil \frac{n}{b} \rceil) + \Theta(n^d)$ dla pewnych stałych $a > 0, b > 1, d > 0$, oraz $T(1) = \Theta(1)$ to:

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

$$\hat{T}(n) = a \cdot \hat{T}\left(\frac{n}{b} + 1\right) + \Theta(n^d) \quad (4.3.1)$$

$$\check{T}(n) = a \cdot \check{T}\left(\frac{n}{b}\right) \quad (4.3.2)$$

Dowód

wielkość	.	liczba podproblemów
n	$c \cdot n^d$	1
n/b	$c \cdot (n/b)^d$	a
n/b^2	$c \cdot (n/(b^2))^d$	a^2

...

koszt na poziomie ' k ' = $c \cdot (n/b^k)^d$

liczba podproblemów na poziomie ' k ' = a^k

suma kosztów ' k '-tym wierszu = $c \cdot (a/b^d)^k \cdot n^d$

Wysokość drzewa rekursji

$$\frac{n}{b^h} = 1 \implies h = \log_b n \quad (4.3.3)$$

Zatem:

$$T(n) = \Theta\left(\sum_{k=0}^{\log_b n} \cdot \left(\frac{a}{b^d}\right)^k n^d\right) \quad (4.3.4)$$

Mogę wziąć θ zamiast o , bo dość dokładnie robię - ale trochę nie

$$\sum_{k=0}^h q^k = \frac{1 - q^{h+1}}{1 - q} \quad \sum_{h=0}^h 1^k = (h+1)$$

$$T(n) = \Theta\left(n^d \sum_{k=0}^{\log_b n} \cdot \left(\frac{a}{b^d}\right)^k\right) \quad (4.3.5)$$

(1) Jeśli $\frac{a}{b^d} < 1$, to:

$$a < b^d \quad (4.3.6)$$

$$\log_b(a) < d \quad \text{zatem} \quad (4.3.7)$$

$$T(n) = \Theta(n^d) \quad (4.3.8)$$

(większość pracy dzieje się z korzenia - okolic korzenia)

(2) Jeśli $\frac{a}{b^d} = 1$, to:

$$a = b^d \quad (4.3.9)$$

$$\log_b(a) = d \quad (4.3.10)$$

$$T(n) = \Theta(n^d \log n) \quad (4.3.11)$$

(suma kosztów w k -tym wierszu - każdy wiersz kontrybuuje równie mocno)

(3) Jeśli $\frac{a}{b^d} > 1$, to:

$$a > b^d \quad (4.3.12)$$

$$\log_b(a) > d \quad (4.3.13)$$

$$T(n) = \Theta(n^{\log_b a}) \quad (4.3.14)$$

(z każdym kolejnym poziomem koszt rośnie - większość złożoności kryje się na dole drzewa rekursji)

Z tego co dzieje się na początku... albo na końcu, bo to może być scalanie Stworzyliście za dużo podproblemów.

Co jeśli rekurencja nie ma n^d , a ma $n \log(n)$? - możemy przybliżyć

Przykład

$$T(n) = 4T\left(\frac{n}{2}\right) + 11n \quad a = 4, b = 2, d = 1 \quad (4.3.15)$$

$$\log_b a = \log_2 4 = 2 > 1 = d \quad \text{to jest przypadek (3)} \quad (4.3.16)$$

$$T(n) = \Theta(n^{\log_a b}) = \Theta(n^{\log_2 4}) = \Theta(n^2) \quad (4.3.17)$$

Przykład

$$T(n) = 4T\left(\frac{n}{3}\right) + 3n^2 \quad a = 4, b = 3, d = 2 \quad (4.3.18)$$

$$\log_b a = \log_3 4 < 2 = d \quad \text{to jest przypadek (1)} \quad (4.3.19)$$

$$T(n) = \Theta(n^d) = \Theta(n^2) \quad (4.3.20)$$

Przykład

$$T(n) = 27T\left(\frac{n}{3}\right) + 0.(3)n^3 \quad a = 27, b = 3, d = 3 \quad (4.3.21)$$

$$\log_b a = \log_3 27 = 3 = d \quad \text{to jest przypadek (2)} \quad (4.3.22)$$

$$T(n) = \Theta(n^d \log n) = \Theta(n^3 \log n) \quad (4.3.23)$$

4.4 Divide and Conquer

1. Podział problemu na mniejsze podproblemy.
2. Rozwiąż rekurencyjnie mniejsze (rozłączne) podproblemy.
3. Połącz rozwiązania problemów w celu rozwiązania problemu wejściowego.

4.5 Wyszukiwanie elementów w portowanej tablicy

- Input - posortowana tablica $A[1..n]$, element x
- Output - indeks i taki, że $A[i] = x$ lub błąd, gdy x nie występuje w A

4.6 Binary search

1. if $n = 1, A[n] = x$ return n , else A does not contain x
2. porównujemy x z $A[\frac{n}{2}]$
3. jeśli $x = A[\frac{n}{2}]$ return $\frac{n}{2}$
4. jeśli $x < A[\frac{n}{2}]$, BinarySearch($A[1..\frac{n}{2} - 1], x$)
5. jeśli $x > A[\frac{n}{2}]$, BinarySearch($A[\frac{n}{2} + 1..n], x$)

Wy nie patrzcie na pseudokody na tablicy, tylko w książce

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + \Theta(1) \quad (4.6.1)$$

$$T(n) = \Theta(\log n) \quad (4.6.2)$$

5 Lecture V - Divide and Conquer

5.1 Potęgowanie liczby

- Input - liczba x , liczba całkowita n

- Output - x^n

Bazowo zachodzi $n - 1$ mnożeń x przez siebie. (czyli $\Theta(n)$ operacji)

$$x \cdot x \cdot \dots \cdot x = x^n \quad (5.1.1)$$

Zróbmy to sprytniej:

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{dla parzystego } n \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{dla nieparzystego } n \end{cases} \quad (5.1.2)$$

Z liniowej liczby mnożeń zeszlismy do logarytmicznej liczby mnożeń.

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + \Theta(1) \quad (5.1.3)$$

$$T(n) = \Theta(\log n) \quad (5.1.4)$$

5.2 Wylczenie n -tej liczby Fibonacciego

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases} \quad (5.2.1)$$

Normalne wywołanie funkcji to $\Theta(\varphi^n)$

Wykorzystajmy podejście bottom-up, liczymy i zapamiętujemy każdorazowo F_2, F_3, \dots, F_n
Osiągnęliśmy złożoność liniową $\Theta(n)$

Istnieje jednak zwarty wzór na $F(n) = \frac{1}{\sqrt{5}} \left(\frac{\varphi^n + \varphi^n}{2} \right)$ a to możemy policzyć logarytmicznie.

Tu pojawiają się liczby - jak one się nazywały - (z sali) niewymierne.

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

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \quad (5.2.2)$$

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

5.3 Mnożenie Liczb

- Input: x, y (liczby n -bitowe)
- Output: $x \cdot y$

Standardowe mnożenie w słupku to $\Theta(n^2)$ mnożeń i $\Theta(n)$ dodawań.
Założmy, że n jest parzyste:

$$x = x_L \cdot 2^{\frac{n}{2}} + x_R \quad (5.3.1)$$

$$y = y_L \cdot 2^{\frac{n}{2}} + y_R \quad (5.3.2)$$

$$x \cdot y = (x_L \cdot 2^{\frac{n}{2}} + x_R) \cdot (y_L \cdot 2^{\frac{n}{2}} + y_R) = \quad (5.3.3)$$

$$= x_L \cdot y_L \cdot 2^n + (x_L y_R + x_R y_L) \cdot 2^{\frac{n}{2}} + x_R y_R \quad (5.3.4)$$

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) \quad (5.3.5)$$

$$a = 4, b = 2, d = 1 \quad (5.3.6)$$

$$\log_b a = \log_2 4 = 2 > 1 = d \quad (5.3.7)$$

$$T(n) = \Theta(n^2) \quad (5.3.8)$$

Asymptotycznie nie zysaliśmy nic.

Ten przypadek pokazuje, że czasami nie wystarczy bezmyślnie podzielić a potem scałić.

A co o tym myślał Gauss - tu jest dużo mnożeń - cztery.

$$(a + ib)(c + id) = ac - bd + i(bc + ad) \quad (5.3.9)$$

$$bc + ad = (a + b)(c + d) - ac - bd \quad (5.3.10)$$

Zobaczmy, że ac, bd są już policzone wyżej - zamiast 4 mnożeń, mamy 3 mnożenia.

$$x \cdot y = x_L y_L 2^n + ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) + x_R y_R \quad (5.3.11)$$

Wykonujemy i zapamiętujemy mnożenia $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$ - zamiast 4 mnożeń, mamy 3 mnożenia.

$\Theta(n)$ - wynika z przeunięć bitowych oraz dodawań.

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \quad (5.3.12)$$

$$a = 3, b = 2, d = 1 \quad (5.3.13)$$

$$\log_b a = \log_2 3 > 1 = d \quad (5.3.14)$$

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.59}) \quad (5.3.15)$$

Najszybszy znany algorytm - na podstawie szybkiej transformaty fouriera $\sim O(n \cdot \log n \cdot \log \log n)$

```
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)

  p1 = multiply(x_L, y_L)
```

```

p2 = multiply(x_R, y_R)
p3 = multiply(x_L + x_R, y_L + y_R)

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

```

Podobnie możemy mnożyć macierze.

5.4 Mnożenie macierzy

- Input: A, B - n -wymiarowe macierze
- Output: $A \cdot B$

Naiwne mnożenie macierzy wykonuje $\Theta(n^3)$ mnożeń.

Podzielmy macierz na 4 równe części:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} \quad (5.4.1)$$

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) \quad (5.4.2)$$

$$T(n) = O(n^3) \quad (5.4.3)$$

Znowu nic nie zyskał. Jesteśmy w stanie wyeliminować jedno mnożenie - osiągając ostatecznie $\Theta(n^{\log_2 7}) \sim \Theta(n^{2.81})$.

Algorytm state of the art - $\Theta(n^2 \text{polylog}(n))$.

5.5 Quick Sort

Algorytm na podział - scalanie już posortowanych. Pozwala na sortowanie w miejscu.

```

A[1..n]
|         |-----|         |
1         p         q         n

```

```

A[1..n]
|         |   <=   |         |   <   |         |
1         p         |pivot|         q         n

```

1. Podziel $A[p..q]$ na dwie tablice: $A[p..k-1]$, $pivot$, $A[k+1..q]$ takie, że:

$$\forall_{i \in [p..k-1]} A[i] \leq pivot, \forall_{j \in [k+1..q]} A[j] > pivot$$

2. Quicksort($A, p, k-1$)
Quicksort($A, k+1, q$)

Przykład - weźmy nieposortowaną tablicę:

```

Quicksort(A,1,n)
[6, 1, 4, 3, 5, 7, 2, 8] # pivot = 6
->
[1, 4, 3, 5, 2, 6, 7, 8]

.
Quicksort(A,1,5)
Quicksort(A,7,8) ->
[1, 4, 3, 2, 5, 6, 7, 8] # pivot = 1
.      .      .

Quicksort(A,2,5) ->
[1, 3, 2, 4, 5, 6, 7, 8] # pivot = 4
.      .      .

Quicksort(A,2,3) ->
[1, 2, 3, 4, 5, 6, 7, 8] # pivot = 3
.      .      .      .      .

```

6 Lecture VI - Quicksort

Rozważmy algorytmy służące do dzielenia tablicy w Quicksorcie

6.1 Lomuto Partition

```

Lomuto Partition(A, p, q) # A[p..q]
    pivot = A[p]
    i = p
    for j = p + 1 to q
        if A[j] <= pivot # expensive |A[p..q]| = n, then (n-1) comparisons ~ Theta(n)
            i = i + 1
            swap (A[i], [j]) # expensive, but if dependent
    swap (A[i], A[p]) # pivot in between A[p..i] and A[i+1..q]
    return i

```

```

A
|*| <= pivot |i| pivot < |j| ? |
p                                     q

```

We either put the ? element in the '<= pivot' part, or '> pivot' part

```

A
| <= pivot | * | pivot < |
p                                     q

```

Example

```

6, 10, 13, 5, 8, 3, 2, 11
* i          j

```

```

swap(5,10)

6, 5, 13, 10, 8, 3, 2, 11
*   i           j

do nothing

6, 5, 13, 10, 8, 3, 2, 11
*   i           j

swap(3, 13)

6, 5, 3, 10, 8, 13, 2, 11
*   i           j

6, 5, 3, 2, 8, 13, 10, 11
*   i           j

6, 5, 3, 2, 8, 13, 10, 11
*   i           j

swap(6, 2)

2, 5, 3, 6, 8, 13, 10, 11
*   i           j

return i = 3

```

Biorąc pod uwagę, że dokonujemy $n - 1$ porównań, złożoność Lomuto Partition wynosi $\Theta(n)$.

6.2 Hoare Partition

```

Hoare Partition(A, p, q) # A[p..q]
    pivot = A[floor((p+q)/2)]
    i = p - 1
    j = q + 1
    while True
        do
            i++
            while A[i] < pivot

        do
            j--
            while A[j] > pivot

        if i >= j return j
        swap(A[i], A[j])

```

* - pivot

Example

```

6, 10, 13, 5, 8, 3, 2, 11
i p      *      q j
i      *      j      # swap(6, 2)

2, 10, 13, 5, 8, 3, 6, 11
i      *      j      # swap(10, 3)

2, 3, 13, 5, 8, 10, 6, 11
      *

2, 3, 13, 5, 8, 10, 6, 11
i j      # swap (13, 5)

2, 3, 5, 13, 8, 10, 6, 11
      *
      j i

```

```

A
| <= pivot | < pivot |
p      j      q

```

return j

W Hoare Partition tracimy pivot który może ulec przesunięciu. Porównań robimy więcej o stałą $n \pm c$, $c = 1$. Złożoność $\Theta(n)$ - zdecydowanie mniej swapów, 2-3 razy mniej niż Lomuto partition.

```

QS(A,p,q)
  if p < q
    r = Partition(A,p,q)
    QS(A,p,r-1)
    QS(A,r+1,q)

```

6.3 Worst Case Analysis for QS

Najgorzej będzie jak każdorazowo będziemy nierówno dzielić po 1-szym elemencie (odwrotnie posortowana tablica).

```

      cn
     /  \
  Theta(1) c(n-1)
       /   \
    Theta(1) c(n-2)
           ...
        /     \
    Theta(1)  Theta(1)

```


$$T(n) = T(n-1) + T(0) + \Theta(n) \quad (6.3.1)$$

$$T(n) = T(n-1) + \Theta(n) \leq \sum_{i=0}^n c(n-i) + \Theta(1) = \quad (6.3.2)$$

$$= c \sum_{i=0}^n (n-i) + \Theta(n) = \quad (6.3.3)$$

$$= c \frac{(n)(n+1)}{2} + \Theta(n) = \quad (6.3.4)$$

$$= O(n^2) \quad (6.3.5)$$

6.4 Best case Analysis for QS

Najlepiej będzie jak dzielimy na pół.

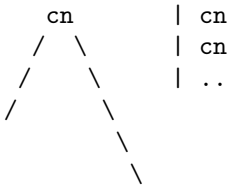
$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) \quad (6.4.1)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad (6.4.2)$$

$$T(n) = \Theta(n \log n) \quad (6.4.3)$$

6.5 Specific case analysis for QS

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) \quad (6.5.1)$$



Po zsumowaniu każde piętro będzie miało koszt cn . Zchodzimy końca wysokości drzewa.

$$\left(\frac{9}{10}\right)^h n = 1 \quad (6.5.2)$$

$$n = \left(\frac{10}{9}\right)^h \quad (6.5.3)$$

$$h = \log_{\frac{10}{9}} n \quad (6.5.4)$$

6.6 Best/Worst case analysis for QS - Intuition

$$L(n) = 2U\left(\frac{n}{2}\right) + \Theta(n) \quad (6.6.1)$$

$$U(n) = L(n-1) + L(0) + \Theta(n) \quad (6.6.2)$$

$$(6.6.3)$$

Zatem rozwiążmy układ równań:

$$L(n) = 2\left(L\left(\frac{n}{2} - 1\right) + \Theta(n)\right) + \Theta(n) \quad (6.6.4)$$

$$L(n) = 2L\left(\frac{n}{2} - 1\right) + \Theta(n) \quad (6.6.5)$$

$$L(n) = \Theta(n \log n) \quad (6.6.6)$$

6.7 Average case analysis for QS

Rozkład T_n nie jest znany do dziś.

Zapiszmy dla $0 \leq k \leq n-1$:

$$T_n = \# \text{ porównań elementów sortowanej tablicy, } |A| = n \quad (6.7.1)$$

$$X_k(n) = \begin{cases} 1 & \text{jeśli partition podzieli tablicę n-elementową na (k, n-k-1)} \\ 0 & \text{w p.p.} \end{cases} \quad (6.7.2)$$

Możemy wyznaczyć wartość oczekiwaną zmiennej losowej X_k :

$$E(X_k) = 1 \cdot P(X_k = 1) + 0 \cdot P(X_k = 0) = 1 \cdot P(X_k = 1) = \frac{(n-1)!}{n!} = \frac{1}{n} \quad (6.7.3)$$

Zapiszmy wobec tego równanie na T_n , uwzględniające wszystkie przypadki:

$$T_n =^{distr.} \begin{cases} T_0 + T_{n-1} + n - 1 & \text{if (0,n-1) - split} \\ T_1 + T_{n-2} + n - 1 & \text{if (1,n-2) - split} \\ \vdots \\ T_k + T_{n-1-k} + n - 1 & \text{if (k,n-k-1) - split} \\ T_{n-1} + T_0 + n - 1 & \text{if (n-1,0) - split} \end{cases} \quad (6.7.4)$$

$$T_n =^{distr.} \sum_{k=0}^{n-1} X_k(T_k + T_{n-k-1} + n - 1) \quad (6.7.5)$$

$$E(T_n) = E\left(\sum_{k=0}^{n-1} X_k(T_k + T_{n-k-1} + n - 1)\right) = \quad (6.7.6)$$

$$E(T_n) = \sum_{k=0}^{n-1} E(X_k(T_k + T_{n-k-1} + n - 1)) = \quad (6.7.7)$$

$$E(T_n) = \sum_{k=0}^{n-1} E(X_k) \cdot E(T_k + T_{n-k-1} + n - 1) = \quad (6.7.8)$$

$$E(T_n) = \frac{1}{n} \sum_{k=0}^{n-1} E(T_k) + E(T_{n-k-1}) + n - 1 = \quad (6.7.9)$$

$$E(T_n) = \frac{1}{n} \left(\sum_{k=0}^{n-1} E(T_k) + \sum_{k=0}^{n-1} E(T_{n-k-1}) + \sum_{k=0}^{n-1} n - 1 \right) = \quad (6.7.10)$$

$$E(T_n) = \frac{1}{n} \sum_{k=0}^{n-1} E(T_k) + \frac{1}{n} \sum_{k=0}^{n-1} E(T_{n-k-1}) + \frac{1}{n} \sum_{k=0}^{n-1} n - 1 = \quad (6.7.11)$$

$$E(T_n) = \frac{1}{n} \sum_{k=0}^{n-1} E(T_k) + \frac{1}{n} \sum_{k=0}^{n-1} E(T_{n-k-1}) + n - 1 \quad (6.7.12)$$

$$E(T_n) = \frac{2}{n} \sum_{k=0}^{n-1} E(T_k) + n - 1 \quad (6.7.13)$$

Podstawmy dla wygody $t_n = E(T_n)$:

$$t_n = \frac{2}{n} \sum_{k=0}^{n-1} t_k + n - 1 \quad \text{rekurencja z pełną historią} \quad (6.7.14)$$

Możemy usunąć historię odejmując od siebie kolejne wyrazy rekurencji.

$$nt_n = 2 \sum_{k=0}^{n-1} t_k + (n-1)n \quad (6.7.15)$$

$$(n-1)t_{n-1} = 2 \sum_{k=0}^{n-2} t_k + (n-2)(n-1) \quad (6.7.16)$$

Zachodzi odejmowanie stronami

$$nt_n - (n-1)t_{n-1} = 2 \sum_{k=0}^{n-1} t_k + (n-1)n - 2 \sum_{k=0}^{n-2} t_k - (n-2)(n-1) \quad (6.7.17)$$

$$nt_n - (n-1)t_{n-1} = 2t_{n-1} + 2(n-1) \quad (6.7.18)$$

$$nt_n = (n+1)t_{n-1} + 2(n-1) \quad (6.7.19)$$

$$\frac{t_n}{n+1} = \frac{t_{n-1}}{n} + 2 \frac{n-1}{n(n+1)} \quad (6.7.20)$$

Dokonajmy podstawienia $f_n = \frac{t_n}{n+1}$, $f_0 = 0$, $f_1 = 0$:

$$f_n = f_{n-1} + 2\frac{n-1}{n(n+1)}, f_0, f_1 = 0 \quad (6.7.21)$$

$$f_n = 2 \sum_{k=1}^n \frac{k-1}{k(k+1)} = \quad (6.7.22)$$

$$f_n = 2 \sum_{k=1}^n \frac{2}{k+1} - \frac{1}{k} = \quad (6.7.23)$$

$$f_n = 4 \sum_{k=1}^n \frac{1}{k+1} - 2 \sum_{k=1}^n \frac{1}{k} = \quad (6.7.24)$$

$$f_n = 4(H_{n+1} - 1) - 2H_n \quad (6.7.25)$$

$$f_n = 4 \left(H_n + \frac{1}{n+1} - 1 \right) - 2H_n \quad (6.7.26)$$

$$f_n = 2H_n - 4 + \frac{4}{n+1} \quad (6.7.27)$$

Wróćmy z podstawienia $t_n = (n+1)f_n$:

$$E(T_n) = t_n = (n+1)f_n = 2nH_n + 2H_n - 4(n+1) + 4 \quad (6.7.28)$$

$$H_n = \ln n + \gamma + \frac{1}{2n} + \Theta\left(\frac{1}{n^2}\right) \quad (6.7.29)$$

Widzimy, że wiodący czynnik $T_n = 2n \ln n + \Theta(n)$. Wiemy dlaczego QS jest dobry - średnio wykona $2n \ln n$ porównań asymptotycznie.

7 Lecture VII - Quicksort - further analysis

$| \leq |p| < \dots < |q| < |$

Możemy wyróżnić dwa pivoty, w obrębie których prowadzimy sortowanie. To wymaga stworzenia nowego algorytmu partition.

1. 1975 Sedgewick (liczba porównań w dual-pivot partition)

$$E(\# \text{ dual pivot partition}) \sim \frac{16}{9}n \implies E(\# \text{ QS}) \sim \frac{32}{15}n \log n$$

2. 2009 Yaroslavsky, Bentley, Block - Dual pivot quick sort

3. 2012 Sebastian Wild, Nebel

$$E(\# \text{ dual pivot partition}) \sim \frac{19}{12}n \implies E(\# \text{ QS}) \sim 1.9n \log n$$

4. 2015 Aumuller Dietzfelbinger - zaprezentowali strategię count oraz pokazali jej optymalność:

$$E(\# \text{ count partition}) \sim \frac{3}{2}n \implies E(\# \text{ QS}) \sim 1.8n \log n$$

7.1 Strategia Count

Zakładamy $p < q$ - rozpatrujemy wartość oczekiwaną, ponieważ jedynie pierwsze sprawdzenie z pivotem jest wymagane.

$$\begin{array}{ccccccc} 1 & & s_{i-1} & & & & l_{i-1} \\ | & | & \leq p & | < \dots < & | q & | < & | i | & ? & | \\ & a & & & & & & & b \end{array}$$

Rozpatrzmy i -ty element w podziale (pamiętając, że $p < q$):

- jeśli $s_{i-1} \geq l_{i-1}$ to porównujemy kolejny $A[i]$ najpierw z p , a potem ewentualnie z q (jeśli $A[i] < p$ to nie musimy porównywać z q)
- jeśli $s_{i-1} < l_{i-1}$ to $A[i]$ porównujemy najpierw z q , a potem ewentualnie z p

$$E(T_n) = E(P_n) = \frac{1}{\binom{n}{2}} \sum_{1 \leq p \leq q \leq n} E(T_{p-1}) + E(T_{q-p-1}) + E(T_q) \quad (7.1.1)$$

Tim Peters - Tim-sort - modyfikacja merge-sorta, wyznaczmy posortowane podciągi przed merge-m, mergeujemy podobnej wielkości tablice - specjalna polityka merge-owania.
... ograniczenie dolne, counting sort w czasie liniowym zbioru wielkości $O(n)$

7.2 Counting Sort

Counting sort ¹ zakłada, że każdy z wejściowych elementów mieści się w przedziale $[0, k]$, dla pewnego $k \in \mathbb{Z}$. Gdy $k = O(n)$, to złożoność algorytmu wynosi $\Theta(n + k) = \Theta(n)$. Do jego wykonania potrzebujemy tablicy pomocniczej.

```
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
```

Counting sort ma własność stabilności - zachowuje elementy tej samej wartości w kolejności, w jakiej występują w tablicy wejściowej.

¹Cormen (194-196) - Chapter 8 - Sorting in Linear Time - 8.2 Counting Sort

7.3 Radix Sort

Radix Sort polega na sortowaniu liczb w systemie pozycyjnym, przy pomocy innego stabilnego sortowania.

```
RADIX-SORT(A, d)
for i = 1..d
    COUNTING-SORT(A, i)
```

8 Lecture VIII

8.1 Poprawność Radix Sort

Indukcja po t -numer cyfry.

1. Jeśli liczby 1-cyfrowe to z poprawności Counting Sorta ok.
2. Załóżmy indukcyjnie Radix Sort jest poprawny do $t - 1$ cyfry.
3. Krok indukcyjny t -ta dwóch liczb jest taka sama. To z założenia indukcyjnego dalej oraz stable property Counting Sorta liczby do t -tej cyfry dalej pozostaną posortowane. t -ta cyfra różna: z poprawności counting sorta OK.

8.2 Złożoność obliczeniowa Radix Sort

| r -bitowy kawałek| $r'b \dots$ | $r'b \dots$ | \dots | $r'b \dots$ |
 b -bitów dzielimy na kawałki (cyfry w podstawie r)

Mamy n , b -bitowych liczb, które dzielię na (r -bitowe cyfry $\frac{b}{r}$ takich cyfr).
Cyfry są z $|\{0, \dots, 2^r - 1\}| = 2^r$. Zatem pojedynczy counting sort n -liczb względem jednej cyfry to:

$$O(n + 2^r) \quad (8.2.1)$$

Zatem Radix Sort będzie miał złożoność obliczeniową

$$O\left(\frac{b}{r}(n + 2^r)\right) \quad (8.2.2)$$

W celu ustalenia najlepszego r - minimalnego f - wykorzystamy funkcję W -Lamberta

$$f(r) = \frac{b}{r}(n + 2^r) \quad (8.2.3)$$

Zapropojujmy funkcję $r = \log n$, wtedy:

$$O\left(\frac{b}{\log n}(n + 2^{\log n})\right) = O\left(\frac{b \cdot n}{\log n}\right) = \quad (8.2.4)$$

$$(8.2.5)$$

Założmy, że zbiór sortowanych elementów to:

$$\{0, \dots, n^d - 1\} - \text{do tego zbioru należą } b\text{-bitowe sortowane liczby} \quad (8.2.6)$$

Wtedy maksymalne $b = \log n^d = d \log n$:

$$(\dots) = O\left(\frac{dn \log n}{\log n}\right) = O(d \cdot n) \quad (8.2.7)$$

8.3 Statystyki pozycyjne

Definition. Statystyka pozycyjna. k -tą statystykę pozycyjną nazywamy k -tą najmniejszą wartość z danego zbioru.

- Co się dzieje, jeśli $k = 1 \rightarrow \Theta(n)$.
- Co się dzieje, jeśli $k = n \rightarrow \Theta(n)$.
- Co się dzieje, jeśli $k = \lfloor \frac{n-1}{2} \rfloor \vee \lfloor \frac{n+1}{2} \rfloor \rightarrow$ sortowanie

8.4 RandomSelect(A,p,q,i)

Nazwa RandomSelect bierze się z tego, że wybieramy losowy element jako pivot. p to indeks początkowy, q to indeks końcowy, i to numer zadanej statystyki pozycyjnej.

```
RandomSelect(A, p, q, i)
  IF p == q return A[p]
  r = Rand_Partition(A,p,q) # jako pivota przyjmieny losowy element
  k = r - p + 1
  IF i == k return A[r]
  IF i < k return RandomSelect(A, p, r-1, i)
  ELSE return RandomSelect(A, r+1, q, i-k)
```

Przykład. Szukajmy 4-tej statystyki pozycyjnej (Pivot oznaczamy '*'):

6, 10, 13, 5, 8, 3, 2, 11
*

Po podziale względem pivota:

6, 5, 8, 3, 2, 10, 13, 11 RandomSelect(A, 1, 8, 4), r = 6, k = 6 - 1 + 1 = 6
*

Bierzemy lewą część:

2, 3, 6, 5, 8 RandomSelect(A, 1, 5, 4)
*

Pivot index: r = 2, k = 2 - 1 + 1 = 2

I dalej:

6, 5, 8 RandomSelect(A, 3, 5, 2)
*

Pivot index: r = 4, k = 4 - 3 + 1 = 2

Zwracamy czwarty element posortowanej tablicy **6** (dla sprawdzenia: posortowana tablica):

2, 3, 5, 6, 8, 10, 11, 13

8.5 Best Case dla RandomSelect

Każdorazowo dzielimy tablicę na pół.

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \quad \text{n to partition} \quad (8.5.1)$$

$$a = 1, b = 2, d = 1, \log_2 1 = 0 < 1 \implies \quad (8.5.2)$$

$$T(n) = \Theta(n) \quad (8.5.3)$$

8.6 Worst Case dla RandomSelect

Każdorazowo wybieramy pivot tak, że dzielimy tablicę na $n - 1$ i 0-elementową część.

$$T(n) = 1T(n - 1) + \Theta(n) \quad \text{partition is unfortunate} \quad (8.6.1)$$

$$T(n) = O(n^2) \quad (8.6.2)$$

8.7 Average Case dla RandomSelect

$$E(T_n) = (n - 1) + \frac{2}{n} \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} E(T_k) \quad (8.7.1)$$

Możemy zapisać (rozbicia na k i $n - k - 1$, z których bierzemy tylko jedno z nich). Wiemy, że $n - 1$ to koszt Partition, zatem:

$$T_n = \begin{cases} T_{n-1} + n - 1 : (0, n - 1) \\ T_{n-2} + n - 1 : (1, n - 2) \\ \vdots \\ T_{\lceil \frac{n}{2} \rceil} + n - 1 : (\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil) \end{cases} \quad (8.7.2)$$

Można to rozwiązać indukcyjnie, aby wykazać, że $E(T_n) = \Theta(n)$.

Uwaga. Te przekształcenia wykonałem po wykładzie

Wiemy, że ograniczenie dolne na T_n wynosi $\Omega(n)$, ponieważ $n - 1 = O(n)$ to sam koszt dla Partition. Ustalmy ograniczenie górne metodą, którą wykorzystaliśmy przy analizie Quick Sorta. Mamy:

$$X_k = \begin{cases} 1 & \text{jeśli partition podzieli tablicę n-elementową na (k, n-k-1)} \\ 0 & \text{w p.p.} \end{cases} \quad (8.7.3)$$

Zauważmy, że $k \in \{0, \dots, \frac{n}{2}\}$, zatem $E(X_k) = \frac{2}{n}$. Zapiszmy następnie:

$$T_n = \sum_{k=0}^{\frac{n}{2}} X_k (T_{n-k-1} + n - 1) \quad (8.7.4)$$

$$T_n = \frac{2}{n} \cdot \left(\sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} T_{n-k-1} + \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} (n - 1) \right) \quad (8.7.5)$$

Widzimy, że druga suma jest $O(n)$, zatem rozważmy dalej pierwszą część:

$$T_n = \frac{2}{n} \sum_{k=\frac{n}{2}-1}^{n-1} T_k + O(n) \quad (8.7.6)$$

$$(8.7.7)$$

Wystarczy pokazać, że pierwszy człon również jest $O(n)$. Zróbmy to indukcyjnie.

$$S_n = \frac{2}{n} \sum_{k=\frac{n}{2}}^{n-1} T_k \quad (8.7.8)$$

$$(8.7.9)$$

Przypadek bazowy $S_1 = T_1 = O(1)$

Założenie indukcyjne $\forall_{k < n} S_k \leq ck$. Przeprowadźmy krok indukcyjny:

$$S_n = \frac{2}{n} \sum_{k=\frac{n}{2}}^{n-1} T_k \leq \frac{2}{n} \sum_{k=\frac{n}{2}}^{n-1} ck = \quad (8.7.10)$$

$$= \frac{2c}{n} \left(\sum_{k=0}^{n-1} k - \sum_{k=0}^{\frac{n}{2}-1} k \right) = \quad (8.7.11)$$

$$= \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{n(n/2-1)}{2} \right) \leq \quad (8.7.12)$$

$$\leq \frac{2c}{n} \left(\frac{1}{8} n(3n-2) \right) = \quad (8.7.13)$$

$$= \frac{3}{4} cn \leq cn \quad (8.7.14)$$

Zatem $S_n \leq cn$ i ostatecznie $T_n = \Theta(n)$.

8.8 Select(A,p,q,i)

Algorytm ma duże podobieństwo z RandomSelect. Nie wybieramy losowego pivota - tylko inteligentnie. Niech $|A[p..q]| = n$.

1. Dzielimy $A[p..q]$ na $\lfloor \frac{n}{5} \rfloor$ pięcio elementowych części oraz ostatnią część rozmiaru ≤ 5 .
2. Sortujemy te grupy i wybieramy z każdej z nich medianę. $M = \{m_1, m_2, \dots, m_{\lfloor \frac{n}{5} \rfloor}\}$
3. Znajdujemy medianę $M : \text{Select}(M, 1, \lceil \frac{n}{5} \rceil, \lfloor \frac{\lceil \frac{n}{5} \rceil}{2} \rfloor) \implies x$. M wygląda jak osobna tablica - da się to zrobić in place.
4. Ustaw x (medianę median) jako pivot Partition(A, p, q) Dalej tak samo jak Random-Select, oczywiście odpaląc rekurencyjnie Select.

Dzielimy na 5 części

|.....|.....|.....|.....|.....|

sort 5-el części, wyzn medianę

```

max
| . | . | . | . | . |
| . | . | . | . | . |
| .m| .m| .m| .m| .m|
| . | . | . | . | . |
| . | . | . | . | . |
min

```

Zapuszczam selecta na M, |M|=5

Pierwsze dwa kroki algorytmu zajmą $O(n)$ - podzielenie tablicy i posortowanie piątek.
Późniejsze kroki są dane jako rekurencja:

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T(?) + O(n) \quad (? \text{ na następnym wykładzie}) \quad (8.8.1)$$

9 Lecture IX - Select

1. Dziel wejściową tablicę na 5-elementowe podtablice i znajdź ich mediany - $\Theta(n)$
2. Select (...) - znajdź medianę median. - $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$
3. Użyj mediany median jako pivot w Partition - $\Theta(n)$
4. Idź do lewej albo prawej podtablicy w zależności od indeksu pivot i uszkaniej statystyki pozycyjnej. $T(?)$

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T(?) + \Theta(n) \quad (9.0.1)$$

Dzielimy na 5 części

```

|.....|.....|.....|.....|.....|
sort 5-el części, wyzn medianę

```

```

max
| .w| .w| .w| . | . |
| .w| .w| .w| . | . |
| .w> w> .M> .s> .s|
| . | . | .s| .s| .s|
| . | . | .s| .s| .s|
min

```

M - mediana median (zakładamy porządek)

w - większe od mediany median (forall i : M < w_i)

s - mniejsze od mediany median (forall i : M < s_i)

". " - części o których nic nie powiemy

Wszystkich piątek jest $\text{ceil}(n/5)$

Wartości mniejszych od M jest $3 \cdot (1/2 \text{ ceil}(n/5) - 1 - 1)$ (minus skrajna oraz mediana median)

Każda piątka kontrubuuje, ale nie liczymy skrajnych piątek - ponieważ wyznaczamy ograniczenie

```
| .w| .w| .w| . | . | . |
| .w| .w| .w| . | . | . |
| .w> w> .M> .l> .l | .s|
| . | . | .s| .l| .l | .s|
| . | . | .s| .l| .l | .s|
```

l - zliczamy

s - ignorujemy (można lepiej, ale nie trzeba)

-||- większych jest $1/2 \text{ ceil}(n/5)$

$$\text{Wartości mniejszych od M} \geq \left(\frac{1}{2} \lceil \frac{n}{5} \rceil - 1 - 1 \right) \cdot 3 \geq \quad (9.0.2)$$

$$\geq \frac{3}{10}n - 6 \quad (9.0.3)$$

Prezentowana tablica

| $3/10 n - 6$ | M | $n - (3/10 m - 6) - 1 = 7/10n + 5$ |

Zatem

$$T(n) \geq T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\frac{7}{10}n + 5\right) + \Theta(n) \quad (9.0.4)$$

$$\frac{3}{4}n \geq \frac{7}{10}n + 5 \quad \text{dla} \quad n > 100 \quad (9.0.5)$$

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + \Theta(n) \quad (9.0.6)$$

Niech $T(1) = \Theta(1)$. Chcemy pokazać, że $T(n) = \Theta(n)$.

Założenie indukcyjne:

$$(\forall k < n) T(k) \leq ck \quad (9.0.7)$$

Krok indukcyjny

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + \Theta(n) \leq c \cdot \frac{n}{5} + c \cdot \frac{3}{4}n + \Theta(n) < \quad (9.0.8)$$

$$c \cdot \frac{19}{20}n + \Theta(n) < \quad (9.0.9)$$

$$cn - \frac{1}{20}cn + \Theta(n) < \quad (9.0.10)$$

$$cn - \frac{1}{20}cn + dn < \quad (9.0.11)$$

$$\text{wyznaczmy} \quad \left(-\frac{1}{20}cn + dn\right) \leq 0 \quad (9.0.12)$$

$$\left(-\frac{1}{20}c + d\right) \leq 0 \quad (9.0.13)$$

$$c \geq 20d \quad (9.0.14)$$

Zatem istnieje takie c , że nierówność jest prawdziwa, więc:

$$T(n) = O(n) \quad (9.0.15)$$

Cel analizy algorytmu - pokazać że rekurencje tego typu mogą się zdarzyć

9.1 Struktury Danych

Interesują nas struktury danych, które implementują *Set* interface.
Ma to być zbiór dynamiczny - możemy dodawać oraz usuwać elementy.
Zakładamy **comparison model**.

Podstawowe metody *Set* interface:

1. *build(A)* - buduje "set" z danych zawartych w A . Mamy $a \in A$, $a.key$ - klucz identyfikujący element.
2. *length(A)* - zwraca moc zbioru A
3. *find(k)* - zwraca element $a \in A$ taki że $a.key = k$ lub null
4. *insert(a)* - dodaj element a do zbioru A
5. *delete(k)* - usuń (czasem zwróć) element zbioru A o kluczu k
6. *find_min()*, *find_max()*, *find_prev(k)*, *find_next(k)* (*find n*), *list_ordered()* - zwróć element o najmniejszym lub największym kluczu k .

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	$\Theta(n)$	$\Theta(n)$	insert $\Theta(1)$, delete $\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
BST	$\Theta()$	$\Theta()$	$\Theta()$	$\Theta()$	$\Theta()$	$\Theta(n)$

Table 1: Porównanie różnych struktur danych

9.2 Binary Search Tree

Drzewo przeszukiwań binarnych

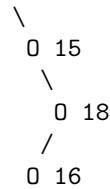
Single Tree Node

```
| parent |
| left | key | values | right |
```

"0" represents a node

```

      0 5
     /  \
    3 0   0 10
   /  \  /  \
  1 0  8 0  0 12
```



```
InorderTreeWalk(p)
1, 3, 5, 8, 10, 12, 15, 16, 18
```

Zakładamy interfejs zbioru (klucze się nie powtarzają). W przeciwnym przypadku zakładamy multizbiór.

BST Property. Niech $x \in T$, x jest węzłem drzewa T (BST), wtedy:

- każdy $y \in x.left$ ma $y.key < x.key$
- każdy $y \in x.right$ ma $y.key > x.key$

9.3 Operacje na BST

```
InorderTreeWalk (x\in T)
  if (x != null)
    InorderTreeWalk (x.left)
    print(x)
    InorderTreeWalk (x.right)
```

$$T(n) = T(k) + \Theta(1) + T(n - 1 - k) \quad (9.3.1)$$

Pokażmy, że $T(n) = \Theta(n)$

Założenie indukcyjne: $\forall k < n \quad T(k) \leq ck$ Krok indukcyjny:

$$T(n) = T(j) + \Theta(1) + T(n - 1 - j) \leq \quad (9.3.2)$$

$$cj + \Theta(1) + c(n - 1 - j) = \quad (9.3.3)$$

$$= cn - c - \Theta(1) \leq cn \quad (9.3.4)$$

Zatem $T(n) = O(n)$, musimy przejść n elementów, zatem ograniczenie dolne również wynosi n , więc $T(n) = \Theta(n)$.

```
TreeSearch(x, k)
  if x == null OR k == x.key
    return x
  if k < x.key
    return TreeSearch(x.left, k)
  else
    return TreeSearch(x.right, k)
```

```
TreeMinimum(x) -> T(n) = O(h)
```

```
TreeMaximum(x) -> T(n) = O(h)
```

```

TreeSuccessor(x)
    if x.right != null
        return TreeMinimum(x.right)
    y = x.p
    while y != null AND x == y.right
        x = y
        y = y.p
    return y

```

TreeSuccessor(x) $\rightarrow T(n) = O(h)$

10 Lecture X

TreeInsert(x, el) $\sim O(h)$ - nie było kodu na wykładzie :/

```

TreeInsert(x, el)
    if x == null
        return el
    if el.key < x.key
        x.left = TreeInsert(x.left, el)
        x.left.p = x
    else
        x.right = TreeInsert(x.right, el)
        x.right.p = x
    return x

```

TreeDelete(x)

1. x jest liściem
 - zwolnij pamięć zajmowaną przez x
 - ustaw wskaźnik jego ojca (na niego na null)
2. x ma jedno poddrzewo
 - x ma syna v to
 - zwalniamy pamięć x
 - ojciec x wskazuje na v
 - v.p wskazuje na x.p
3. x ma dwa poddrzewa
 - znajdź następnik x \rightarrow y
 - zastąp dane x danymi y
 - skasuj y

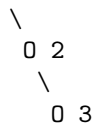
10.1 Wysokość Drzewa BST

Wysokość drzewa to liczba krawędzi wzdłuż najdłuższej ścieżki od korzenia do liścia.

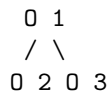
$$h = (n - 1) = O(n) \quad (10.1.1)$$

Worst Case

0 1



Best Case

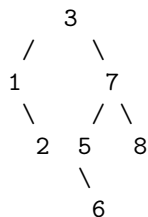


Definition. Drzewo zbalansowane. Mówimy, że drzewo jest zbalansowane jeśli jego wysokość to $O(\log n)$.

10.2 BST_Sort

Dodaj wszystkie elementy tablicy A do drzewa BST. InorderTreeWalk(T)

3 7 5 6 8 1 2



QS



Widzimy znaczące podobieństwo w porównaniach.

$$E(\text{Time}(\text{BST_SORT})) = E(\text{Time}(\text{QuickSort})) = \Theta(n \log n) \quad (10.2.1)$$

$$\text{Time}(\text{BST_SORT}) = \sum_{x \in T} \text{depth}(x) \quad (10.2.2)$$

$$E\left(\sum_{x \in T} \text{depth}(x)\right) = \Theta(n \log n) \quad (10.2.3)$$

$$E\left(\frac{1}{n} \sum_{x \in T} \text{depth}(x)\right) = \Theta(\log n) \quad (10.2.4)$$

średnia głębokość węzła w losowym drzewie BST (10.2.5)

$$h = \max_{x \in T} \{\text{depth}(x)\} \quad (10.2.6)$$

$$\frac{1}{n} \sum_{x \in T} \text{depth}(x) \leq \frac{1}{n} ((n - \sqrt{n})(\log n) + \sqrt{n} \cdot \sqrt{n}) \leq \log n + 1 = O(\log n), \text{ ale } h = O(\sqrt{n}) \quad (10.2.7)$$

Theorem. Wysokość BST. Niech T będzie losowym drzewem BST o n -węzłach, wtedy:

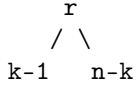
$$E(h(T)) \leq 3 \log_2 n + o(\log n) \quad (10.2.8)$$

Proof. Nierówność Jensena jeśli f -wypukła, to:

$$f(E(X)) \leq E(f(X)) \quad (10.2.9)$$

1. Nierówność Jensena
2. Zamiast analizować zmienną losową H_n , będziemy się zajmować $Y_n = 2^{H_n}$
3. Pokażemy, że $E(Y_n) = O(n^3)$
4. $2^{E(H_n)} \leq E(2^{H_n}) = E(Y_n) = O(n^3)$
5. $E(H_n) = 3 \log_2 n + o(\log n)$

Pokażmy, że $E(Y_n) = O(n^3)$.



Zakładając że korzeń tworzy $(k-1, n-k)$ -split:

$$H_n =^d = 1 + \max\{H_{k-1}, H_{n-k}\} \quad (10.2.10)$$

$$Y_n =^d = 2 \max\{Y_{k-1}, Y_{n-k}\} \quad (10.2.11)$$

$$Z_{n,k} =^d = \begin{cases} 1 & \text{jesli korzeń } n\text{-el drzewa wykonuje } (k-1, n-k)\text{-split} \\ 0 & \text{w p.p.} \end{cases} \quad (10.2.12)$$

$$E(Z_{n,k}) = 1 \cdot P((k-1, n-k)\text{-split}) = \frac{(n-1)!}{n!} = \frac{1}{n} \quad (10.2.13)$$

$$Y_n =^d = \sum_{k=1}^n Z_{n,k} \cdot 2 \max\{Y_{k-1}, Y_{n-k}\} \quad (10.2.14)$$

$$E(Y_n) = E\left(\sum_{k=1}^n Z_{n,k} \cdot 2 \max\{Y_{k-1}, Y_{n-k}\}\right) \quad (10.2.15)$$

$$E(Y_n) = 2 \sum_{k=1}^n E(Z_{n,k} \cdot \max\{Y_{k-1}, Y_{n-k}\}) \quad (10.2.16)$$

$$E(Y_n) = 2 \sum_{k=1}^n E(Z_{n,k}) \cdot E(\max\{Y_{k-1}, Y_{n-k}\}) \quad (10.2.17)$$

$$E(Y_n) = \frac{2}{n} \sum_{k=1}^n E(\max\{Y_{k-1}, Y_{n-k}\}) \quad (10.2.18)$$

$$\leq_{(\max xy \leq x+y)} \frac{2}{n} \sum_{k=1}^n E(Y_{k-1}) + E(Y_{n-k}) \quad (10.2.19)$$

$$E(H_n) = O(\log n), H_n = \log_2 Y_n \quad (10.2.20)$$

$$Y_{k-1} = 2^1 0, Y_{n-k} = 2^1 1 \quad (10.2.21)$$

$$\max 2^{10}, 2^{11} = 2^{11} \quad (10.2.22)$$

$$2^{10} + 2^{11} = 3 \cdot 2^{10} \quad (10.2.23)$$

$$= \frac{2}{n} \sum_{k=1}^n E(Y_{k-1}) + \sum_{k=1}^n E(Y_{n-k}) \quad (10.2.24)$$

$$= \frac{4}{n} \sum_{k=0}^{n-1} E(Y_k) \quad (10.2.25)$$

$$Y_n = E(Y_n) \quad (10.2.26)$$

$$y_n \leq \frac{4}{n} \sum_{k=0}^{n-1} y_k \quad (10.2.27)$$

$$ny_n \leq 4 \sum_{k=0}^{n-1} y_k \quad (10.2.28)$$

$$y_n = O(n^3) \quad (10.2.29)$$

Dowód indukcyjny. Założenie indukcyjne $y_0 = y_1 = 0, \forall k < ny_k \leq cn^3$

$$\text{krok indukcyjny} \quad y_n \leq \frac{4}{n} \sum_{k=0}^{n-1} y_k \quad (10.2.30)$$

$$\leq_{\text{ind}} \frac{4}{n} \sum_{k=0}^{n-1} ck^3 = \quad (10.2.31)$$

$$= \frac{4c}{n} \sum_{k=0}^{n-1} k^3 = \quad (10.2.32)$$

$$= \frac{4c}{n} \cdot \frac{n^2(n-1)^2}{4} = \quad (10.2.33)$$

$$= cn(n-1)^2 \leq cn^3 \quad (10.2.34)$$

Zatem:

$$E(Y_n) = O(n^3) \quad (10.2.35)$$

□

Dokładny wynik pokazany przez Devroye 1986r.

$$E(H_n) \sim 2.9882 \log_2 n \quad (10.2.36)$$

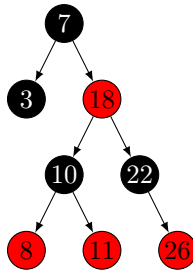
11 Lecture XI

11.1 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. ($\text{black_height}(x)$, inaczej $\text{bh}(x)$). Prosta ścieżka oznacza, że nie zawracamy, zawsze idziemy w dół.

11.2 Red Black Tree Example



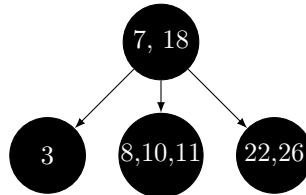
Programując - ostatni liść - *nil* nie ma klucza, kolor jest czarny, a wskaźnik na ojca - to każdy węzeł. Liście drzewa RB to wszystkie *nil*-węzły.

Przykład Czarna wysokość $\text{bh}(18) = 2$

Lemat Niech T będzie drzewem czerwono-czarnym o n -węzłach. Wtedy:

$$\text{wysokość}(T) \leq 2 \log_2(n + 1) \quad (11.2.1)$$

Dowód Czarni rodzice wchłaniają czerwone dzieci.



W drzewie binarnym liczba liści wynosi $n + 1$
(zawsze dokładamy 2 liście do każdego węzła - można to pokazać indukcyjnie)

2-3-4-Tree. Liczba liści nie zmienia się.

Mamy $n + 1$ liści w drzewie czerwono-czarnym oraz w 2-3-4-drzewie (dowód - indukcyjnie)

- Niech h - wysokość drzewa czerwono-czarnego.
- Niech h' - wysokość odpowiadającego mu 2-3-4-drzewa.

Zauważmy, że $h' = bh(\text{korzenia RB drzewa})$. Ograniczmy liczbę liści za pomocą funkcji od tej wysokości

$$2^{h'} \leq \# \text{liści} \leq 4^{h'} \quad (11.2.2)$$

Węzły binarne o wysokości h' dają $2^{h'}$ węzłów.

Węzły 2-3-4 o wysokości h' dają $4^{h'}$ węzłów.

Naszych liści jest $n + 1$, zatem:

$$2^{h'} \leq n + 1 \quad (11.2.3)$$

$$h' \leq \log_2(n + 1) \quad (11.2.4)$$

Z konstrukcji wchłaniania wiemy, że $h \leq 2h'$ (ponieważ każdy czarny węzeł może wchłaniać czerwone dzieci - z 2 razy wyższego drzewa). Zatem:

$$h \leq 2 \log_2(n + 1) \quad (11.2.5)$$

W Javie 8 HashMapy były implementowane jako drzewa czerwono-czarne.

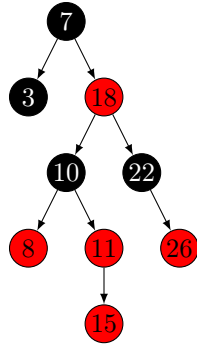
Modyfikacja drzewa czerwono-czarnego obejmuje operacje różne od BST. Drzewo będzie wtedy zmieniać swoją strukturę aby zachować czarną wysokość - stąd również nazwa - self-balancing trees. Operacje niemodyfikujące drzewa czerwono-czarnego są tożsame z operacjami na drzewach BST.

11.3 Insert w Red Black Trees

RB_Insert(T, z)

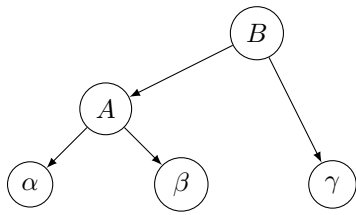
1. Wstawiamy węzeł z do drzewa T tak jak w przypadku BST
2. Ustawiamy kolor węzła z na czerwony
3. Naprawiamy drzewo T - wywołujemy funkcję RB_Fixup(T, z)

Chcemy umieścić nowy węzeł (15) w drzewie czerwono-czarnym.

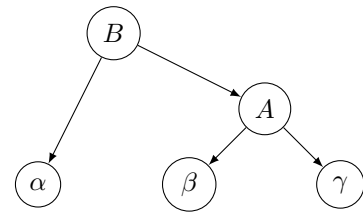


Operacje używane w procedurze Fixup

1. **recolor** - $O(1)$ - zmiana koloru węzła - z czerwonego na czarny, z czarnego na czerwony
2. **rotate** - $O(1)$ - rotacja węzła x w lewo lub w prawo.



Before Right Rotation

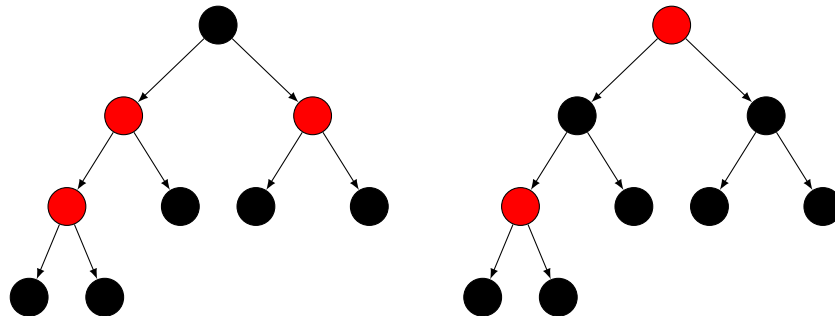


After Right Rotation

$$(\forall a \in \alpha b \in \beta c \in \gamma) (a \leq B \leq b \leq c \leq A) \quad (11.3.1)$$

RB_Fixup(T,z)

Case 1 - z jest czerwony, ojciec x , wujek $w = z.p.p \rightsquigarrow$ inne dziecko, $x = z.p$ jest czerwony oraz w -czerwony.

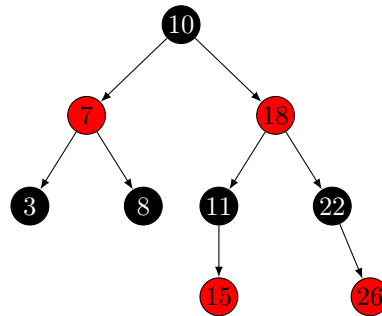


Case 2 - z - czerwony, x - czarny, w - czarny, zachodzi zig-zag

Case 3 - z - czerwony, x - czarny, w - czarny, bez zig-zag

Podałem się z rysowaniem tego w tikz

Ostatecznie



Wnioski

- Fixup - $O(\log n)$
- Insert - $O(\log n)$
- RB_Insert - $O(\log n)$

Inne drzewa od Red Black Trees to drzewa AVL (różnica stałych przy logarytmach), self-leaning left trees, skip list.

Następny wykład - kolejna struktura implementująca interfejs set

12 Lecture XI

12.1 Wzbogacanie struktur danych

TBD.

13 Lecture XII

13.1 Funkcje Hashujące

Universal Hash Property. Prawdopodobieństwo kolizji dla funkcji hashującej wynosi:

$$Pr(h_{a,b}(x) = h_{a,b}(y)) = \frac{1}{m} \quad (13.1.1)$$

$$f_{a,b}(x) = (ax + b) \mod p \quad (13.1.2)$$

$$g : \mathbb{Z}_l \rightarrow B.t.ż. \forall_{i \in B} |\{y \in \mathbb{Z}_p : g(y) = i\}| \leq \left\lceil \frac{p}{m} \right\rceil \quad (13.1.3)$$

Naturalny wybór $g(y) = y \mod m$

$$h_{a,b}(x) = g(f_{a,b}(x)) \quad (13.1.4)$$

$$\mathcal{H} = \{h_{a,b} t.ż. a, b \in \mathbb{Z}_l, \quad a \neq 0\} \quad (13.1.5)$$

Lemat. Dla $x, y \in A$, t.ż. $x \neq y$ zdefiniujemy:

$$\delta_H(x, y) = \delta_g(\mathbb{Z}_p, \mathbb{Z}_p) = \sum_{x, y \in \mathbb{Z}_p} \delta_g(x, y) \quad (13.1.6)$$

$$(13.1.7)$$

Funkcja δ - zliczająca kolizje.

$$\delta_f = \begin{cases} 1 & \text{jeśli } f(x) = f(y) \\ 0 & \text{w p.p.} \end{cases} \quad (13.1.8)$$

Dowód. Niech $r, s \in \mathbb{Z}_p$, $r \neq s$, para (r, s) odpowiada $(f_{a,b}(x), f_{a,b}(y))$, ponieważ $a \neq 0$, $x \neq y$, $f_{a,b}(x) \neq f_{a,b}(y)$. Możemy skorzystać ze znajomości algebry abstrakcyjnej:

$$ax + b = r \pmod{p} \quad ay + b = s \pmod{p} \quad (13.1.9)$$

Wiemy, że za pomocą rozszerzonego algorytmu Euklidesa możemy znaleźć unikalne a, b , takie że zadane kongruencje będą spełnione. Znajdźmy takie a, b , że nie zajdzie kolizja. Zatem:

$$(r, s) = (f_{a,b}(x), f_{a,b}(y)), \text{ to} \quad (13.1.10)$$

$$H(a, b)(x) = H(a, b)(y) \iff g(r) = g(s), \text{ stąd} \quad (13.1.11)$$

$$\delta_H(x, y) = \sum_{x, y \in \mathbb{Z}_p} \delta_g(x, y) \quad (13.1.12)$$

Lemat 2. $\forall_{x, y \in A} \delta_H(x, y) \leq \frac{|H|}{|B|} = \frac{|H|}{m}$. Dowód:

$$m_i = |\{y \in \mathbb{Z}_p : g(y) = i\}| < \left\lceil \frac{p}{m} \right\rceil \quad (13.1.13)$$

$$p, m \in \mathbb{Z} : \left\lceil \frac{p}{m} \right\rceil \leq \left(\frac{p-1}{m} \right) + 1 \quad (13.1.14)$$

Dla ustalonego $r \in \mathbb{Z}_p$ mamy co najwyżej $\frac{p-1}{m}$'s'-ów kolidujących. Możliwe $r \in \mathbb{Z}_p$ jest p stąd mamy:

$$\delta_H(x, y) \leq \frac{p(p-1)}{m} = \frac{|H|}{m} = \frac{|H|}{|B|} \quad (13.1.15)$$

Wybierając jedną z tych funkcji (1 out of m) $\leq \frac{1}{m}$

14 Lecture XIII

14.1 Programowanie Dynamiczne - Wstęp

Dzisiejszy wykład prowadzi p. Gębala. 05.05.2024

Dzielimy problem rekurencyjnie - ale nie rozwiązujemy go w ten sposób, ponieważ mniejsze podproblemy nie są rozłączne - tak jak w divide and conquer.

14.2 Przykład programowania dynamicznego - Ciąg Fibonacciego

$$F(n) = F(n-1) + F(n-2) \quad (14.2.1)$$

$$F(0) = 0 \quad (14.2.2)$$

$$F(1) = 1 \quad (14.2.3)$$

14.3 Najdłuższy rosnący podciąg

Input: $a_1, \dots, a_n \in \mathbb{N}$

Output: największe k , takie że istnieje:

- ciąg indeksów $1 \leq i_1 < i_2 < \dots < i_k \leq n$
- $a_{i_1} < a_{i_2} < \dots < a_{i_k}$

Patrzmy zakładając że znamy rozwiązanie dla $N-1$, co jeśli dodamy n -ty element.

1. $L(i)$ - długość najdłuższego podciagu w zbiorze $[1..i]$ z elementem końcowym w i
2. $L(i) = 1 + \max_{1 \leq j < i} \{L(j) : a_i > a_j\}$

Przykład

i	0	1	2	3	4	5
a _i	2	3	1	5	4	3
L(i)	1	2	1	3	3	2

```
for i=1 to n do
    L(i) = 1 + max_{1 ≤ j < i} {L(j) : a_i > a_j}
```

```
return max_{1 ≤ i ≤ n} L(i)
```

Programowanie dynamiczne zakłada zapisywanie poprzednich kroków, tu:

$$L(i) \ (\forall i \leq n) \quad (14.3.1)$$

chcąc odzyskać ciąg powinniśmy zdefiniować:

$\text{prev}(i) = j$

1. Złożoność czasowa $O(n^2)$ (for + max)
2. Złożoność pamięciowa $O(n)$ (każdy $L(i)$ $1 \leq i < n$)

14.4 Problem wyznaczania reszty

Input:

- $c_1 < c_2 < \dots < c_n$ - zbiór nominałów $\in \mathbb{N}$
- R - reszta do wydania

Output:

- minimalne k , takie że k -monet wystarczy do wydania R

Taktyka zachłanna nie działa dla np. 1,4,5,8:

- zachłanny - 5,1,1,1 - 4 monety
- optymalny - 4,4 - 2 monety

Rozwiążmy za pomocą programowania dynamicznego.

$$L(i) - \text{minimalna liczba monet do wydania reszty } i \quad (14.4.1)$$

$$L(i) = 1 + \min_{1 \leq j \leq n} \{L(i - c_j) : c_j \leq i\} \quad (14.4.2)$$

$$L(0) = 0 \quad (14.4.3)$$

c_1, c_2, c_3 = 1, 4, 5

Per i sprawdzamy każdą resztę 1, 4, 5

i	0	1	2	3	4	5	6	7	8
L(i)	0	1	2	3	1	1	2	3	2

Złożność $O(n \cdot R)$, liczymy minimum w pętli for. Prev backtrace

```

0 1 1 1 4 5 5 1 4
----->
          ----->

```

i-4 = 4 jmp to 4

Co jeśli mamy $\{2, 4, 5\} \in C$, wtedy:

```

i 0 1
L(i) 0 +infty

```

Nie da się wydać reszty 1.

Fakt 1. Jeżeli zbiór monet zawiera nominał 1, to rozwiązanie istnieje dla każdego $R \in \mathbb{N}$. Decyzja kiedy występuje największa liczba, której nie potrafimy wydać jest problemem NP-trudnym.

Zachłanny algorytm działa dla zbioru monet, które są wielokrotnościami siebie, a w szczególności gdy

$$\forall_{i,j} i < j \rightarrow 2c_1 \leq c_j \quad (14.4.4)$$

Długość danych $n \cdot \log c_n + \log_2 R = m$ - bitowe wejście, jeśli $n \log c_n \leq \log R$, wtedy:

$$O(nR) = O(n \cdot 2^{O(m)}) \quad (14.4.5)$$

R - liczba, a nie wielkość zapisu danych.

14.5 Rozkład liczby pierwszej

Input: p - liczba, długość $\log_2(p)$ (bitowa)

Output: Czynniki pierwsze rozkładu p

$$O(\sqrt{p}) \rightarrow O(\sqrt{(\sqrt{2^{\log_2(p)}})}) \quad (14.5.1)$$

14.6 Knapsack - Problem Plecakowy

Input:

- n par (waga, wartość) (w_i, v_i)
- ograniczenie górne na pojemność plecaka W .

Output:

- $I \subseteq \{1, \dots, n\}$ tż:
 1. $\sum_{i \in I} w_i \leq W$
 2. $\sum_{i \in I} v_i$ jest największa

Istotnym założeniem, które musimy podjąć jest:

$$\forall_i w_i \leq W \quad (14.6.1)$$

Ponieważ musimy ignorować pojedyncze przedmioty, które są większe od pojemności plecaka.

Niech $V(n)$ - maksymalna wartość na n przedmiotach.

$$V(n, w) = \max\{v(n-1, w), V(n-1, w-w_n) + v_n\} \quad (14.6.2)$$

Wyjaśnienie wyboru parametrów funkcji max:

- $v(n-1, w)$ - nie bierzemy n -tego przedmiotu
- $V(n-1, w-w_n) + v_n$ - bierzemy n -ty przedmiot, ale musimy zmniejszyć pojemność plecaka o wagę w_n .

Podajmy kroki początkowe w rekurencji:

$$V(0, *) = 0 \quad (14.6.3)$$

$$V(n, W) = \max\{V(n-1, w), V(n-1, w-w_n) + v_n\} \quad (14.6.4)$$

$$V(0, w) = V(j, 0) = 0 \quad (\forall j \in \{0, \dots, n\} \text{ i } w \in \{0, \dots, w\}) \quad (14.6.5)$$

```
for i <- 1 to n do
  for w <- 0 to W do
    if w_i > w then V(i,w) = V(i-1,w)
    else V(i,w) <- max(V(i-1,w), V(i-1,w-w_i) + v_i)
```

$$O(n \cdot W) \quad (14.6.6)$$

$$O(n \cdot 2^{O(m)}) \quad (14.6.7)$$

Zobaczmy, że jeśli $w \leftarrow 2^{20} \cdot w$ (dodajemy 20 zer binarnie) 2^{20} większy czas, to jest algorytm wykładniczy. Jeśli $W = O(n)$ to algorytm jest n^2 .

- Insertion sort - dynamicznie dodajemy element $n+1$ do posortowanej listy długości n

14.7 Optymalne Mnożenie Macierzy

Input: Macierze A_1, \dots, A_n , $A_i : m_{i-1} \times m_i$

Przykład $(10,2) * (2,10) * (10,3)$

- $(10*2*10) + (10*10*3) = 500$ mnożeń

- $(10*2*3) + (2*10*3) = 120$ mnożeń

$$c(i, j) - \text{ optymalny koszt przemnożenia } A_i \times \dots \times A_j \quad (14.7.1)$$

$$c(i, i) = 0 \quad (14.7.2)$$

$$c(i, j) = \min_{i \leq k < j} (c(i, k) + c(k+1, j) + m_{i-1} \cdot m_k \cdot m_j) \quad (14.7.3)$$

$$i < j \text{ ostatnie mnożenie } k \quad (14.7.4)$$

$\frac{n^2}{2}$ wartość $\cdot n = O(n^3)$

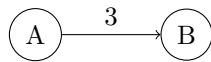
Wzór rekurencyjny, ale liczymy od dołu. Należy udowodnić poprawność rozwiązania.

15 Lecture XIV

15.1 Programowanie Dynamiczne - Kontynuacja

15.2 Grafy Skierowane

$G = (V, E)$, $|V| = n$, $|E| = m$, V - wierzchołki, E - krawędzie.



15.3 Najkrótsze ścieżki w DAG'ach - Directed Acyclic Graph

Grafy skierowane acykliczne nie posiadają cykli. Jesteśmy w stanie posortować grafy skierowane acykliczne w kolejności topologicznej.

Graf S1. Może być więcej niż jedno źródło/ujęcie.

Chcemy policzyć najkrótsze ścieżki od S do każdego innego wierzchołka.

Założmy że chcemy dojść do A .

$$L(A) = \min\{L(S) + w(s, A), L(C) + w(C, A)\} \quad (15.3.1)$$

Input: $G=(V,E)$

$S \in V$ = source vertice

for each $v \in V$

$L(v) = \infty$ // jeżeli nie ma trasy to dystans będzie ustalony na nieskończoność

$L(S) = 0$

for each $v \in V \setminus \{s\}$ // w porządku topologicznym

$L(v) = \min_{(u,v) \in E} \{L(u) + w(u,v)\}$

Nie chcemy w programowaniu dynamicznym rekurencji, ponieważ nasze podproblemy będą się powtarzać. To jest zasadnicza różnica między programowaniem rekurencyjnym (np. divide and conquer), a dynamicznym. Będziemy zapamiętywać rozwiązania.

Przed pętlą mamy $\Theta(|V|)$, a w pętli $\Theta(\text{indeg}(V))$, suma wszystkich krawędzi przychodzących, czyli mamy $\Theta(|E|)$. Złożoność zadanego algorytmu to $\Theta(n + m)$, w najgorszym przypadku - mając najwięcej $m = n^2$ krawędzi, mamy $\Theta(n + n^2)$.

Jak tworzyć algorytmy dynamiczne:

1. Zdefiniować podproblem.
2. Zdefiniować kolejność na podproblemach.
3. Zdefiniować relację.

15.4 Edit Distance Problem

Input: w_1, w_2 - słowa $|w_1| = n, |w_2| = m, \Sigma$ - alfabet Output: $\text{EditDistance}(w_1, w_2)$ - minimalna liczba operacji dodania, usunięcia, podmiany znaków w słowach $w_1 \rightsquigarrow w_2$

Przykład - chcemy przejść ze SNOWY do SUNNY

S_NOWY

SUNN_Y

010110 -> w sumie 3 operacje.

- Rozpychamy U, zmieniamy O na N, Usuwamy W. Mamy 3 operacje.

_SNOW_Y

SUN__NY

1101110 -> w sumie 5 operacji.

- Wstawiamy S, podmieniamy S na U, Usuwamy O, Usuwamy W, Wstawiamy N

$E(i, j)$ - edit distance $w_1[1 \dots i], w_2[1 \dots j]$

Z jakich podproblemów dochodzimy do $E(i, j)$?

- dodanie litery do w_2 - $E(i, j - 1) + 1$
- usunięcie litery z w_2 - $E(i - 1, j) + 1$ Dopasowujemy do w_2 bez jednej litery.
- podmiana listery z w_2 - $E(i - 1, j - 1) + 1$ Podmiana litery in place
- bez zmian w_2 - $E(i - 1, j - 1)$

$\text{diff}(w_1[i], w_2[i])$ - zwraca 0 lub 1 w zależności czy jest różnica w znakach.

```
for i=0 to m
  E(i,0) = i
for j=0...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) + diff(w[i],w[j]))
```

Analiza złożoności obliczeniowej - pętla1 - $\Theta(m)$, pętla2 - $\Theta(n)$, pętla ostatnia $\Theta(n \cdot m)$.
 Złożoność pamięciowa $\Theta(m \cdot n)$, lub jeśli nie zależy nam na krokach to $\Theta(\min\{m, n\})$
 (pamiętamy każdorazowo ostatnie dwa wiersze).

```

      S N O W Y
    0 1 2 3 4 5
S  1 0 1 2 3 4
U  2 1 1 2 3 4
N  3 2 1 2 3 4
N  4 3 2 2 3 4
Y  5 4 3 3 3 3

```

Therefore

```
EditDistance("SNOWY", "SUNNY") = 3
```

Co na ogół jest podproblemami - np. prefix ciągu, podciąg zwarty (consecutive).

16 Lecture XV

16.1 Kopiec binarny (Binary Heap)

- Pełne drzewo binarne przetrzymywane w tablicy.

```

I : 1 2 3 4 5 6 7 8 9 10
A : [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

```

```

      16
     / \
    14  10
   / \ / \
  8  7 9 3
 / \ /
2  4 1

```

```

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

```

16.2 Własność kopca (maksymalnego)

$$\forall_i A[\text{parent}(i)] > A[i] \quad (16.2.1)$$

- Wysokość węzła to długość najdłuższej prostej ścieżki od tego węzła do liścia.

```

HEAPIFY(A, i)
  l = left(i)
  r = right(i)

```

```

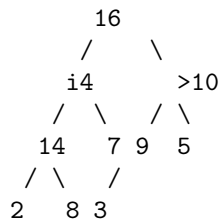
# find maximum element from i, l, r
if (l <= size(A) AND A[l] > A[i])
    largest = l
else
    largest = i
if (r <= size(A) AND A[r] > A[largest])
    largest = r
# end find

# swap if so
if largest != i
    swap(A[i], A[largest])
HEAPIFY(A, largest)

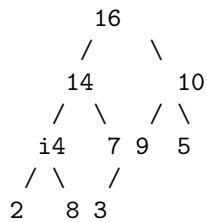
```

Przykład

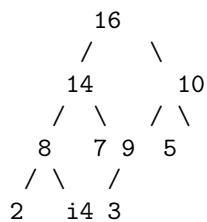
1st step



2nd step



3rd step



Złożoność obliczeniowa

i

```

      /      \
(2/3 el.) (1/3 el.)
..          ..
..          ..
.. (schodek)

```

```

L = 2^h - 1
P = 2^{h-1} - 1 - liczba elementów

```

$$T(n) = O(1) + T(2/3n) \quad (16.2.2)$$

Z Master Theorem mamy:

$$a = 1 \quad (16.2.3)$$

$$b = \frac{3}{2} \quad (16.2.4)$$

$$d = 0 \quad (16.2.5)$$

$$(16.2.6)$$

Zatem $\log_{\frac{3}{2}} 1 = d = 0$, więc mamy $n^d \log n = \log n$

$$T(n) = O(\log n) \quad (16.2.7)$$

$\text{floor}(\text{size}(A)/2)$ to indeks pierwszego nie-liścia. Pierwszym nie-liściem jest parent ostatniego liścia.

```

BuildHeap(A)
  size(A) = length(A)
  for i = floor(size(A)/2) to 1 // i--
    HEAPIFY(A,i)

```

A: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7

```

      4
     / \
    1   3
   / \ / \
  2  16 9 10
 / \ /
14 8 7

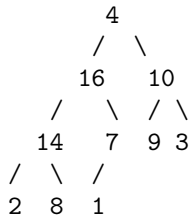
```

```

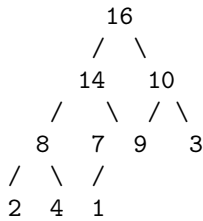
...
intermediate-steps
...

```

AT i = 2



AT i = 1



RESULT: 16, 14, 10, 14, 8, 7, 9, 3, 2, 4, 1

Złożność obliczeniowa dla BuildHeap.

$$|A| = n, \frac{n}{2} \text{razy Heapify} \quad (16.2.8)$$

$$O(n \log n) \quad (16.2.9)$$

Fact. Kopiec. W n -elementowym kopcu binarnym mamy co najwyżej $\lceil \frac{n}{2^{h+1}} \rceil$ węzłów o wysokości h . Dowód.

Indukcja po h . Dla $h = 0$ (liście) mamy co najwyżej $\frac{n}{2^{0+1}}$ liści to jest prawda.

Założenie indukcyjne $\forall_{k < h} \# \text{węzłów o wysokości } k \leq \lceil \frac{n}{2^{k+1}} \rceil$

Krok indukcyjny. Węzły o wysokości $k - 1$ zał. ind $\leq \lceil \frac{n}{2^{k-1+1}} \rceil$. Zatem węzłów o wysokości h mamy co najwyżej $\frac{1}{2} \lceil \frac{n}{2^k} \rceil \leq \lceil \frac{n}{2^{k+1}} \rceil$

Złożoność obliczeniową BuildHeap można również wyrazić jako

$$O \left(\sum_{h=1}^{\log n} \# \text{węzłów o wysokości } h \cdot h \right) \leq \quad (16.2.10)$$

$$\leq O \left(\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot h \right) = \quad (16.2.11)$$

$$= O \left(n \sum_{h=0}^{\log n} \frac{h}{2^{h-1}} \right) = \quad (16.2.12)$$

$$= O \left(\frac{n}{(1 - \frac{1}{2})^2} \right) = \quad (16.2.13)$$

$$= O(n) \quad (16.2.14)$$

Istnieje HeapSort.

16.3 Kolejka Priorytetowa (PQ)

- Insert(Q,x)
- Maximum(Q) : return Q[1], O(1)
- ExtractMax(Q) - zwraca element o najw. priorytecie, usuń z Q
- Increase/Decrease Key(Q,x,y) - zmieniamy z x na y
- Delete(Q,i)
- Union(Q1,Q2) : BuildHeap([Q1,Q2]), O(|Q1|+|Q2|)

```
Delete(Q, i)
    Q[i] = Q[size(Q)]
    size(Q)--
    if (Q[i] < Q[parent(i)])
        Heapify(Q, i)
    else
        while (i > 1 && Q[parent(i)] < Q[i])
            swap(Q[i], Q[parent(i)])
            i = parent(i) : O(log n)
```

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

```
ExtractMax(Q)
    if Q.size < 1 return null
    else
        max=Q[1]
        Q[1] = Q[size(Q)]
        size(Q)--
        Heapify(Q,1)
        return max
```

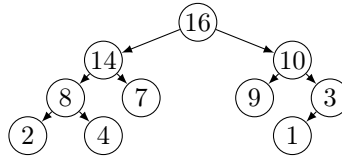
17 Lecture XVI

17.1 Kolejka priorytetowa - Priority Queue

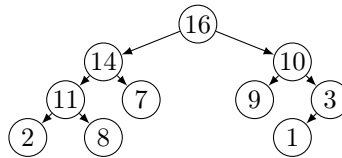

```

Decrease/Increase Key(Q, i, newKey)
  if Q[i] > newKey # decrease
    Q[i] = newKey
    Heapify(Q, i)
  else if Q[i] < newKey # increase
    while i > 1 && Q[parent(i)] < newKey
      Q[i] = Q[parent(i)]
      i = parent(i)
    Q[i] = newKey

```



Decrease/IncreaseKey(Q, 9, 11)



17.2 Inne struktury danych

1. TREAP (1996) - Drzewo BST i Kopiec
2. ZIP-TREE (2021)

17.3 Grafy

Graf prosty to struktura $G = (V, E)$, gdzie:

- V - zbiór wierzchołków $\{1, \dots, n\}$, $|V| = n$
- $E \subseteq \{\{i, j\} : i, j \in V, i \neq j\}$, $|E| = m$

Graf skierowany to struktura $G = (V, E)$, gdzie:

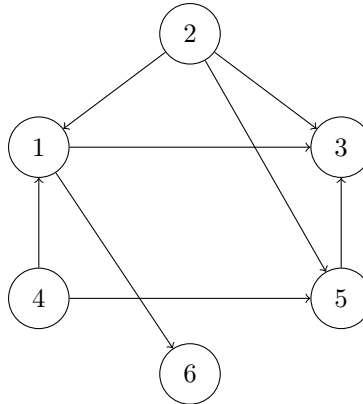
- V - zbiór wierzchołków $\{1, \dots, n\}$, $|V| = n$
- $E \subseteq \{(i, j) : i, j \in V, i \neq j\}$, $|E| = m$

17.4 Listy sąsiedztwa

Grafy mogą być przechowywane w postaci **Listy sąsiedztwa**. Używa się jej w przypadku grafów rzadkich, czyli takich które mają mało krawędzi. Dla każdego wierzchołka V_i przechowujemy listę sąsiadów.

V1 | 3 6
V2 | 1 3 5
V3 | 2
V4 | 1 5
V5 | 3
V6 |

Odpowiada następującemu grafowi:



Złożoność pamięciowa przechowywania tego grafu to $O(n + m) = O(|V| + |E|)$, gdzie m to liczba krawędzi. **Wielkość grafu** definiujemy przez $|V| + |E|$. Zatem jest to liniowe, względem wielkości grafu. Sprawdzenie czy krawędź istnieje można zrobić w $O(n)$, ponieważ musimy przejść przez wszystkie sąsiadujące wierzchołki.

Można listę wskaźnikową zastąpić drzewem BST.

17.5 Macierz sąsiedztwa

Niech $A = (a_{i,j}), i, j \in \{1, \dots, n\}$ będzie macierzą sąsiedztwa grafu $G = (V, E)$.

$$a_{i,j} = \begin{cases} 1 & \text{jeżeli } (i, j) \in E \\ 0 & \text{jeżeli } (i, j) \notin E \end{cases} \quad (17.5.1)$$

Złożoność pamięciowa to $O(n^2)$, ponieważ mamy n^2 elementów w macierzy. Gdy graf jest gęsty $|E| = O(n^2)$ ma to sens. Sprawdzenie czy krawędź istnieje jest w $O(1)$, ponieważ wystarczy zbadać wartość $a_{i,j}$.

;-((Michał tu był)

Drzewo to jest graf, który nie ma cykli.

```
EXPLORE(G,v) # G - Graf, v - wierzchołek startowy
    visited(v) = true
    previsit(v)
    for each edge (v,u) in E
        if not visited(u) EXPLORE(G,u)
    postvisit(v)
```

Mówimy, że G jest **Grafem spójnym**, jeżeli dla każdego wierzchołka $v \in V$ istnieje ścieżka z v do u .

17.6 DFS - Depth First Search

```
DFS(G)
  for each vertex v in G
    visited(v) = false
  for each vertex v in G
    if not visited(v) EXPLORE(G,v)
```

Złożoność obliczeniowa DFS to $O(|V| + |E|)$, ponieważ w najgorszym przypadku przechodzimy przez wszystkie wierzchołki i krawędzie. DFS działa w czasie liniowym od wielkości grafu.

17.7 Zliczanie komponentów spójnych

```
ConnectedComponents
  cc = 1
  previsit(v):
    ccnum[v] = cc
  for each vertex v in G
    visited(v) = false
  for each vertex v in G
    if not visited(v)
      EXPLORE(G,v)
  c++
```

17.8 Globalny zegar

```
previsit(v):
  pre[v] = clock
  clock+=1

postvisit(v):
  post[v] = clock
  clock+=1
```

18 Lecture XVII

18.1 Drzewo przejścia w DFS

Własność 1. Jeżeli istnieje ścieżka z v do u . $u, v \in V$

- $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$, jeżeli istnieje ścieżka z v do u .
- $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$, jeżeli nie istnieje ścieżka z v do u .

Nazewnictwo:

- Tree Edge - krawędź, która prowadzi do potomka w drzewie DFS.

- Back Edge - krawędź powrotna, czyli taka, która prowadzi do wierzchołka, który już został odwiedzony.
- Cross Edge - krawędź do wierzchołka, który nie jest potomkiem.
- Forward Edge - krawędź do potomka, który nie jest bezpośrednim dzieckiem.

Własność 2. $(u, v) \in E$.

- Tree/Forward edge. $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- Back edge. $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- Cross edge. $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$

Własność 3. W grafie skierowanym istnieje cykl iff DFS występuje Back Edge. D-d. $v_0 \rightarrow v_1 \dots v_k \rightarrow v_0$ jest cyklem. Powiedzmy, że DFS odwiedzi jako pierwszy w tym cyklu wierzchołek v_i , dalej eksplorując natknie się na v_{i-1} , wtedy krawędź (v_i, v_{i-1}) będzie Back Edge, ponieważ v_{i-1} jest już odwiedzony.

18.2 Sortowanie topologiczne

Sortowanie topologiczne elementów grafu. Nie jesteśmy w stanie posortować grafu cyklicznego. Chcemy sortować topologicznie grafy skierowane acykliczne (DAG).

- $G = (V, E)$ - graf skierowany acykliczny.
- $(V, <)$ - porządek topologiczny na V .

Musimy zachować $\text{post}(u) < \text{post}(v)$, jeżeli istnieje krawędź $(u, v) \in E$. Pomysł:

1. Najpierw wykonajmy DFS zapisując wartości pre, post
2. Zwróćmy wierzchołki w malejącym porządku po post.

W grafie skierowanym $G = (V, E)$ powiemy, że wierzchołki $u, v \in V$ są **połączone**, jeżeli istnieje ścieżka z u do v oraz z v do u .

Definicja. Silnie spójna komponenta grafu skierowanego $G = (V, E)$. Będziemy mówić, że v_1, \dots, v_k tworzą silnie spójną komponentę w grafie G , jeśli:

- $\forall_{i,j} \quad v_i, v_j \in V$ są połączone.
- Nie istnieje wierzchołek $u \in V$ taki, że u jest połączony z każdym v_i i v_j .

Tworzymy Metagraf Silnie spójnych składowych.

- **Źródłem (Source)** jest wierzchołek, do którego nie wchodzi żadna krawędź z innej komponenty.
- **Ujściem (Sink)** jest wierzchołek, z którego nie wychodzi żadna krawędź do innej komponenty.

Własność 4. Wierzchołek z najwyższą wartością post w silnie spójnej komponentie jest źródłem tej komponenty.

Własność 5. Niech C, C' będą silnie spójnymi składowymi w grafie skierowanym G , oraz istnieje w G krawędź (u, v) , gdzie $u \in C$ i $v \in C'$. Wtedy maksymalna wartość post wierzchołka z C jest większa niż maksymalna wartość post z C' .

D-d. Rozważmy dwa przypadki

- DFS najpierw odwiedzi wierzchołek $u \in C$ przed wierzchołkami z C' . Jasno widzimy, że $\text{post}(u) > \text{post}(v)$.
- DFS najpierw odwiedzi wierzchołek z $v \in C'$ przed wierzchołkami z C . DFS wyeksploruje wierzchołki z C' oraz pozostałe silnie spójne składowe dalej, ale explore nie przejdzie przez C , ponieważ nie może się cofnąć. Następne posty w C będą miały zatem większą wartość niż posty w C' .

Własność 6. Niech $G^R = (V, E^R) : E^R = \{(v, u), (u, v) \in E\}$. Źródło grafu G^R jest ujściem w meta-grafie z G .

Algorytm.

- Input: $G = (V, E)$ - graf skierowany.
- Output: Metagraf silnie spójnych składowych G

Korki algorytmu:

1. Wylicz G^R
2. Wykonaj DFS na G^R i zapisz post.

```
while G nie pusty
    v = wierzchołek z największą wartością post
    S = EXPLORE(G, v)
    V = V \ S
```

Złożoność obliczeniowa algorytmu to $O(|V| + |E|)$, ponieważ wyznaczamy G^R , wykonujemy DFS na grafie G^R oraz G .

19 Lecture XVIII

Pathfinding

20 Lecture XIX

20.1 Dowód dla Dijkstra Algorithm

1. Prezentujemy założenie indukcyjne. $d \in \mathbb{R}_+, w \in \mathbb{R}_+$

$$(\forall_{x \in \mathbb{R}}) \text{dist}(x) \leq d \quad (20.1.1)$$

Czy możemy wagi krawędzi rozszerzyć z \mathbb{R}_+ na \mathbb{R} ?

- W grafie, w którym znajdują się ujemne cykle nie ma sensu przeprowadzać Algorytmu Dijkstry.
- W grafie, w którym nie znajdują się ujemne cykle można przeprowadzić Algorytm Dijkstry, pomimo występowania krawędzi o ujemnych wagach. Nie zmienia to faktu, iż dowód indukcyjny takiego algorytmu jest niemożliwy.

W algorytmie Dijkstry wykonujemy procedurę update, która jest bezpieczna na wielokrotne jej powtarzanie. Jeśli dystans do u był już ustawiony poprawnie oraz na najkrótszej ścieżce od S przechodzi przez u do v , to wtedy dystans do v zostanie poprawnie ustawiony, zakładając, że dystans do u jest poprawnie ustawiony.

```
update((u,v) \in E)
    if dist(u) + w(u,v) < dist(v):
        dist(v) = dist(u) + w(u,v)
        prev(v) = u
```

Jaka jest możliwie najdłuższa możliwa ścieżka w grafie, którego krawędzie mogą mieć ujemne wagi. Najdłuższa ścieżka (w sensie liczby krawędzi) będzie przechodzić przez $|V| - 1$ krawędzi. Nie może być dłuższa, ponieważ wtedy powstałaby ścieżka długości $|V|$, która musiałaby być cyklem z krawędziami ujemnej wagi, a założyliśmy że tak nie jest.

20.2 Algorytm Bellmana-Forda

- Input: $G = (V, E)$, $e \in E : w_e \in \mathbb{R}$, bez ujemnych cykli $s \in V$
- Output: $\forall v \in V$, do którego da się dojść z S , mamy wyznaczone $\text{dist}(v)$, $\text{prev}(v)$ - najkrótszą możliwą ścieżkę

```
for all v in V
    dist(v) = infinity
    prev(v) = null
dist(s) = 0
repeat |V|-1 times
    for all e in E
        update(e)
```

Złożoność obliczeniowa algorytmu - $O(|V| \cdot |E|)$

20.3 Algorytmy Zachłanne

20.4 Definicja Drzewa

Definition. Drzewo. Acykliczny spójny graf nieskierowany.

20.5 Minimalne drzewo rozpinające, MST - Minimum Spanning Tree

Minimalne drzewo rozpinające jest potrzebne do stworzenia najtańszych ścieżek między wierzchołkami.

- Input: Graf $G = (V, E)$, $e \in E$
- Output: Drzewo $T = (V, \mathcal{E})$ t.j. $\mathcal{E} \subseteq E$ oraz $\text{weight}(T) = \sum_{e \in E} w(e)$ jest minimalna.

Chcielibyśmy z grafu stworzyć drzewo, o minimalnej sumie wag składających się na niego krawędzi.

Własności drzewa rozpinającego:

20.6 Własności minimalnego drzewa rozpinającego

- Usunięcie krawędzi należącej do cyklu nie rozspójni grafu.
- Drzewo o n wierzchołkach ma $n - 1$ krawędzi.
- Definicja Równoważna. Każdy spójny nieskierowany graf $G = (V, E)$ taki, że $|E| = |V| - 1$ jest drzewem. Załóżmy, że G ma cykl i $e \in E$ należy do tego cyklu. Wtedy $G - e$ jest grafem spójnym z własności pierwszej. Ale nasz graf G ma $|V| - 1$ krawędzi, więc nie może być spójny, ponieważ usunięcie krawędzi e spowoduje rozspójnienie grafu, zatem G nie może mieć cyklu, więc jest drzewem.
- Minimalne drzewo rozpinające nie musi być unikalne.

20.7 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 .

Dowód. Niech T to minimalne drzewo rozpinające grafu G . Z założeń wiemy, że krawędzie należące do X są częścią minimalnego drzewa rozpinającego. Jeśli $e \in T$ to wszystko jest ok. Załóżmy zatem, że $e \notin T$. Wtedy zmodyfikujmy T w taki sposób, że $\tilde{T} = T \setminus \{e'\} \cup \{e\}$, gdzie e' jest krawędzią z T , która przechodzi pomiędzy S i $V \setminus S$. \tilde{T} jest nieskierowany, ponieważ wszystkie krawędzie biorą się z grafu nieskierowanego. Poprzez usunięcie krawędzi e' krawędź e jest jedyną krawędzią, która przechodzi pomiędzy S i $V \setminus S$, zatem nie może tworzyć cyklu. Spójność grafu \tilde{T} jest zachowana, ponieważ usunięcie krawędzi e' nie rozspójnia grafu. \tilde{T} ma $|V| - 1$ krawędzi, zatem jest drzewem.

Skoro T jest MST:

$$\text{weight}(\tilde{T}) = \text{weight}(T) - \text{weight}(e') + \text{weight}(e) \quad (20.7.1)$$

$$\text{weight}(T) \leq \text{weight}(\tilde{T}) \quad (20.7.2)$$

$$w(e) \leq w(e') \text{ Skoro } e \text{ jest krawędzią o najmniejszej wadze} \quad (20.7.3)$$

W takim razie $\text{weight}(\tilde{T}) = \text{weight}(T)$, zatem \tilde{T} również jest minimalnym drzewem rozpinającym grafu G . Zatem $X \cup \{e\}$ jest częścią minimalnego drzewa rozpinającego grafu G .