Notatki z Algorytmów i Struktur Danych

Jakub Kogut

17 czerwca 2025

Spis treści

1 Wstęp **5**
1.1 Informacje . . . 5
1.2 Ocenianie . . . 5

2 Wykład 2025-03-03 **5**
2.1 Przykładowy Problem . . . 5
2.2 Jak mierzyć złożoność algorytmów . . . 5
2.3 Przykład algorytmu . . . 6
2.4 Przykład działania Merge Sort . . . 6
2.5 Złożoność Merge Sort . . . 7

3 Wykład 2025-03-10 **7**
3.1 Notacja Asypmtotyczna . . . 7
3.2 Rekurencja . . . 9

4 Wykład 2025-03-17 **9**
4.1 Drzewo rekursji . . . 9
4.2 Metoda iteracyjna . . . 10
4.3 Master Theorem . . . 10
4.4 Metoda dziel i zwyciężaj (D&C) . . . 12
4.4.1 Algorytm – Binary Search . . . 12
4.4.2 Algorytm – potęgowanie liczby do naturalnej potęgi . . . 13
4.4.3 Obliczenie n-tej liczby Fibonacciego . . . 13
4.4.4 Mnożenie liczb . . . 14
4.4.5 Mnożenie macierzy . . . 15
4.4.6 Quick Sort . . . 16

5 Wykład 2025-03-24 **17**
5.1 Quick Sort . . . 17
5.1.1 Lemuto Partition . . . 17
5.1.2 Hoare Partition . . . 18
5.1.3 Analiza Worst Case . . . 19
5.1.4 Best Case Analysis . . . 20
5.1.5 Rozważenie przypadku mieszanego . . . 21
5.1.6 Average Case Analysis . . . 21

6 Wykład 2025-03-25 **23**
6.1 Dual Pivot Quick Sort . . . 23
6.1.1 Strategia Count Partition . . . 24
6.2 Comparison Model . . . 24
6.2.1 Drzewo decyzyjne . . . 24
6.2.2 Twierdzenie . . . 25

6.2.3 Counting Sort . . . 25
6.3 Stable Sorting Property . . . 26
6.3.1 Radix Sort . . . 26

7 Wykład 2025-03-31 **27**
7.1 Definicja Statystyki Pozycyjnej . . . 27
7.1.1 Definicja . . . 27
7.1.2 Przykład . . . 27
7.2 Algorytm Random Select . . . 27
7.3 Algorytm Select . . . 30

8 Wykład 2025-04-07 **31**
8.1 Set Interface . . . 31
8.1.1 Binary Search Tree – BST . . . 32
8.1.2 Operacje na BST . . . 32
8.1.3 Wysokość drzewa BST . . . 38

9 Red-Black Trees (RB) **40**
9.1 Własności . . . 41
9.2 Operacje . . . 42
9.2.1 RB-Insert . . . 42
9.2.2 Operacje używane w FixUp . . . 42
9.2.3 RB-Delete . . . 43

10 Wykład 2025-04-28 **43**
10.1 Direct Access Array . . . 43
10.2 Hash Tables . . . 43
10.2.1 Universal Hash Property . . . 43
10.2.2 Wartość oczekiwana kolizji . . . 44
10.3 Wzbogacona struktura danych . . . 44
10.3.1 Algorytm OS-Select . . . 45
10.3.2 OS-Rank . . . 45
10.3.3 Metodologia wzbogacania struktur danych . . . 45
10.3.4 Drzewa Przedziałowe (Interval Trees) . . . 45

11 Wykład 2025-05-05 **47**
11.1 Problem znalezienia najdłuższego rosnącego podciągu . . . 47
11.2 Problem wydawania reszty . . . 48
11.3 Problem plecakowy – Knapsack . . . 48
11.4 Optymalne mnożenie macierzy . . . 49

12 Wykład 2025-05-12 **49**
12.1 Edit Distance . . . 50

13 Wykład 2025-05-19 **51**
13.1 Kopiec binarny . . . 51
13.1.1 Własności kopca (maksymalnego) . . . 52
13.1.2 Heapify . . . 52
13.1.3 Build Heap . . . 54
13.1.4 Przykład . . . 54
13.1.5 Złożoność Obliczeniowa BuildHeap . . . 56
13.2 Kolejka priorytetowa . . . 56
13.3 Podsumowanie . . . 58

14 Wykład 2025-05-20 **58**
14.1 Grafy skierowane . . . 58
14.2 Lista sąsiedztwa . . . 58
14.2.1 Macierz sąsiedztwa . . . 59
14.3 Depth First Search . . . 59
14.3.1 Złożoność obliczeniowa . . . 60

15 Wykład 2025-05-26 **60**
15.1 DFS i porządek topologiczny . . . 60
15.1.1 Własność 1 . . . 60
15.1.2 Własność 2 . . . 60
15.1.3 Własność 3 . . . 60
15.2 Sortowanie topologiczne DAG'ów . . . 60
15.3 costam . . . 61
15.3.1 Własność 4 . . . 61
15.3.2 Własność 5 . . . 61
15.3.3 Własnośc 6 . . . 61
15.4 Strongly Connected Componets algorytm . . . 62

16 Wykład 2025-06-02 **62**
16.1 Przeszukiwanie w szerz (Breath First Search) . . . 62
16.2 Algorytm Dijkstry . . . 63

17 Wykład 2025-06-03 **64**
17.1 Algorytm Bellmana-Forda . . . 64
17.2 Algorytmy zachłanne . . . 64

18 Wykład 2025-06-16 **67**
18.1 Algorytm Kruskala . . . 67

19 Wykład 2025-06-17 **67**
19.1 Min Cut Problem . . . 67
19.2 Algorytm Prim'a . . . 69
19.3 Kodoowanie Huffmana . . . 69
19.3.1 W jaki sposób tworzyć prefix-free codes? . . . 70

20 Ćwiczenia **71**
20.1 Lista 2 . . . 71
20.1.1 zadanie 1 . . . 71
20.1.2 zadanie 2 . . . 71
20.1.3 zadanie 3 . . . 72
20.1.4 zadanie 4 . . . 73
20.1.5 zadanie 5 . . . 74
20.1.6 zadanie 6 . . . 74
20.1.7 zadanie 7 . . . 75
20.2 Lista 3 . . . 75
20.2.1 zadanie 1 . . . 75
20.2.2 zadanie 2 . . . 76
20.2.3 zadanie 3 . . . 76
20.2.4 zadanie 4 . . . 77
20.2.5 zadanie 5 . . . 79
20.2.6 zadanie 6 . . . 80
20.2.7 zadanie 7 . . . 81
20.2.8 zadanie 8 . . . 82
20.2.9 zadanie 9 . . . 82

20.2.10 zadanie 10 83
20.3 lista 5 83
20.3.1 zadanie 6 83
20.4 lista 6 83
20.4.1 zadanie 2 83
20.4.2 zadanie 3 84
20.4.3 zadanie 4 84
20.4.4 zadanie 5 84

1 Wstęp

To będa notatki z przedmiotu Algorytmy i struktury danych na Politechnice Wrocławskiej na kierunku Informatyka Algorytmiczna rok 2025 semestr letni.

1.1 Informacje

Prowadzący Przedmiot: **Zbychu Gołębiewski**

- Należy kontaktować się przez maila: mail
- Konsultacje **216/D1**:
  - Wtorek 13:00-15:00
  - Środa 9:00-11:00
- Wiecej info na stronie przedmiotu
- Literatura
  - Algorithms, Dasgupta, Papadimitriou, Vazirani
  - Algorithms, Sedgewick, Wayne (strona internetowa książki)
  - Algorithms Designs, Jon Kleinberg and Eva Trados
  - Wprowadzenie do algorytmów, Cormen, Leiserson, Rivest, Stein
  - Sztuka programowania (wszystkie tomy), Donald E. Knuth

1.2 Ocenianie

Ocena z kursu składa się z:

- Oceny z egzaminu – E
- Oceny z ćwiczeń – C
- Oceny z laboratorium – L

Wszystkie oceny są z zakresu $[0, 100]$. Ocena końcowa jest wyliczana ze wzoru:

$$K = \frac{1}{2}E + \frac{1}{4}C + \frac{1}{4}L$$

2 Wykład 2025-03-03

2.1 Przykładowy Problem

Sortowanie:

- Input: n liczb $a_1, a_2, \ldots, a_n, |A|$, gdzie $|A|$ to długośc tablicy
- Output: permutacja $a'_1, a'_2, \ldots, a'_n$ taka, że $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Najważniejsze w algorytmach jest to, żeby były POPRAWNE: edge case, ...

2.2 Jak mierzyć złożoność algorytmów

1. Worst Case Analysis T(n) ← stosowane najcześciej
2. Average Case Analysis
   - zakładamy pewnien rozkład prawdopodobieństwa na danych wejściowych
   - T – zmienna losowa liczby operacji wykonanych przez algorytm

     $$T(n) = \max\{\#\text{operacji dla danego wejścia}\}$$

   - $E[T]$ – wartość oczekiwana $T \rightarrow$ średnia liczba operacji, to co nas interesuje

2.3 Przykład algorytmu

W tej sekcji mamy pokazany przykład jak pisać pseudo kod:

Algorithm 1 Merge Sort

```
1: procedure MergeSort(A, 1, n)
2:     if |A[1..n]| == 1 then
3:         return A[1..n]
4:     else
5:         B = MergeSort(A, 1, ⌊n/2⌋)
6:         C = MergeSort(A, ⌊n/2⌋, n)
7:         return Merge(B, C)
8:     end if
9: end procedure
```

Algorithm 2 Merge

```
1: procedure Merge(X[1..k], Y[1..n])
2:     if X = ∅ then
3:         return Y
4:     else if Y = ∅ then
5:         return X
6:     else if X[1] ≤ Y[1] then
7:         return [X[1]] × Merge(X[2..k], Y[1..n])
8:     else
9:         return [Y[1]] × Merge(X[1..k], Y[2..n])
10:    end if
11: end procedure
```

2.4 Przykład działania Merge Sort

Example: Sorting the array [10, 2, 5, 3, 7, 13, 1, 6] **step by step**

1. **Initial split:**

$$[\,10,\,2,\,5,\,3,\,7,\,13,\,1,\,6\,] \quad \longrightarrow \quad [\,10,\,2,\,5,\,3\,] \quad \text{and} \quad [\,7,\,13,\,1,\,6\,].$$

2. **Sort the left half** [10, 2, 5, 3]**:**
   (a) Split into [10, 2] and [5, 3].
   (b) MergeSort([10, 2]):
   - Split into [10] and [2].
   - Each is already sorted (single element).
   - Merge: [2, 10].

   (c) MergeSort([5, 3]):
   - Split into [5] and [3].
   - Each is already sorted.
   - Merge: [3, 5].

   (d) Merge [2, 10] and [3, 5] to get [2, 3, 5, 10].

3. **Sort the right half** [7, 13, 1, 6]**:**
   (a) Split into [7, 13] and [1, 6].

(b) MergeSort([7, 13]):
- Split into [7] and [13].
- Each is already sorted.
- Merge: [7, 13].

(c) MergeSort([1, 6]):
- Split into [1] and [6].
- Each is already sorted.
- Merge: [1, 6].

(d) Merge [7, 13] and [1, 6] to get [1, 6, 7, 13].

4. **Final merge:** Merge the two sorted halves:

$$[\,2,\,3,\,5,\,10\,] \quad \text{and} \quad [\,1,\,6,\,7,\,13\,] \quad \longrightarrow \quad [\,1,\,2,\,3,\,5,\,6,\,7,\,10,\,13\,].$$

Hence, after all the recursive splits and merges, the final sorted array is:

$$[\,1,\,2,\,3,\,5,\,6,\,7,\,10,\,13\,].$$

2.5 Złożoność Merge Sort

- Złożoność czasowa
  - $T(n) = 2T(n/2) + \Theta(n)$
  - $T(n) = \Theta(n \log n)$
- Złożoność pamięciowa
  - $M(n) = n + M(n/2)$
  - $M(n) = \Theta(n)$

3 Wykład 2025-03-10

3.1 Notacja Asypmtotyczna

Na wykładzie będziemy omawiali:

- Notację dużego O $O(n)$ //ograniczenie górne
  - Definicja $O(n)$:

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

  - Uwaga!
    Jeśli

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

    to

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

  - Przykład:
    * $2n^2 = O(n^3)$ dla $n_0 = 2, c = 1$ Definicja jest spełniona
    * $f(n) = n^3 + O(n^2)$ jest to jeden z sposobów użycia $O(n)$

$$\exists h(n) = O(n^2) \quad \text{takie, że} \quad f(n) = n^3 + h(n)$$

- Notację omega //ograniczenie dolne
  - Definicja
$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n)\}$$
  - Przykład
    - $n^3 = \Omega(2n^2)$
    - $n = \Omega(\log n)$
- Notację theta $\theta(n)$ //ograniczenie z dwóch stron
  - Definicja
$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$
  - Przykład
    - $n^3 = \Theta(n^3)$
    - $n^3 = \Theta(n^3 + 2n^2)$
    - $logn + 8 + \frac{1}{12n} = \Theta(\log n)$
  - Uwaga!
$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$
    Można to zapisać jako klasy funkcji:
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$
- Patologiczny przykład: mamy funkcje $g(n) = n$ oraz $f(n) = n^{1+sin\frac{\pi n}{2}}$, a więc
$$f(n) = \begin{cases} n^2 & \text{dla n parzystych} \\ n & \text{dla n nieparzystych} \end{cases}$$
  wtedy
$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$
$$\limsup_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$
  zatem $f \neq O(g)$ oraz $g \neq O(f)$
- o małe
  - Definicja
$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < c \cdot g(n)\}$$
    Równoważnie
$$lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
  - Przykład
    - $n^2 = o(n^3)$ i $n^2 O(n^3)$ ale $n^2 \neq o(n^2)$
    - $n = o(n^2)$

3.2 Rekurencja

- Metoda podstawienia (metoda dowodu indukcyjnego)
  1. Zadnij Odpowiedź (bez stałych)
  2. Sprawdź przez indukcję czy odpowiedź jest poprawna
  3. Wylicz stałe

  – Przykład
    * $T(n) = T(\frac{n}{2}) + n$
    * Pierwotny strzał: $T(n) = O(n^3)$
    * cel: Pokazać, że $\exists c > 0 : T(n) \leq c \cdot n^3$
      · warunek początowy: $T(1) = 1 \leq c$
      · krok indukcyjny: załóżmy, że $\forall k \leq n : T(k) \leq ck^3$

$$T(n) = 4T(\frac{n}{2}) + n \leq 4c(\frac{n}{2})^3 + n = \frac{1}{2}cn^3 + n \leq cn^3 \quad \text{dla} \quad c \geq 2$$

    jednakże "Przestrzeliliśmy" znacznie, spróbojmy wzmocnić założenie indukcyjne:

$$T(n) \leq c_1k^2 - c_2k, k < n$$

    wtedy mamy:

$$T(n) = 4T(\frac{n}{2}) + n \leq 4(c_1(\frac{n}{2})^2 - c_2(\frac{n}{2})) + n = c_1n^2 - 2c_2n + n \leq c_1n^2 - c_2n$$

    zatem $c_1 = 1, c_2 = 1$ i $T(n) = O(n^2)$ □

  – Przykład
    * $T(n) = 2T(\sqrt{n}) + \log n$
    załóżmy, że n jest potęgą liczby 2, czyli $n = 2^m$

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

    Co implikuje

$$T(2^{\frac{m}{2}}) \to S(m)$$

    wtedy

$$S(m) = 2S(\frac{m}{2}) + m$$

    rozwiązując rekurencję otrzymujemy

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

    zatem

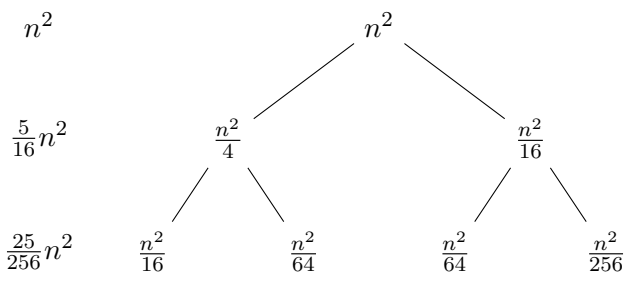
$$T(n) = \log n \log \log n$$

4 Wykład 2025-03-17

4.1 Drzewo rekursji

Przykład dzewa rekursji:

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



Uwaga!

Nie jest to formalne rozwiązanie problemu. Nie można używać drzewa rekursji do dowodzenia złożoności algorytmów. Jest to jedynie intuicyjne podejście do problemu. Trzeba policzyć to na piechote, aby było formalnie.
Aby policzyć $T(n)$ musimy policzyć sumę wszystkich wierzchołków w drzewie rekursji.

$$T(n) = \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k \cdot n^2 = n^2 \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k = n^2 \frac{1}{1-\frac{5}{16}} = n^2 \frac{16}{11} = \frac{16}{11} n^2$$

A wiec $T(n) = O(n^2)$
Możemy to policzyć dokładniej dostajac mniejsze wyrazy w sumie.

$$T(n) = O(\hat{T}(n)) = O(\check{T}(n))$$

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

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

4.2 Metoda iteracyjna

Weźmy na przykład taką rekurencję:

$$T(n) = 3T(\frac{n}{4}) + n$$

Zobaczmy co się dzieje po podstawieniu rekurencji do samej siebie:

1. $T(n) = 3T(\frac{n}{4}) + n$
2. $T(n) = 3(3T(\frac{n}{16}) + \frac{n}{4}) + n = 3^2 T(\frac{n}{16}) + \frac{3}{4}n + n$
3. $T(n) = 3^2(3T(\frac{n}{64}) + \frac{n}{16}) + \frac{3}{4}n + n = 3^3 T(\frac{n}{64}) + \frac{3}{16}n + \frac{3}{4}n + n$
4. ...[1]

A więc ogólnie wychodzi:

4.3 Master Theorem

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

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

gdzie a i b są stałymi, a $f(n)$ jest funkcją nieujemną. Wtedy:

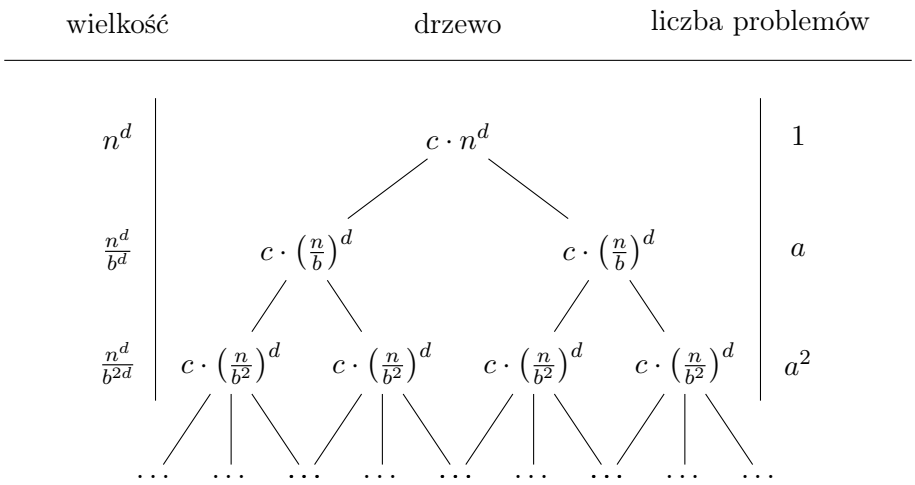
1. $\Theta(n^d)$ jeśli $d > \log_b a$
2. $\Theta(n^d \log n)$ jeśli $d = \log_b a$
3. $\Theta(n^{\log_b a})$ jeśli $d < \log_b a$

[1] Warto zauważyć, że jest to analogicznie do liczenia sumy wszystkich nodów drzewa rekursji

Szkic D-d

Do przedstawienia problemu użyjemy drzewa rekursji. Rozważmy rekurencję:

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



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

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

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

4. rozważmy 3 przypadki:

   (a) $d > \log_b a$ root – he
   $$T(n) = \Theta(n^d)$$ avy

   (b) $d = \log_b a$ równo
   $$T(n) = \Theta(n^d \log n)$$

   (c) $d < \log_b a$ leaf – he
   $$T(n) = \Theta(n^{\log_b a})$$ avy

Przykłady

- $T(n) = 4T(\frac{n}{2}) + 11n$
  Wtedy kożystając z **Master Theorem** mamy:
  $$a = 4, b = 2, d = 1$$
  Jak i również
  $$\log_b a = \log_2 4 = 2 > 1 = d \implies T(n) = \Theta(n^2)7$$

- $T(n) = 4T(\frac{n}{3}) + 3n^2$
  Wtedy
  $$a = 4, b = 3, d = 2$$
  Jak i również
  $$\log_b a = \log_3 4 > 2 = d \implies T(n) = \Theta(n^{\log_3 4})$$
- $T(n) = 27T(\frac{n}{3}) + \frac{n^2}{3}$
  Wtedy
  $$a = 27, b = 3, d = 2$$
  Jak i również
  $$\log_b a = \log_3 27 = 3 > 2 = d \implies T(n) = \Theta(n^3 \log n)$$

4.4 Metoda dziel i zwyciężaj (D&C)

Na czym ona polega?

1. Podział problemu na mniejsze podproblemy [2]
2. Rozwiazanie rekurencyjnie mniejsze podpoblemy
3. połącz rozwiązania podproblemów w celu rozwiązania problemu wejściowego

4.4.1 Algorytm – Binary Search

- **Input**: posortowania tablica `A[1..n]` oraz element `x`
- **Output**: indeks `i` taki, że `A[i] = x` lub `0` jeśli `x` nie występuje w `A`
- przebieg algorytmu:

Algorithm 3 Binary Search

```
1:  procedure BinarySearch(A, x)
2:      l = 1
3:      r = |A|
4:      while l ≤ r do
5:          m = ⌊(l+r)/2⌋
6:          if A[m] = x then
7:              return m
8:          else if A[m] < x then
9:              l = m + 1
10:         else
11:             r = m − 1
12:         end if
13:     end while
14:     return 0
15: end procedure
```

- **Asypmtotyka** Algorytm spełnia następująca rekurencje:
  $$T(n) = T(\frac{n}{2}) + \Theta(1)$$
  Rozwiązując za pomocą **Master Theorem** otrzymujemy:
  $$T(n) = \Theta(\log n)$$

[2] W zapisie rekurencyjnym $T(n) = cT(cn) + \underline{n^d}$

4.4.2 Algorytm – potęgowanie liczby do naturalnej potęgi

- **Problem**: obliczanie x^n
  Można rozbić mnożenie n x na odpowiednie podproblemy:

$$x^n = \underbrace{x \cdot x \cdot \cdots \cdot x}_{\frac{n}{2}} \cdot \underbrace{x \cdot x \cdot \cdots \cdot x}_{\frac{n}{2}}$$

  A więc mamy:

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{dla n parzystych} \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{dla n nieparzystych} \end{cases}$$

- **Asymptotyka**:
  Algorytm spełnia następującą rekurencję:

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

  Rozwiązując za pomocą **Master Theorem** otrzymujemy:

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

4.4.3 Obliczenie n-tej liczby Fibonacciego

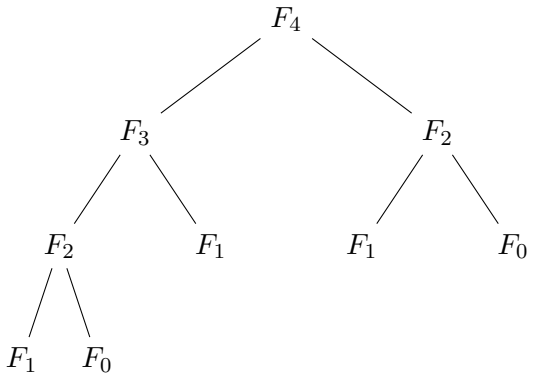
- **Problem**:

$$F_n = \begin{cases} 0 & \text{dla n} = 0 \\ 1 & \text{dla n} = 1 \\ F_{n-1} + F_{n-2} & \text{dla n} > 1 \end{cases}$$

- **Algorytmy**:

  1. Naiwna rekurencja używająca definicji.

     Obliczanie F_4



     Kontynułując dostajemy asymptotyke rzędu $\Theta(\phi^n)$
  2. *bottom up* – iteracyjne obliczanie kolejnych liczb Fibonacciego. Asymptotyka wynosi $\Theta(n)$
  3. Kożystanie z wzoru wynikającego z rozwiązanej rekurencji:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

     Problem z tym podejsciem polega na niedokładnym przybiłżeniu przez komputery wartości ϕ

4. Kożystając z lematu:
$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n = \begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix},$$

Dowód. (a) Warunek początkowy: dla $n = 0$
$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^0 = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}F_1 & F_0\\ F_0 & F_{-1}\end{pmatrix}$$

(b) Krok indukcyjny:
załóżmy, że dla pewnego k zachodzi:
$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^k = \begin{pmatrix}F_{k+1} & F_k\\ F_k & F_{k-1}\end{pmatrix}$$

wtedy dla $k+1$ mamy:
$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{k+1} = \begin{pmatrix}F_{k+2} & F_{k+1}\\ F_{k+1} & F_k\end{pmatrix}$$
$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{k+1} = \begin{pmatrix}F_{k+1} & F_k\\ F_k & F_{k-1}\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix} = \begin{pmatrix}F_{k+1}+F_k & F_{k+1}\\ F_k+F_{k-1} & F_k\end{pmatrix} = \begin{pmatrix}F_{k+2} & F_{k+1}\\ F_{k+1} & F_k\end{pmatrix}$$

□

Algorytm ten ma złożoność $\Theta(n \log n)$

4.4.4 Mnożenie liczb

- **Input**: x, y takie, że $\max\{|x|, |y|\}$
- **Output**: $x \cdot y$
- **Algorytmy**:
  1. standardowe mnożenie szkolne – mnożenia w słupku jego asyptotyka wynosi $\Theta(n^2)$
  2. Podejście metodą **D&C**
     - **Podejście**: Rozbijamy liczby na dwie równe części, a następnie mnożymy je przez siebie Możemy zapisać x oraz y jako:
$$x = \underbrace{x_L \cdot 2^{\frac{n}{2}}}_{\frac{n}{2}\text{bitów}} + \underbrace{x_R}_{\frac{n}{2}\text{bitów}}$$
$$y = \underbrace{y_L \cdot 2^{\frac{n}{2}}}_{\frac{n}{2}\text{bitów}} + \underbrace{y_R}_{\frac{n}{2}\text{bitów}}$$
Proces mnożenia wygląda następująco:
$$\begin{aligned} x \cdot y &= (x_L \cdot 2^n + x_R) \cdot (y_L \cdot 2^n + y_R) \\ &= x_L y_L \cdot 2^{2n} + ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) \cdot 2^n \\ &\quad + x_R y_R \end{aligned}$$
Generalnie wszytkie wykonywane powyżej operacje są giga tanie bo opreacje takie jak mnożenie przez 2^k wiąże się jedynie z przesunięciem bitowym.
     - **Asymptotyka**: Nasz algorytm spełnia następującą rekurencje na podstawie zapisanego wyżej równania
$$T(n) = 4T(\frac{n}{2}) + \Theta(n)$$
Kożystając ponownie z **Master Theorem** można wywnioskować, że algorytm ma złożoność $\Theta(n^2)$. Zatem nie ma żadnego znacznego przyśpieszenia, nawet prawdopodobnie stała ukryta w $\Theta(n^2)$ jest gorsza niż w standardowym podjesciu

3. Metoda Gaussa
   - Rozważmy mnożenie liczb zespolonych

$$(a+ib)(c+id) = ac + i(ad+bc) + bd$$

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

   zatem

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

   - **Asymptotyka**: algorytm ten spełnia rekurencje

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

   Z **Master Theorem** otrzymujemy, że algorytm ma złożoność $\Theta(n^{\log_2 3})$, a $\log_2 3 \approx 1.58$

4. Istnieją jeszcze szybsze, nowsze algorytmy mnożenia liczb, takie jak algorytm Schönhage'a-Strassena bazuje ono na szybkiej transformacie Fouriera *Fast Fourier Transform*, który ma złożoność $\Theta(n \log n \log\log n)$. Jednakże, trzeba wziąść pod uwagę stałą ukrytą w Θ. W praktyce, dla liczb o rozmiarze do 10^6 lepiej jest użyć standardowego algorytmu mnożenia.

Trochę pseudo kodu dla mnożenia liczb:

Algorithm 4 Mnożenie liczb

```
1:  procedure MULTIPLY(x, y)
2:      n = max{|x|, |y|}
3:      if n = 1 then
4:          return x · y
5:      end if
6:      x_L, x_R = LetfMost ⌈n/2⌉ , RightMost ⌈n/2⌉ bits of x
7:      y_L, y_R = LetfMost ⌈n/2⌉ , RightMost ⌈n/2⌉ bits of y
8:      p_1 = Multiply(x_L, y_L)
9:      p_2 = Multiply(x_R, y_R)
10:     p_3 = Multiply(x_L + x_R, y_L + y_R)
11:     return p_1 · 2^{2n} + (p_3 − p_1 − p_2) · 2^n + p_2
12: end procedure
```

4.4.5 Mnożenie macierzy

- **Input**: dwie macierze A, B rozmiaru $n \times n$
- **Output**: macierz $C = A \cdot B$
- **Algorytmy**:
  1. Naiwne mnożenie macierzy – jego złożoność wynosi $\Theta(n^3)$ bo aby policzyć jedną komórkę macierzy C musimy wykonać n mnożeń (i $n-1$ dodawań [3]), a skoro macierz C ma n^2 komórek to złożoność wynosi $\Theta(n^3)$
  2. Algorytm Strassena – **D&C**
     - **Podejście**: Rozbijamy macierze na 4 równe części

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

[3] w sumie n^2 operacji

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

gdzie

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

– **Asymptotyka**: Algorytm ten spełnia rekurencje

$$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$$

Z **Master Theorem** otrzymujemy, że algorytm ma złożoność $\Theta(n^{\log_2 7})$, a $\log_2 7 \approx 2.81$ [4]

4.4.6 Quick Sort

Algortym **Merge Sort** ociera się o minimalną granicę złożoności sortowania, która wynosi $\Theta(n \log n)$, jednaże jest z nim problem związany z pamięcia: nie sortuje w miejscu, a więc wymaga dodatkowej pamięci.

- **Input**: tablica $A[1..n]$
- **Output**: posortowana tablica A
- **Algorytm**: `QuickSort(A, p, q)`
  1. Podziel tablicę `A[p...q]` na dwie podtablice `A[p...k-1]` oraz `A[k+1...q]`, gdzie `A[k]` jest elementem rozdzielającym – *pivotem*[5] tak, że:

$$\forall i \in [p...k-1] : A[i] \leq A[k] : \forall j \in [k+1...q] : A[j] \geq A[k]$$

  2. Odpalamy rekurencyjnie `QuickSort(A, p, k-1)` oraz `QuickSort(A, k+1, q)`
- **Przyklad**:
  1. mamy dane $A = [6, 1, 4, 3, 5, 7, 2, 8]$, wybieramy *pivot* jako 6
  2. Przebieg partycjonowania:

$$A = [1, 4, 3, 5, 2, 6, 7, 8]$$

  Elementy mniejsze niż 6 znalazły się po lewej stronie, większe po prawej. Pozycja pivota: $A[6] = 6$.
  3. Rekurencyjnie sortujemy dwie części:
     – `QuickSort(A, 1, 5)` dla tablicy $[1, 4, 3, 5, 2]$, np. wybieramy pivot 1:

$$A = [1, 4, 3, 5, 2]$$

     Po dalszym sortowaniu otrzymamy $[1, 2, 3, 4, 5]$
     – `QuickSort(A, 7, 8)` dla $[7, 8]$, który już jest posortowany.

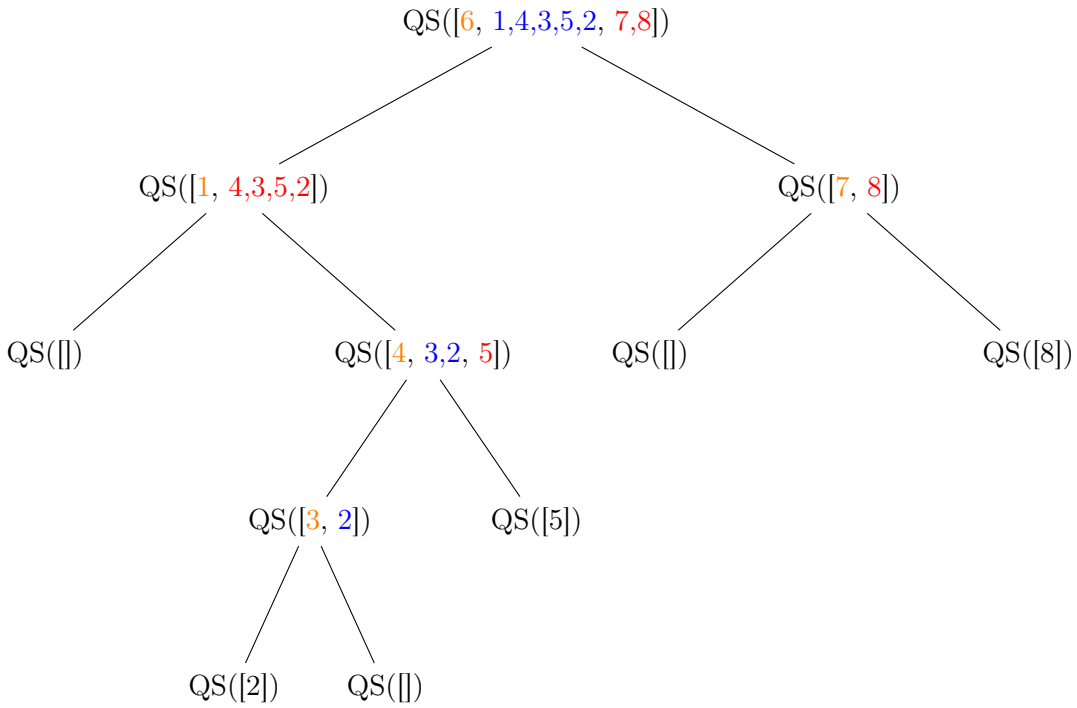
[4] Aby zejść do rekurencji $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$ trzeba wykonać pewne, bardziej wyrafinowane triki, które nie są dokładnie opisane tutaj. Z algorytmu zapisanego wyżej wynika że rekurencja to $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$, a wieć złożoność wynosi $\Theta(n^3)$

[5] o tym jak ten pivot jest wybierany będziemy mówić później

4. Finalna posortowana tablica:

$$A = [1, 2, 3, 4, 5, 6, 7, 8]$$

Na koniec przykład w drzewie rekursji:



5 Wykład 2025-03-24

5.1 Quick Sort

5.1.1 Lemuto Partition

- **Input**: tablica $A[1..n]$
- **Output**: posortowana tablica A
- **Algorytm**: `Lemuto(A, p, q)`

Algorithm 5 Lemuto Partition

```
1:  procedure LEMUTO(A, p, q)
2:      pivot = A[p]
3:      i = p
4:      for j = p + 1 to q do
5:          if A[j] < pivot then
6:              i = i + 1
7:              swap A[i] ↔ A[j]
8:          end if
9:      end for
10: end procedure
```

- **Przykład:**
  1. zaczynamy z nieposotowaną tablicą $A = [6, 10, 13, 5, 8, 3, 2, 11]$
  2. wybieramy pivot $A[1] = 6$
  3. inicjalizujemy $i = 1$

4. iterujemy przez tablicę od $j = 2$ do $j = 8$:
   - $j = 2$: $A[2] = 10$ (nie mniejsze od pivot)
   - $j = 3$: $A[3] = 13$ (nie mniejsze od pivot)
   - $j = 4$: $A[4] = 5$ (mniejsze od pivot)
     * $i = i + 1 = 2$
     * zamiana $A[2] \leftrightarrow A[4] \Rightarrow A = [6, 5, 13, 10, 8, 3, 2, 11]$
   - $j = 5$: $A[5] = 8$ (nie mniejsze od pivot)
   - $j = 6$: $A[6] = 3$ (mniejsze od pivot)
     * $i = i + 1 = 3$
     * zamiana $A[3] \leftrightarrow A[6] \Rightarrow A = [6, 5, 3, 10, 8, 13, 2, 11]$
   - $j = 7$: $A[7] = 2$ (mniejsze od pivot)
     * $i = i + 1 = 4$
     * zamiana $A[4] \leftrightarrow A[7] \Rightarrow A = [6, 5, 3, 2, 8, 13, 10, 11]$
   - $j = 8$: $A[8] = 11$ (nie mniejsze od pivot)
5. zamiana pivot $A[1] \leftrightarrow A[4] \Rightarrow A = [2, 5, 3, 6, 8, 13, 10, 11]$
6. pivot 6 jest na pozycji 4

- **Asymptotyka**: Algorytm ten wykonuje w głownej pętli $n-1$ porównań, natomiast wersja Lemuto Partition wymaga dodatkowo $n - 1$ zamian elementów.

5.1.2 Hoare Partition

- **Input**: Tablica $A[1..n]$
- **Output**: Posortowana Tablica A
- **Algorytm**: `Hoare(A, p, q)`

Algorithm 6 Hoare Partition

```
1:  procedure HOARE(A, p, q)
2:      pivot = A[(p+q)/2]
3:      i = p − 1
4:      j = q + 1
5:      while True do
6:          i = i + 1
7:          while A[j] > pivot do
8:              j = j − 1
9:              if ≥ j then
10:                 break
11:             end if
12:         end while
13:         swapA[i] ↔ A[j]
14:     end while
15: end procedure
```

- **Przykład:** Generalnie algorytm ten działa na zasadzie zamiany elementów w tablicy względem *pivotu* tak, że jeżeli jest element mniejszy od *pivotu* to zamieniamy go z elementem większym od *pivotu* z drugiej strony tablicy. Algorytm kończy się gdy wszytkie elementy mniejsze od *pivotu* są po lewej stronie, a większe po prawej.

1. zaczynamy z nieposotowaną tablicą $A = [6, 10, 13, 5, 8, 3, 2, 11]$
2. wybieramy pivot $A[\frac{1+8}{2}] = A[4] = 5$

3. inicjalizujemy $i = 0$ i $j = 9$
4. iterujemy aż $i \geq j$:
   - $i = 1$: $A[1] = 6$ (większe od pivot)
   - $j = 8$: $A[8] = 11$ (większe od pivot)
     - $j = 7$: $A[7] = 2$ (mniejsze od pivot)
     - zamiana $A[1] \leftrightarrow A[7] \Rightarrow$

2	10	13	5	8	3	6	11

   - $i = 2$: $A[2] = 10$ (większe od pivot)
   - $j = 6$: $A[6] = 6$ (większe od pivot)
     - $j = 5$: $A[5] = 3$ (mniejsze od pivot)
     - zamiana $A[2] \leftrightarrow A[5] \Rightarrow$

2	3	13	5	8	10	6	11

   - $i = 3$: $A[3] = 13$ (większe od pivot)
   - $j = 4$: $A[4] = 8$ (większe od pivot)
     - $j = 3$: $A[3] = 13$ (większe od pivot)
     - zamiana $A[3] \leftrightarrow A[3] \Rightarrow$

2	3	5	13	8	10	6	11

   - $i = 4$: $A[4] = 8$ (większe od pivot)
   - $j = 3$: $A[3] = 5$ (pivot), kończymy algorytm
5. pivot 5 jest na pozycji 3 i wszystkie elementy są podzielone względem niego

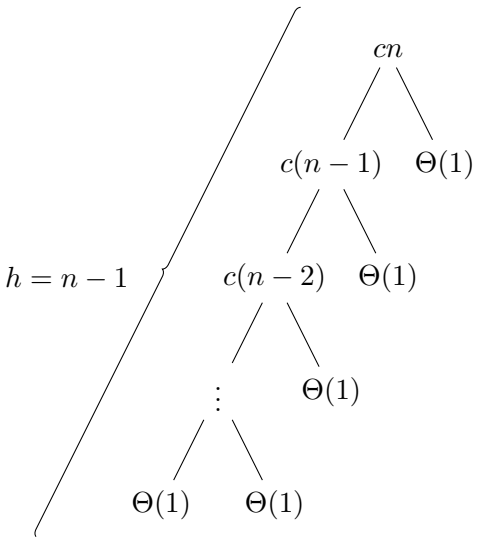
- **Asymptotyka**: *Hoare Partition* wykonue $n \pm c$ porównań – o stałą więcej niżeli *Lemuto Partition*, ale za to wykonuje mniej zamian elementów. W praktyce *Hoare Partition* jest szybszy. Całościowa Asymptotyka wynosi $\Theta(n)$

5.1.3 Analiza Worst Case

Algorytm sortowania Quick Sort zachowuje się najgorzej w przypadku, gdy dostaje tablicę odwrotnie posortowaną. Wszystkie elementy będą znajdowały się po złej stronie *pivotu*.
Zostaje spełniana rekurencja:

$$T(n) = T(n-1) + \underbrace{T(0)}_{\text{pusta lewa tablica}} + \Theta(n)$$

Można zauważyć, że nie zadziała tu **Master Theorem**, trzeba rozwiązać ja na przykład drzewem rekursji:



Z drzewa rekursji wynika, że powyższa rekurencja to:

$$\begin{aligned} T(n) &= T(n-1+\Theta(n) \\ &\leq \sum_{k=0}^{n} (c(n-k)+\Theta(1)) \\ &= c\sum_{k=0}^{n}(n-k)+\Theta(n) \\ &= c\sum_{k=0}^{n} k+\Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Ograniczenie dolne analogicznie...

5.1.4 Best Case Analysis

Algorytm sortowania Quick Sort zachowuje się najlepiej w przypadku, gdy dostaje tablicę posortowaną. Wszystkie elementy będą znajdowały się po dobrej stronie *pivotu*.
Zostaje spełniana rekurencja:

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

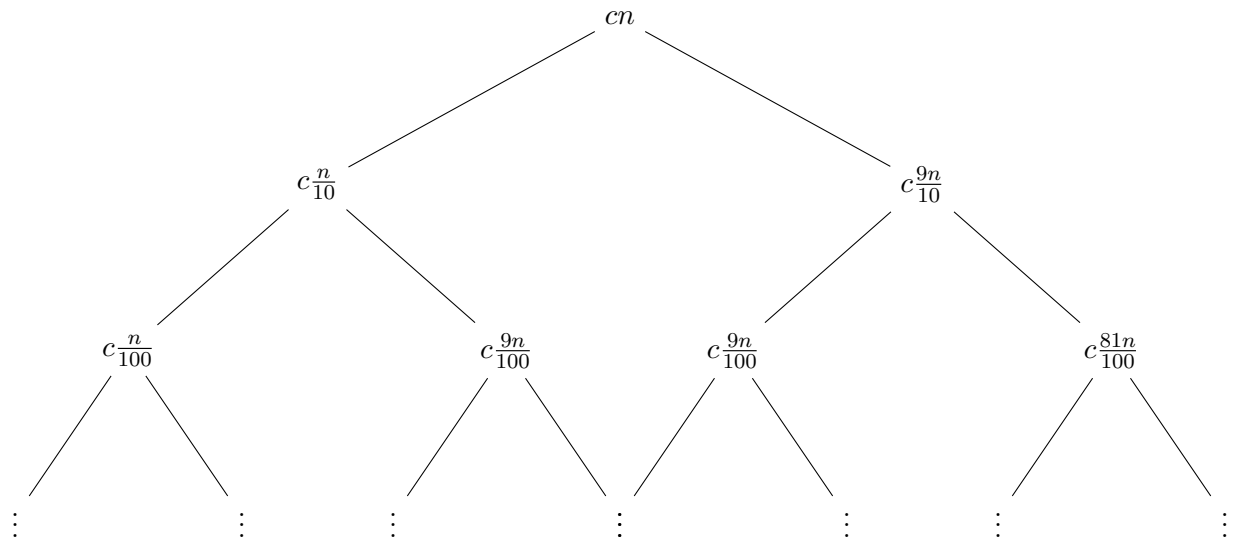
Można zauważyć, z **Master Theorem**, że asymptotyka wynosi:

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

Rozważmy przypadek, w którym algorytm wykonuje się nie koniecznie optymalną ilość razy. Powiedzmy, że spełnia on taką rekurencje:

$$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$$

Rozważając drzewo rekursji możemy zauważyć, że



Każdy wiersz tego drzewa sumuje się do cn. Wysokość drzewa wynosi $\log_{10/9} n$, zatem złożoność wynosi $\Theta(n \log_{10/9} n)$, co jest tak naprawdę równe $\Theta(n \log n)$.

5.1.5 Rozważenie przypadku mieszanego

Rozważmy przypadek, w którym algorytm raz wykonuje się z best casem – dzieli się tablica na pół, a raz z worst casem – dzieli się tablica na 1 i $n-1$ elementów.
Zostaje spełniana rekurencja:

$$L(n) = 2U(\frac{n}{2}) + \Theta(n)$$
$$U(n) = L(n-1) + \Theta(n)$$

gdzie L symbolizuje best case, natomiast U worst case. Rozwiązując powyższą rekurencje otrzymujemy:

$$\begin{aligned} L(n) &= 2(L(\frac{n}{2} - 1) + \Theta(n)) + \Theta(n) \\ &= 2L(\frac{n}{2} - 1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

5.1.6 Average Case Analysis

Algorytm Quick Sort da się "zabezpieczyć" przed złym rozkładem danych poprzez losowym wybraniem pivota i następnie swapnięcie go z naszym deterministycznym miejscem. W ten sposób bedziemy mieli zawsze jednostajnie losowy rozkład danych.
Wprowadźmy

$$T_n \text{ – zmienna losowa liczby porównań w Quick Sorcie sortowanej tablicy } A, |A| = n$$

Do dziś nie jest znany rozkład zmiennej losowej T_n.
Niech X będzie zmienną indykatorową:

$$X_k^{(n)} = \begin{cases} 1 \text{ jeśli partition podzieli tablice n-elementową na } (k, (n-k-1)) \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

Teraz rozważmy zachowanie zmiennej losowej T_n:

$$T_n \stackrel{\mathrm{d}}{=} \begin{cases} T_0 + T_{n-1} + n - 1 \text{ jeśli } (0, n-1) \text{ jest partitionem} \\ T_1 + T_{n-2} + n - 1 \text{ jeśli } (1, n-2) \text{ jest partitionem} \\ \vdots \\ T_k + T_{n-k-1} + n - 1 \text{ jeśli } (k, n-k-1) \text{ jest partitionem} \\ \vdots \\ T_{n-1} + T_0 + n - 1 \text{ jeśli } (n-1, 0) \text{ jest partitionem} \end{cases}$$

Stosując zmienną indykatową X otrzymujemy

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

Rozważmy niezależność zmiennych $X_k^{(n}$ i T_k. Są one niezależne, ponieważ ilość porównań nie jest zależna od tego jak poźniej bedzie dzielić się tablica. Zatem można zapisać

$$\mathbb{E}[X_k^{(n)} T_k] = \mathbb{E}[X_k^{(n)}]\mathbb{E}[T_k]$$

Skoro przyjmujemy jednostajny rozkład danych wejścowych to wartość oczekiwana $X_k^{(n)}$ wynosi:

$$\mathbb{E}[X_k^{(n)}] = 1 \cdot P(X_k^{(n)} = 1) + 0 \cdot P(X_k^{(n)} = 0) = P(X_k^{(n)} = 1) = \frac{(n-1)!}{n!} = \frac{1}{n}$$

Teraz policzmy wartość oczekiwaną T_n

$$\begin{aligned}
\mathbb{E}[T_n] &= \mathbb{E}\left[\sum_{k=0}^{n-1} X_k^{(n)}(T_k + T_{n-k-1} + n - 1)\right] \\
&= \sum_{k=0}^{n-1} \mathbb{E}[X_k^{(n)}(T_k + T_{n-k-1} + n - 1)] \\
&= \sum_{k=0}^{n-1} \mathbb{E}[X_k^{(n)}]\mathbb{E}[T_k + T_{n-k-1} + n - 1] \\
&= \sum_{k=0}^{n-1} \frac{1}{n}\mathbb{E}[T_k + T_{n-k-1} + n - 1] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} \mathbb{E}[T_k] + \mathbb{E}[T_{n-k-1}] + \mathbb{E}[n-1] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} \mathbb{E}[T_k] + \frac{1}{n}\sum_{k=0}^{n-1} \mathbb{E}[T_{n-k-1}] + \frac{1}{n}\sum_{k=0}^{n-1} n - 1
\end{aligned}$$

Można zauważyć, że $\sum_{k=0}^{n-1} \mathbb{E}[T_k] = \sum_{k=0}^{n-1} \mathbb{E}[T_{n-k-1}]$ ponieważ jest to doawanie tych samych rzeczy w innej kolejności (od przodu i od tyłu). Zatem

$$\begin{aligned}
\mathbb{E}[T_n] &= \frac{2}{n}\sum_{k=0}^{n-1} \mathbb{E}[T_k] + \frac{1}{n}\sum_{k=0}^{n-1} n - 1 \\
&= \frac{2}{n}\sum_{k=0}^{n-1} \mathbb{E}[T_k] + n - 1
\end{aligned}$$

Przyjmijmy oznaczenie $\mathbb{E}[T_n] = t_n$, wtedy

$$t_n = \frac{2}{n}\sum_{k=0}^{n-1} t_k + n - 1$$

Jest to rekurencja Można ją rozwiązać w następujący sposób:

$$t_n = \frac{2}{n}\sum_{k=0}^{n-1} t_k + n - 1 \quad | \cdot n$$

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

Podstawmy za $n \to n-1$

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

Odejmując stronami równanie otrzymujemy:

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

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

Przekształcając otrzymujemy:

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

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

Przyjmijmy kolejne oznaczenie $s_n = \frac{t_n}{n+1}$, wtedy

$$s_n = s_{n-1} + 2\frac{n-1}{n(n+1)}$$

jest już prostą rekurencją, którą można łatwo rozwiązać iteracyjnie.

$$\begin{aligned}
s_n &= 2\sum_{k=1}^{n} \frac{k-1}{k(k+1)} \\
&= 2\sum_{k=1}^{n} \left(\frac{2}{k+1} - \frac{1}{k}\right) \\
&= 4\sum_{k=1}^{n} \frac{1}{k+1} - 2\sum_{k=1}^{n} \frac{1}{k} \\
&= 4\left(H_n + \frac{1}{n+1} - 1\right) - 2H_n \\
&= 4H_n + 4\frac{1}{n+1} - 4 - 2H_n \\
&= 2H_n + \frac{4}{n+1} - 4
\end{aligned}$$

gdzie H_n to n-ty element ciągu harmonicznego. Podstawiając z powrotem $t_n \leftarrow s_n(n+1)$ otrzymujemy

$$t_n = 2(n+1)H_n + 4(n+1) - 4 = 2nH_n + 2H_n + 4n$$

Kożystając z faktu, że $H_n = \ln n + \gamma + O(\frac{1}{n})$ otrzymujemy

$$\mathbb{E}[T_n] = 2n\log n + 2\gamma n + \Theta(n)$$

6 Wykład 2025-03-25

6.1 Dual Pivot Quick Sort

W roku 1975 Sedgewick pokazał, że

$$\mathbb{E}\left[\#\text{porównań w dual pivot partition}\right] \approx \frac{16}{9}n \implies$$

$$\implies \mathbb{E}\left[\#\text{porównań w dual pivot Quick Sortcie Sedgwick}\right] \approx \frac{32}{15}n\log n$$

Jak się to liczy?
Niech T_n - #porównań w dual pivot Quick Sortcie Sedgwick oraz P_n - #porównań w dual pivot partition. Rekurencja spełnia takie równanie:

$$\mathbb{E}\left[T_n\right] = \mathbb{E}\left[P_n\right] + \frac{1}{\binom{n}{2}} \sum_{1\le p\le q\le n} \left(\mathbb{E}[T_{p-1}] + \mathbb{E}[T_{q-p-1}] + [T_{n-q}]\right)$$

Poźniej rozwiązuje się to analogicznie jak na poprzednim wykładzie rozwiązywana była rekurencja dla T_n 5.1.6

Natomiast w roku 2009 pan Yaroslavsky, Bentley, Blach opracował poprzez testowanie w Javie (nie miał backgroundu matematycznego) lepszy algortytm Quick Sort. Poźniej w 2012 Sebastian Wild, Nedel pokazali, że

$$\mathbb{E}\left[\#\text{porównań w Dual Pivot Quick Sorcie Yaroslavskiego}\right] \approx 1.9n\log n$$

2015, Aufmüller, Dietzfelbinger zaprojektowali *Strategie Count* oraz pokazali jej optymalność

$$\mathbb{E}\left[\#\text{porównań w Count Partition}\right] \approx \frac{3}{2}n \implies$$

$$\implies \mathbb{E}\left[\#\text{porównań w Dual Pivot z Count Partition}\right] \approx 1.8n \log n$$

Wartość oczekiwana pojawia się w tych asymptotykach ponieważ jest element losowści zwiazany z porównaniami elementu z pivotami. Nie zawsze trzeba bedzię go porównywać z jednym i drugim pivotem.

6.1.1 Strategia Count Partition

- Zakładamy $p < q$

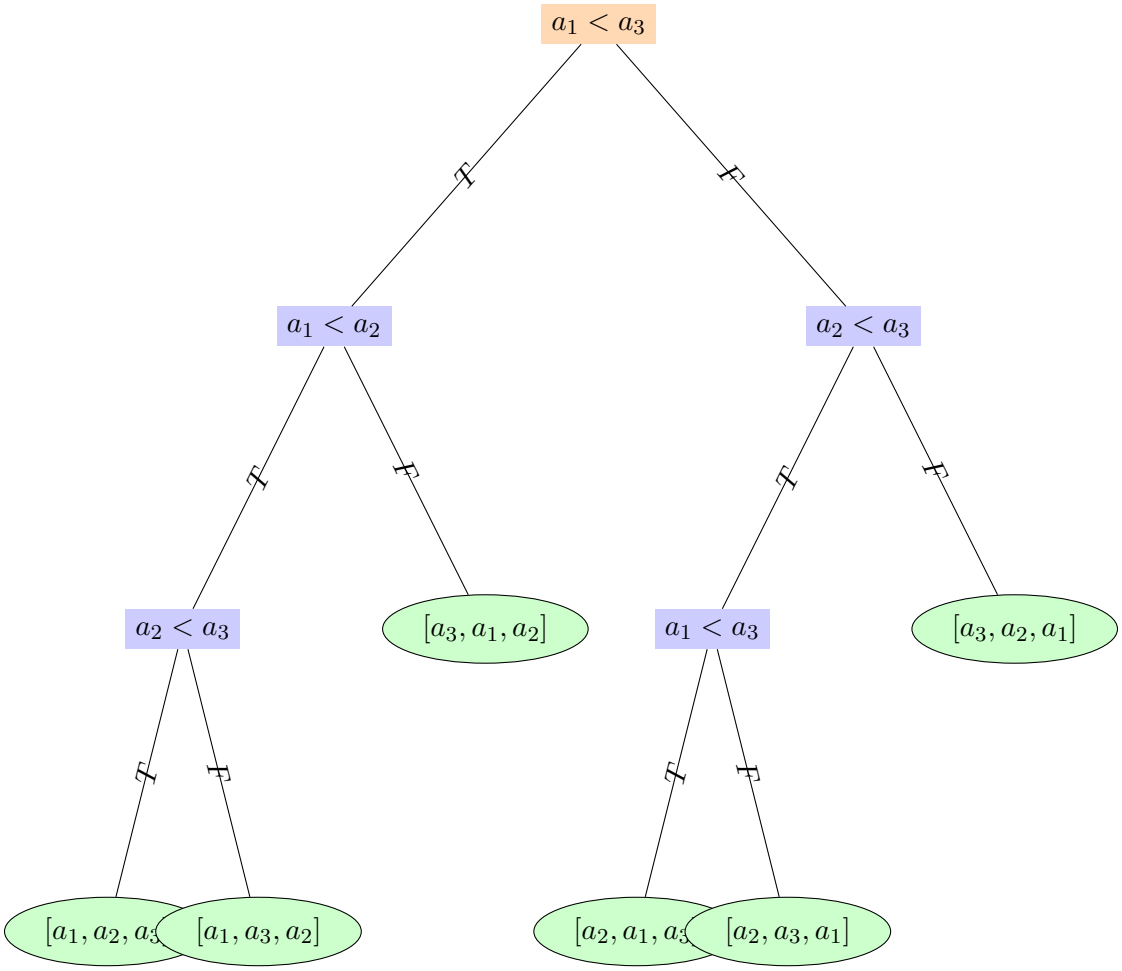
$$\left[\, a \,\middle|\, \underbrace{\cdots}_{S_{i-1}}\, p \,\middle|\, \cdots\, q \,\middle|\, \underbrace{\cdots}_{L_{i-1}}\, \boxed{i} \,\middle|\, b \,\right]$$

- rozpatrzmy i-ty element tablicy
- jeśli $s_{i-1} \geq l_{i-1}$ to porównujemy $A[i]$ najpierw z p jeżeli jest potrzeba to z q
- jesli $s_{i-1} \leq l_{i-1}$ to porównujemy $A[i]$ najpierw z q jeżeli jest potrzeba to z p[6]

6.2 Comparison Model

6.2.1 Drzewo decyzyjne

Jak wygląda to na przykładzie? Mamy daną tablice $[a_1, a_2, a_3]$ i chcemy ją posortować. W drzewie decyzyjnym każdy węzeł to porównanie dwóch elementów. W liściu znajduje się permutacja elementów.



[6] jest to swoiste przewidywanie przeszłości na podstawie ilości elementów, które są już posortowane

- dla dowolnego algorytmu sortowania w *Comparison Model* można znaleść drogę na drzewie decyzyjnym
- W tym drzewie występuje $n!$ liści będacych permutacjami tablicy
- *Worst Case* odpowiada najdłuższej ścieżce w drzewie decyzyjnym
- drzewo binarne pełne odpowiada najlepszemu algorytmowi sortowania
- drzewo binarne pełne o wysokości h ma conajwyżej 2^h liści, ale liści w drzewie decyzyjnym powinno być przynajmniej $n!$ zatem

$$2^h \geq n! \implies h \geq \log_2 n!$$

6.2.2 Twierdzenie

Dolne ograniczenie na liczbę porównań w problemie sortowania w *Comparison Model* wynosi $\Omega(n \log n)$

Dowód. Rozważmy drzewo decyzyjne dla sortowania n elementów. Każda ścieżka w drzewie decyzyjnym odpowiada jednej permutacji. Zatem drzewo decyzyjne ma przynajmniej $n!$ liści. Trzeba pokazać, że wysokość drzewa decyzyjnego jest przynajmniej $\log_2 n!$.

$$2^h \geq n! \implies h \geq \log n!$$

Używając wzoru Stirlinga

$$\begin{aligned} h \geq \log n! &= \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1+O(1))\right) \\ &= \log\left(\frac{n}{e}\right)^n + \log\left(\sqrt{2\pi n}(1+O(1))\right) \\ &= \underbrace{n\log n}_{\Theta(n\log n)} - \underbrace{n\log e}_{\Theta(n)} + \underbrace{\log\sqrt{2\pi n}}_{\Theta(\log n)} + \underbrace{\log(1+O(1))}_{\Theta(1)} \\ &= \Omega(n\log n) \end{aligned}$$

Zatem dolne ograniczenie na liczbę porównań w problemie sortowania w *Comparison Model* wynosi $\Omega(n \log n)$

□

6.2.3 Counting Sort

- **Input**: Tablica $A, |A| = n, \forall i A[i] \in \{0, 1, 2, ..., k\}$, k - stała
- **Output**: Posortowana tablica A
- **Algorytm**:
- **Przykład**:
  1. $A = [4, 1, 3, 4, 3]$, $C = [0, 0, 0, 0]$, $B = [0, 0, 0, 0, 0]$ (tablice indeksowane od 1)
  2. $A = [4, 1, 3, 4, 3]$, $C = [1, 0, 2, 2]$, $B = [0, 0, 0, 0, 0]$
  3. $A = [4, 1, 3, 4, 3]$, $C = [1, 1, 3, 5]$, $B = [0, 0, 0, 0, 0]$
  4. Ostatnia pętla, skupmy się na tablicy B:
     (a) $j = 5$: $B[5] = A[5] = 3$, $C[3] = 4$
     (b) $j = 4$: $B[4] = A[4] = 4$, $C[4] = 3$
     (c) $j = 3$: $B[3] = A[3] = 3$, $C[3] = 3$
     (d) $j = 2$: $B[3] = A[2] = 1$, $C[1] = 1$
     (e) $j = 1$: $B[1] = A[1] = 4$, $C[4] = 2$
     $B = [1, 3, 3, 4, 4]$ - posortowana tablica
- **Asymptotyka**: $\Theta(n + k)$, gdzie $k = O(n)$

Algorithm 7 Counting Sort

```
1: procedure CountingSort(A, n, k)
2:     for i = 1 to k do                                          ▷ Θ(k)
3:         C[i] = 0
4:     end for
5:     for j = 1 to n do                                          ▷ Θ(n)
6:         C[A[j]] = C[A[j]] + 1
7:     end for
8:     for i = 2 to k do                                          ▷ Θ(k)
9:         C[i] = C[i] + C[i − 1]
10:        C[i] = C[i] + C[i − 1]          ▷ liczba elementów mniejszych lub równych i
11:    end for
12:    for j = n downto 1 do                                      ▷ Θ(n)
13:        B[C[A[j]]] = A[j]
14:        C[A[j]] = C[A[j]] − 1
15:    end for
16:    return B                                                   ▷ Θ(1)
17: end procedure
```

6.3 Stable Sorting Property

Algorytm zachowuje kolejność równych sobie elementów z tablicy wejściowej

6.3.1 Radix Sort

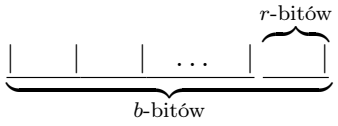
- **Input**: Tablica $A, |A| = n, \forall i A[i] \in \{0, 1, 2, ..., k\}$, k - stała
- **Output**: Posortowana tablica A
- **Algorytm**: Zastosuj Counting Sort dla każdej cyfry liczby
- **Przykład**:

$$A = \begin{bmatrix} 329 \\ 457 \\ 657 \\ 839 \\ 436 \\ 720 \\ 355 \end{bmatrix} \xrightarrow{\text{Counting Sort}} \begin{bmatrix} 720 \\ 355 \\ 436 \\ 457 \\ 657 \\ 329 \\ 839 \end{bmatrix} \xrightarrow{\text{Counting Sort}} \begin{bmatrix} 329 \\ 355 \\ 436 \\ 457 \\ 657 \\ 720 \\ 839 \end{bmatrix}$$

  Algorytm zachowuje kolejność równych sobie elementów z tablicy wejściowej – *Stable Sorting Property*, co umożliwia sortowanie po kolejnych cyfrach.

- **Poprawność**: Indukcja po t – numer cyfry
  1. jeśli $t = 1$ to poprawność *Counting Sort* jest trywialna
  2. załóżmy, że Counting Sort jest poprawny dla $t - 1$ – cyfrowej liczby
  3. Krok indukcyjny:
     (a) t-ta cyfra ... liczby jest ...: to z założenia indukcyjnego oraz *stable counting property Counting Sorta* liczby do t–tej cyfty dajej poprawnie posortowane
     (b) t-ta cyfra różna: z poprawności *Counting Sort* ok.
- **Złożoność obliczeniowa**:
  – n b-bitowych liczb

– liczba b-bitowa dzielę na r-bitowych cyfr ($\frac{b}{r}$ takich liczb):



– cyfry są z $\mid \{0, \ldots, 2^r - 1\} \mid = 2^r$

– *Counting Sort* sortujemy $\underline{n}$ liczb względem jednej liczby. A więc wykonujemy

$$\Theta(n + 2^r)$$

operacji.

Zatem *Radix Sort* będzie miał złożoność obliczeniową

$$\Theta\left(\frac{b}{r}(n + 2^r)\right)$$

machając trochę rękoma wybieramy r jako $r = \log n$, wtedy

$$\Theta\left(\frac{b}{\log n}\left(n + 2^{\log n}\right)\right) = \Theta\left(\frac{bn}{\log n}\right)$$

Jeśli $\{0, \ldots, n^d - 1\}$ - zakres sortowanych liczb, wtedy $b = \log n^d = d \log n$, a więc

$$\Theta\left(\frac{d \log n n}{\log n}\right) = \Theta\left(\frac{\cancel{\log n} n}{\cancel{\log n}}\right) = \Theta(d \cdot n)$$

7 Wykład 2025-03-31

Problem którym się teraz zajmiemy to tak zwany **Statystyka Pozycyjna**

7.1 Definicja Statystyki Pozycyjnej

7.1.1 Definicja

k-tą statystyką pozycyjną nazywamy k-tą najmniejszą wartością z zadnaego zbioru.

7.1.2 Przykład

- $k = 1 \to O(n)$
- $k = n \to O(n)$
- $k = \left\lfloor \frac{n-1}{2} \right\rfloor$ $k = \left\lfloor \frac{n+1}{2} \right\rfloor \to$ sortowanie $O(n \log n)$

7.2 Algorytm Random Select

- **Input**: Tablica $A, |A| = n$, liczba k
- **Output**: k-ta statystyka pozycyjna
- **Algorytm**:
- **Przykład**: $A = [6, 10, 13, 5, 8, 3, 2, 11]$, RandomSelect$(A, 1, 8, 4)$ Dla tablicy

$$A = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\ \hline \end{array}$$

wywołujemy procedurę `RandomSelect(A, 1, 8, 4)` aby znaleźć 4-ty najmniejszy element.

Algorithm 8 Random Select

```
1: procedure RandomSelect(A, p, q, i)
2:     if p = q then
3:         return A[p]
4:     end if
5:     r = RandomPartition(A, p, q)                ▷ losowa partycja
6:     k = r − p + 1
7:     if i = k then
8:         return A[r]                              ▷ A[r] jest i-tą statystyką pozycyjną
9:     else if i < k then
10:        return RandomSelect(A, p, r − 1, i)
11:    else
12:        return RandomSelect(A, r + 1, q, i − k)
13:    end if
14: end procedure
```

Krok 1.
Zakładamy, że procedura `RandomPartition` wybiera pivot 8.
Po partycjonowaniu tablica wygląda następująco:

$$A = \begin{array}{cccccccc} 6 & 5 & 3 & 2 & \underline{8} & 13 & 10 & 11 \end{array}$$

Pivot znajduje się na pozycji $r = 5$, a $k = r - p + 1 = 5 - 1 + 1 = 5$. Ponieważ poszukiwany rząd $i = 4$ jest mniejszy od $k = 5$, następuje wywołanie rekurencyjne:

$$\texttt{RandomSelect}(A, 1, 4, 4)$$

dla lewego podzbioru

$$\boxed{6}\boxed{5}\boxed{3}\boxed{2}.$$

Krok 2.
Dla podtablicy

$$B = \boxed{6}\boxed{5}\boxed{3}\boxed{2},$$

zakładamy, że pivot został wybrany jako 2.
Po partycjonowaniu otrzymujemy:

$$B = \boxed{\underline{2}}\boxed{5}\boxed{3}\boxed{6}$$

gdzie pivot 2 znajduje się teraz na pozycji $r = 1$. Obliczamy $k = r - p + 1 = 1 - 1 + 1 = 1$.
Ponieważ $i = 4 > 1$, odejmujemy k od i, czyli nowa wartość to $i = 4 - 1 = 3$, i wywołujemy:

$$\texttt{RandomSelect}(B, 2, 4, 3)$$

dla podtablicy

$$B' = \boxed{5}\boxed{3}\boxed{6}.$$

Krok 3.
Dla podtablicy

$$C = \boxed{5}\boxed{3}\boxed{6},$$

zakładamy, że pivotem jest 6.
Po partycjonowaniu otrzymujemy:

$$C = \boxed{5}\boxed{3}\boxed{\underline{6}}$$

gdzie pivot 6 znajduje się na pozycji $r = 3$ (przyjmując indeksowanie od 1). Wówczas $k = r - p + 1 = 3 - 1 + 1 = 3$.
Skoro $i = 3$ jest równe k, zwracamy pivot:

$$\texttt{RandomSelect}(C, 1, 3, 3) = 6.$$

Wniosek:
Procedura zwraca wartość $\boxed{6}$, czyli 4-ty najmniejszy element w tablicy A.

- **Złożoność obliczeniowa**:
  - *Best Case*: W najlepszym przypadku dzielimy tablicę na dwie równe części, zatem

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

    Używając *Master Theorem* otrzymujemy

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

  - *Worst Case*: W najgorszym przypadku dzielimy tablicę na $n - 1$ i 1 elementów (– pivot wybrany przez partition), a więc

    $$T(n) = T(n-1) + \Theta(n)$$

    Używając *Master Theorem* otrzymujemy

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

  - *Average Case*: W średnim przypadku musimy policzyć wartość oczekiwaną takiej zmiennej losowej:

    $$T_n = \begin{cases} T_{n-1} + n - 1 : (0, n-1) \\ T_{n-2} + n - 1 : (1, n-2) \\ \vdots \\ T_1 + n - 1 : (n-2, 1) \end{cases}$$

    Aby formalizować analizę, wprowadzamy zmienną indykatorową:

    $$X_k^{(n)} = \begin{cases} 1, & \text{jeśli przy partycjonowaniu tablicy } n\text{-elementowej nastąpiło rozbicie na} \\ & (k, n-k-1) \text{ oraz dalsze wywołanie rekurencyjne odbywa się na} \\ & \text{podtablicy zawierającej szukany element,} \\ 0, & \text{w przeciwnym przypadku.} \end{cases}$$

    Wówczas możemy zapisać:

    $$T_n = \sum_{k=0}^{n-1} X_k^{(n)} \left(T\Big(\max(k,\, n-k-1)\Big) + n - 1 \right).$$

    Przyjmując, że wybór pivotu jest jednostajny, mamy:

    $$\mathbb{E}\Big[X_k^{(n)}\Big] = \frac{1}{n} \quad \text{dla } k = 0, 1, \ldots, n-1.$$

    Zatem, stosując wartość oczekiwaną i liniowość tej wartości, otrzymujemy:

    $$\mathbb{E}[T_n] = \frac{1}{n} \sum_{k=0}^{n-1} \left(\mathbb{E}\Big[T\big(\max(k,\, n-k-1)\big)\Big] + n - 1 \right).$$

Ponieważ gdy pivot jest szukanym elementem (dla $k = 0$ lub $k = n - 1$, w sensie odpowiedniego ustawienia) nie następuje rekurencja, przyjmujemy $T_0 = 0$. Dla pozostałych przypadków (czyli dla $1 \leq k \leq n - 2$) dalsze wywołanie odbywa się na jednej z podtablic o rozmiarze nie większym niż $\lceil n/2 \rceil$ (ze względu na symetrię rozbicia). Możemy więc oszacować:

$$t_n = \mathbb{E}[T_n] \leq n - 1 + \frac{n-2}{n} t_{\lceil n/2 \rceil}.$$

Łatwo (przez indukcję) pokazać, że taka rekurencja implikuje, iż

$$t_n = O(n).$$

Wniosek: Średnia liczba porównań w algorytmie RandomSelect wynosi $\Theta(n)$, czyli algorytm działa w czasie liniowym w średnim przypadku.

7.3 Algorytm Select

Algorytm zwany jest także algorytmem *Magicznych Piątek*

- **Input**: Tablica $A, |A| = n$, liczba k
- **Output**: k-ta statystyka pozycyjna
- **Algorytm**:
  1. dzielimy $A[p \dots q]$ na $\lfloor \frac{n}{5} \rfloor$ pięcio elementowe częsci oraz ostatnią część z resztą $\Theta(n)$
  2. Sortujemy każdą pięcio elementową część i wybieramy z nich mediany: $T(\frac{\lceil n \rceil}{5}$

$$M = \left\{ m_1, m_2, \dots, m_{\lfloor \frac{n}{5} \rfloor} \right\}$$

  3. Znaleść medianę z tablicy M: wywołujemy Select$(M, 1, \lceil \frac{n}{5} \rceil, \left\lfloor \frac{\lceil \frac{n}{10} \rceil}{2} \right\rfloor)$ i oznaczmy ją jako X[7]
  4. Ustaw X jako pivot w Partition $\Theta(n)$
  5. idz do lewej albo prawej podtablicy w zależności od indeksu pivota i szukaj statystykju pozycyjnej $T(?)$
- **Asymptotyka**: Algorytm spełnia rekurencję:

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

Gdzie $T(?)$ oznacza czas rekurynceyjnego wywołania algorytmu Select dla reszty tablicy (pozotałych elementów niżeli napewno mniejsze/większe od X). Rozmiar *?* jest zależny od dystrybucji elementów w tablicy. W najgorszym przypadku ograniczenie dolne na *pewną część* jest rzędu:

$$\geq \left(\frac{1}{2}\frac{\lceil n \rceil}{5}\right) - 1 - 1 \cdot 3 \geq \frac{3n}{10} - 6$$

Zatem górne ograniczenie na *?* jest rzędu:

$$n - \left(\frac{3n}{10} - 6\right) - 1 \leq \frac{7n}{10} + 5$$

A więc Algorytm *Select* spełnia rekurencję:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right) + \Theta(n)$$

[7] nazywana też mediana median

Dla uproszczenia możemy przyjąć $T\left(\frac{7n}{10}+5\right) = T\left(\frac{3n}{4}\right)$, ponieważ $\frac{3}{4}n \geq \frac{7}{10}n + 5$ dla $n \geq 20$. Przyjmujemy, że dla n<20 algorytm działa w czasie stałym. Reukrencja algorytmu *Select* jest teraz następująca:

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

Rozwiążemy ją stosując indukcję:

- Base case: $T(n) = \Theta(1)$ dla $n < 20$
- Założenie indukcyjne:

$$\forall k \leq n : T(k) \leq ck$$

- Krok indukcyjny:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + \Theta(n) \\ &\leq c \cdot \frac{n}{5} + c \cdot \frac{3n}{4} + \Theta(n) \\ &= \frac{c}{5}n + \frac{3c}{4}n + \Theta(n) \\ &\leq \left(\frac{c}{5} + \frac{3c}{4} + \Theta(1)\right) n \\ &= \left(\frac{4c+15c}{20} + \Theta(1)\right) n \\ &= \left(\frac{19c}{20} + \Theta(1)\right) n \end{aligned}$$

Dla naszego wybranego, dużego c możemy przyjąć, że $\frac{19c}{20} + \Theta(1) \leq c$, zatem

$$T(n) \leq cn$$

□

8 Wykład 2025-04-07

8.1 Set Interface

Zakładamy, że każda struktura ma pole nazwane kluczem $a \in A \rightarrow \exists 1 \geq a.key \geq$ length() , które jest unikalne dla każdego elementu. Klucz jest liczbą całkowitą, a jego wartość jest niezmienna. Wartości kluczy są różne dla różnych elementów, a więc $a.key \neq b.key$ dla $a \neq b$.
Do budowy zbiorów używamy *Set Interface*, który definiuje następujące operacje:

- build(A) - buduje "set" z danych zawartych w tablicy A
- find(k) - zwraca element o kluczu k z "setu"
- length() - zwraca liczbę elementów w "secie"
- insert(a) - dodaje element a do "setu"
- delete(a) - usuwa element a z "setu"
- find_min() - zwraca element o najmniejszym kluczu
- find_max() - zwraca element o największym kluczu
- find_next(k) - zwraca element o kluczu większym od k
- find_prev(k) - zwraca element o kluczu mniejszym od k

- order() - zwraca elementy w "secie" w kolejności rosnącej kluczy

Zobaczmy sobie kilka przykładow asymptotyki operacji dla różnych struktur danych:

Struktura	**build**	**find**	**input, delete**	**find_max find_min**	**find_next find_prev**	**order()**
unsorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$[8]	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
sorted array	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
unsorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1), \Theta(n)$[9]	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
BST	$\Theta(n)$	–	–	–	–	–
Direct Access Array	$\Theta(K)$	$\Theta(1)$	$\Theta(1)$	$\Theta(K)$	$\Theta(K)$	$\Theta(K)$

8.1.1 Binary Search Tree – BST

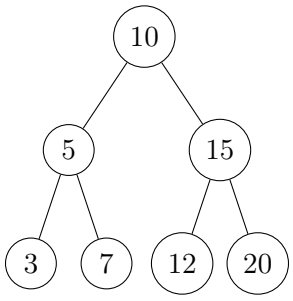
Po polsku drzewo przeszukiwań binarnych. Struktura ta zawiera następujące pola:

- parent – wskaźnik na rodzica
- left – wskaźnik na lewe dziecko
- right – wskaźnik na prawe dziecko
- key – klucz
- ... – inne pola

Drzewa BST mają następujące właściwości, niech T będzie drzewem BST, $x \in T$ – x jest węzłem drzewa T

- każdy $y \in x.left$ ma klucz mniejszy od klucza x
- każdy $y \in x.right$ ma klucz większy od klucza x

Przykład drzewa BST:



8.1.2 Operacje na BST

- **InorderTreeWalk**: operacja do wypisania wszystkich elementów w drzewie w kolejności rosnącej kluczy. Algorytm działa rekurencyjnie, odwiedzając najpierw lewe poddrzewo, następnie węzeł, a na końcu prawe poddrzewo.
  Na powyższym przykładzie wywołanie `InorderTreeWalk(`T`)` wypisze 3, 5, 7, 10, 12, 15, 20.

[8] nie $\Theta(1)$ ponieważ potrzeba czasu na realokacje pamięci itp.
[9] należy znaleść element do usunięcia, samo usuwanie $\Theta(1)$

Algorithm 9 InorderTreeWalk

```
1: procedure INORDERTREEWALK(x)
2:     if x ≠ NIL then
3:         InorderTreeWalk(x.left)
4:         print(x.key)
5:         InorderTreeWalk(x.right)
6:     end if
7: end procedure
```

Asymptotyka: w zależności od rozmiaru lewego poddrzewa – k, algorytm spełnia rekurencję:

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

Rozwiążemy to rekurencyjnie:

- założenie indukcyjne: $\forall k < n : T(k) \leq ck$
- krok indykcyjny:

$$\begin{aligned} T(n) &= T(j) + \Theta(1) + T(n-1-j) \\ &\leq cj + \Theta(1) + c(n-1-j) \\ &= cn - c + \Theta(1) \\ &\leq cn \end{aligned}$$

Zatem asymptotyka algorytmu to $T(n) = \Theta(n)$

- **TreeSearch**: operacja do wyszukiwania elementu w drzewie. Algorytm działa rekurencyjnie, porównując klucz szukanego elementu z kluczem węzła. Jeśli klucz jest mniejszy, algorytm przeszukuje lewe poddrzewo, jeśli większy – prawe poddrzewo.
  Na powyższym przykładzie wywołanie `TreeSearch(`T`, 12)` zwróci węzeł o kluczu 12.

Algorithm 10 TreeSearch

```
1: procedure TREESEARCH(x, k)
2:     if x = null or k = x.key then
3:         return x
4:     else if k < x.key then
5:         return TreeSearch(x.left, k)
6:     else
7:         return TreeSearch(x.right, k)
8:     end if
9: end procedure
```

Asymptotyka: złożoność algorytmu jest zależna jedynie od wysokości drzewa, a więc:

$$T(n) = O(h)$$

- **TreeMin, TreeMax** operacje proste idące jedynie w lewo lub w prawo, odpowiednio. Na przykład Algorytm działa iteracyjnie, przeszukując lewe poddrzewo.
  Złożoność algorytmu to $\Theta(h)$, analogicznie jak w przypadku `TreeMax`.

- **TreeSuccessor** zwraca następnika węzła x w drzewie BST. Algorytm działa rekurencyjnie, porównując klucz węzła z kluczem x. Jeśli klucz jest mniejszy, algorytm przeszukuje lewe poddrzewo, jeśli większy – prawe poddrzewo.
  Złożoność algorytmu to $\Theta(h)$.

- **BST-Inset**: operacja do dodawania elementu do drzewa BST. Algorytm działa rekurencyjnie, porównując klucz nowego elementu z kluczem węzła. Jeśli klucz jest mniejszy, algorytm przeszukuje lewe poddrzewo, jeśli większy – prawe poddrzewo.

Algorithm 11 TreeMin

```
1: procedure TreeMin(x)
2:     while x.left ≠ null do
3:         x = x.left
4:     end while
5:     return x
6: end procedure
```

Algorithm 12 TreeSuccessor

```
1: procedure TreeSuccessor(x)
2:     if x.right ≠ null then
3:         return TreeMin(x.right)
4:     else
5:         y = x.parent
6:         while y ≠ null and x = y.right do
7:             x = y
8:             y = y.parent
9:         end while
10:        return y
11:    end if
12: end procedure
```

Algorithm 13 BST-Insert

```
1: procedure BST-Insert(T, z)
2:     y = null
3:     x = T.root
4:     while x ≠ null do
5:         y = x
6:         if z.key < x.key then
7:             x = x.left
8:         else
9:             x = x.right
10:        end if
11:    end whilez.parent = y
12:    if y = null then T.root = z
13:    else if z.key < y.key then y.left = z
14:    elsey.right = z
15:    end ifz.left = null z.right = null
16: end procedure
```

- **BST-Delete**: operacja do usuwania elementu z drzewa BST. Algorytm działa rekurencyjnie, porównując klucz usuwanego elementu z kluczem węzła. Jeśli klucz jest mniejszy, algorytm przeszukuje lewe poddrzewo, jeśli większy – prawe poddrzewo.

  1. x jest liściem $\rightarrow$ zwolnij pamięć zajmowaną przez x, ustaw wskaźnik jego ojca na null
  2. x ma jedno poddrzewo $\rightarrow$, czyli ma jednego syna v, to zamienić pamięć z x na v, ustawić wskaźnik ojca x na v, a wskaźnik ojca $v.p$ na $x.p$
  3. x ma dwa poddrzewa $\rightarrow$ znajdź następnika x – y za pomocą `TreeSuccessor`, zamień pamięć z x na y, a następnie usuń y z drzewa.

Algorithm 14 BST-Delete

```
1: procedure BST-DELETE(T, z)
2:     if z.left = null then y = z.right
3:     else if z.right = null then y = z.left
4:     elsey = TreeSuccessor(z) z.key = y.key y = y.right
5:     end ify.parent = z.parent
6:     if z.parent = null then T.root = y
7:     else if z = z.parent.left then z.parent.left = y
8:     elsez.parent.right = y
9:     end if
10: end procedure
```

Złożoność algorytmu to $\Theta(h)$.

- **BST-Sort**: operacja do sortowania elementów w drzewie BST. Algorytm działa rekurencyjnie, odwiedzając najpierw lewe poddrzewo, następnie węzeł, a na końcu prawe poddrzewo.

Algorithm 15 BST-Sort

```
1: procedure BST-SORT(T)
2:     InorderTreeWalk(T.root)
3: end procedure
```

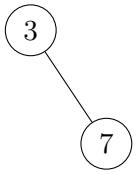
Złożoność algorytmu to $\Theta(n)$.

Na przykładzie wygląda to następująco:

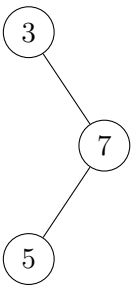
Teraz przedstawiamy drzewo BST krok po kroku przy wstawianiu elementów z tablicy:

Krok 1: Wstawienie 3

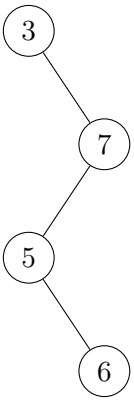
Krok 2: Wstawienie 7



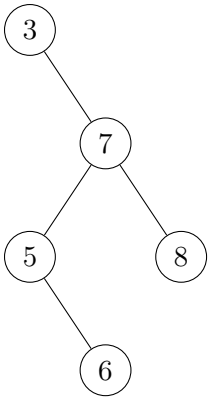
Krok 3: Wstawienie 5



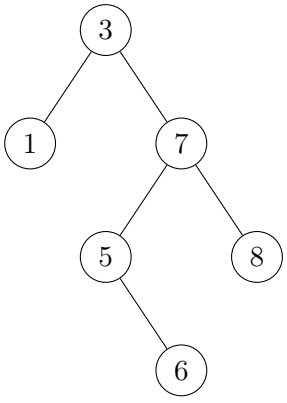
Krok 4: Wstawienie 6



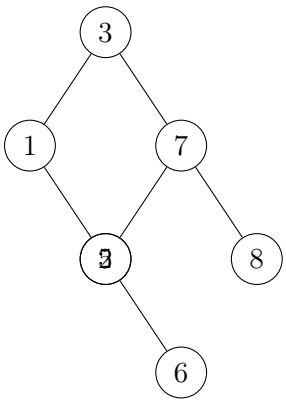
Krok 5: Wstawienie 8



Krok 6: Wstawienie 1



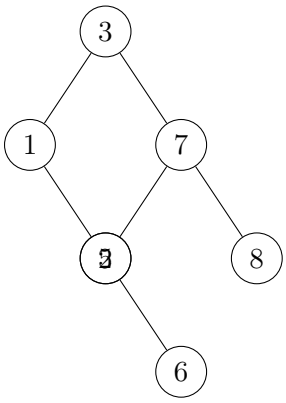
Krok 7: Wstawienie 2



W powyższych diagramach:

- **Krok 1:** Początkowy węzeł, w którym wstawiono 3.
- **Krok 2:** 7 zostaje wstawione jako prawy syn węzła 3.
- **Krok 3:** 5 wstawiono jako lewy syn węzła 7.
- **Krok 4:** 6 wstawiono jako prawy syn węzła 5.
- **Krok 5:** 8 zostaje dodane jako prawy syn węzła 7.
- **Krok 6:** 1 wstawione jako lewy syn węzła 3.
- **Krok 7:** 2 wstawiono jako prawy syn węzła 1.

Tak powstaje ostatecznie drzewo BST:



Porównując algorytm `BST-Sort` z `QuickSort` można zauważyć, że

$$\mathbb{E}\left[\text{Time(BST-Sort)}\right] = \mathbb{E}\left[\text{Time(QuickSort)}\right] = \Theta(n \log n)$$

Gdzie:

$$\text{Time(BST-Sort)} = \sum_{x \in T} \text{depth}(x)$$

A więc:

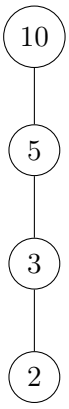
$$\mathbb{E}\left[\sum_{x \in T} \text{depth}(x)\right] = \sum_{x \in T} \mathbb{E}\left[\text{depth}(x)\right] = \Theta(n \log n)$$

Dzieląc obustronnie przez n otrzymujemy:

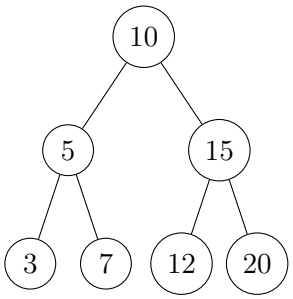
$$\mathbb{E}\underbrace{\left[\frac{1}{n}\sum_{x \in T} \text{depth}(x)\right]}_{\substack{\text{średnia głębokość} \\ \text{węzła w losowym} \\ \text{drzewie BST}}} = \Theta(\log n)$$

8.1.3 Wysokość drzewa BST

- **Worst Case**: W najgorszym przypadku każdy węzeł drzewa ma tylko jedno dziecko, co prowadzi do powstania struktury liniowej (podobnej do listy). W takim przypadku wysokość drzewa wynosi $h = O(n)$. Przykład:



- **Best Case**: Mówimy, że drzewo jest zbalansowane jeśli jego wysokość jest rzędu $O(\log n)$. W takim przypadku każdy węzeł ma co najwyżej dwóch potomków, a drzewo jest zbalansowane. Przykład:



Twierdzenie

Niech T będzie losowym drzewem BST o n węzłach. Wtedy:

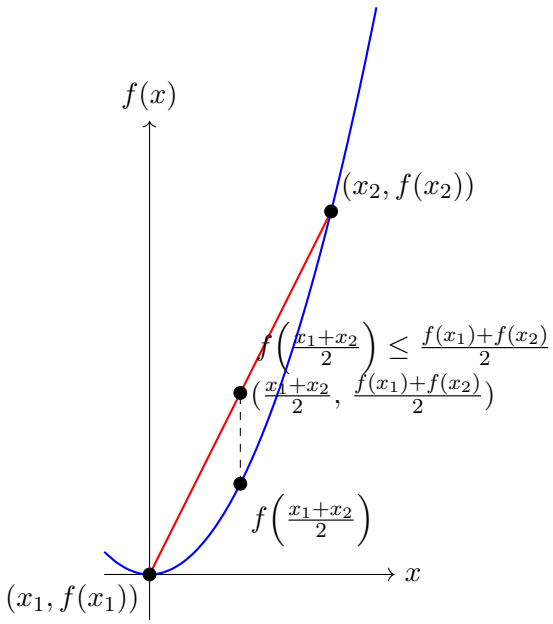
$$\mathbb{E}\left[h(T)\right] \leq 3\log_2 n + o(\log n)$$

Dowód. 1. **Nierówność Jensena**: Niech f-wypukła funkcja, wtedy

$$f\left(\mathbb{E}\left[X\right]\right) \leq \mathbb{E}\left[f(X)\right]$$

2. Zamiast analizować zmienną losową $h(T)$ (oznaczmy H_n jako wysokość drzewa BST o n węzłach) będziemy się zajmować zmienną $Y_n = 2^{H_n}$.

3. Pokażemy, że $\mathbb{E}[Y_n] = O(n^3)$



Warunkując, że korzeń r tworzy $(k-1, n-k)$-split mamy

$$H_n = 1 + \max\{H_{k-1}, H_{n-k}\}$$

Zatem

$$Y_n = 2\max\{Y_{k-1}, Y_{n-k}\}$$

Niech Z będzie zmienną indykatorową, zdefiniowaną w następujący sposób:

$$Z_{n,k} = \begin{cases} 1, & \text{jeżeli } r \text{ wykonuje } (k-1, n-k)\text{-split} \\ 0, & \text{w przeciwnym przypadku} \end{cases}$$

Wtedy

$$Y_n = \sum_{k=1}^{n} Z_{n,k} \cdot 2\max\{Y_{k-1}, Y_{n-k}\}$$

Nakładając wartość oczekiwaną na obie strony, mamy:

$$\mathbb{E}[Y_n] = \mathbb{E}\left[\sum_{k=1}^{n} Z_{n,k} \cdot 2\max\{Y_{k-1}, Y_{n-k}\}\right]$$

Zmienne $Z_{n,k}$ oraz $max\{Y_{k-1}, Y_{n-k}\}$ są niezależne na podstawie podobnego argumentu, jak przy QuickSorcie, a więc

$$\mathbb{E}[Y_n] = 2\sum_{k=1}^{n} \mathbb{E}[Z_{n,k}] \cdot \mathbb{E}\left[\max\{Y_{k-1}, Y_{n-k}\}\right]$$

Rozważmy teraz wartość oczekiwaną $Z_{n,k}$:

$$\mathbb{E}[Z_{n,k}] = 1 \cdot P(Z_{n,k} = 1) + 0 \cdot P(Z_{n,k} = 0) = P(Z_{n,k} = 1) = \frac{(n-1)!}{n!} = \frac{1}{n}$$

A więc

$$\mathbb{E}[Y_n] = \frac{2}{n}\sum_{k=1}^{n} \mathbb{E}\left[\max\{Y_{k-1}, Y_{n-k}\}\right]$$

Można to ograniczyć z góry przez (tracimy zdecydowanie mniej, rzędu stałej, jeżeli robimy to ograniczenie na Y_n, nie na H_n):

$$\mathbb{E}[Y_n] \leq \frac{2}{n}\sum_{k=1}^{n} (\mathbb{E}[Y_{k-1}] + \mathbb{E}[Y_{n-k}]) =$$

$$= \frac{2}{n}\sum_{k=1}^{n}\mathbb{E}[Y_{k-1}] + \frac{2}{n}\sum_{k=1}^{n}\mathbb{E}[Y_{n-k}]$$

Zauwazmy, że są to te same sumy liczone w przeciwnej kolejności:

$$\mathbb{E}[Y_n] \leq \frac{4}{n}\sum_{k=0}^{n-1}\mathbb{E}[Y_k]$$

Przyjmijmy oznaczenie $y_n = \mathbb{E}[Y_n]$.

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

Udowodnimy powyższe przy użyciu indukcji:

- Base Case: $y_0 = y_1 = 0$
- założenie indukcyjne:
$$\forall k < n : y_k \leq cn^3$$
- krok indukcyjny:
$$y_n \leq \frac{4}{n}\sum_{k=0}^{n-1} y_k \leq \frac{4}{n}\sum_{k=0}^{n-1} ck^3 = \frac{4c}{n}\sum_{k=0}^{n-1} k^3$$

  Ograniczmy to z góry przez całkę:

$$y_n \leq \frac{4c}{n}\int_0^n x^3 dx = \frac{4c}{n}\cdot\frac{n^4}{4} = cn^3$$

Zatem z definicji mamy:

$$y_n = O(n^3)$$

A więc cofając nasze oznaczenie:

$$\mathbb{E}[Y_n] = O(n^3)$$

4. Zauważmy, że
$$2^{\mathbb{E}[H_n]} \leq \mathbb{E}[2^{H_n}] = \mathbb{E}[Y_n] = O(n^3)$$

   Jeżeli weźmiemy logarytm z obu stron to:

$$E[H_n] = 3\log_2 n + o(\log n)$$

□

Dokładny wynik pokazany przez Luc Devroye w 1986 roku:

$$\mathbb{E}[H_n] = 2.9882\log_2 n$$

Czyli ograniczając maksimum z góry przy użyciu sumy tracimy nie wiele.

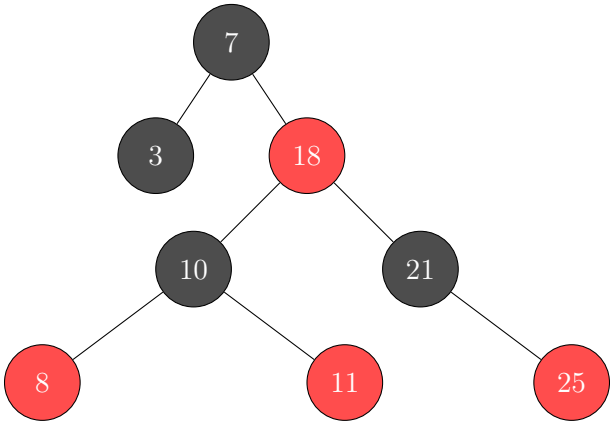
9 Red-Black Trees (RB)

Na dzisiejszym wykładzie zaczynamy temat drzew czerwono-czarnych. Drzewa czerwono-czarne są to zbalansowane drzewa BST, które zapewniają, że wysokość drzewa jest ograniczona przez $2 \cdot \log_2 n$. Dzięki temu operacje na drzewach czerwono-czarnych mają złożoność $\Theta(\log n)$.

9.1 Własności

- BST
- Każdy węzeł ma kolor czerwony lub czarny
- czerwony węzeł nie może mieć czerwonego rodzica
- $\forall x$ każda prosta ścieżka od węzła x ma tyle samo czarnych węzłów: (blackHeight(x), redHeight(x))

Przykład:

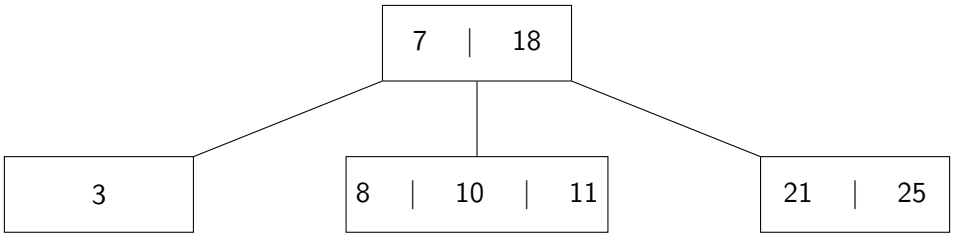


Lemat

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

$$\text{hight}(T) = 2\log_2(n+1)$$

Dowód. Przeprowadzimy dowód w ten sposób. Niech T–dowolne drzewo czerwono-czarne. Czarne węzły "wchłaniają" czerwone węzły, rozważmy to na przykładzie drzewa:



Można zaobserwować, że nie zmienia się liczba liści drzewa. Takie drzewo nazywamy *2-3-4 Tree* jest ona równa $n+1$ Rozważmy wyskość tych drzew. Niech h–wysokość RBT, h'–wysokość 2-3-4 Tree. Wtedy

$$h' = 2h$$

$$2^{h'} \leq n+1 \leq 4^{h'}$$

biorąc logarytm:

$$h' \leq \log_2(n+1) \leq 2h'$$

Zbierając wszystko razem:

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

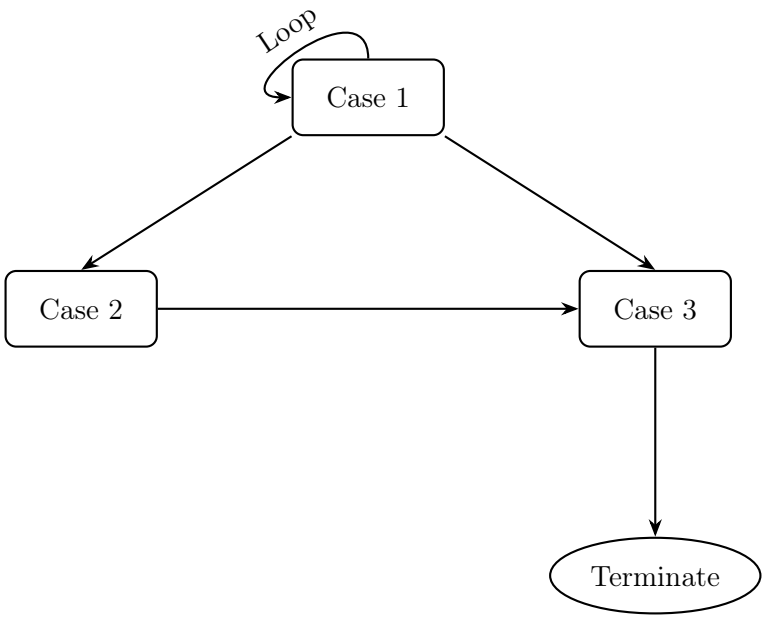
□

9.2 Operacje

9.2.1 RB-Insert

1. Wstaw element z do drzewa jako do BST
2. ustaw kolor wstawianego noda na czerwony
3. FixUp
   (a) z–czerwony, $z.parent$–czarny:
      i. recolor
   (b) z–czerwony, $z.parent$–czerwony, wujek–czarny, czyli tak zwany *zig-zag*
      i. rotate
      ii. recolor
   (c) z–czerwony, $z.parent$–czerwony, wujek–czarny, ale prosta linia
      i. rotate na x i dziadku
      ii. recolor

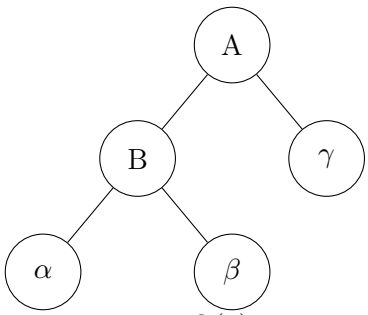
Każdy przypadek naprawy wykonuje się w $\Theta(1)$. Kiedy naprawiamy drzewo przypadki mogą się wykonać następująco



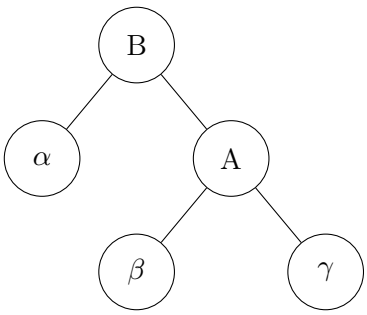
9.2.2 Operacje używane w FixUp

1. recolor → zmień kolor węzła $\Theta(1)$
2. rotate → zobaczmy na przykładzie:

Before Right Rotation



After Right Rotation



Złożoność operacji to $\Theta(1)$

9.2.3 RB-Delete

10 Wykład 2025-04-28

10.1 Direct Access Array

- klucze należą do zbioru $0, \ldots, k-1$, gdzie k–rozmiar tablicy
- klucze będziemy utożsamiać z adresami w pamięci komputera
- operujemy w ramach pamięci WORD-Ram

Jednakże występują pewne ograniczenia:

- adresy są 64-bitowe $\leftrightarrow K \leq 2^{64}$
- każdy obiekt musi mieć unikatowy klucz. Weźmy na przykład baze danych z numerami PESEL.

10.2 Hash Tables

Jest to pewne ulepszenie DAA. Metoda ta polega na przypisaniu klucza do adresu w tablicy. Wykorzystuje się do tego funkcję haszującą, która przekształca klucz na adres w tablicy. Funkcja haszująca powinna być:

$$h : \{0, \ldots, k-1\} \to \{0, \ldots, m-1\}$$

gdzie m–rozmiar tablicy oraz $m \ll k$.
Ponieważ $k \gg m$, to h nie może być różnowartościowa:

$$\exists i \neq j : h(i) = h(j)$$

Chcemy, aby h zwracała wyniki z rozkładu jednostajnego na zbiorze $\{0, \ldots, m-1\}$. Zobaczmy sobie kilka przykładów funkcji haszujących:

- Division hash function[10]
$$h(k) = k \mod m$$
- Counter, Wegmann 1977 Universal hash function:
$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$
gdzie p–liczba pierwsza, a, b–liczby losowe z przedziału $[1, p-1]$, $p > k$.
Zdefinujmy sobie następujący zbiór funkcji haszujacych
$$\mathcal{H}(p, m) = \{h_{a,b} : a, b \in \{0, \ldots, p-1\}\}$$

10.2.1 Universal Hash Property

Twierdzenie

$$\forall k_i \neq k_j \in \{0, \ldots, k-1\} : P_{h \in \mathcal{H}}\left(h(k_i) = h(k_j)\right) \leq \frac{1}{m}$$

Dowód. Zakładamy, że $h_{a,b}$ ma universal hashing property, wtedy

$$h_{a,b}(x) = h_{a,b}(y), x \neq y$$

a więc

$$ax + b \equiv ay + b \mod p$$

[10] jednym z problemów tej funkcji jest to, że nie jest ona jednostajna.

Skoro $p > k, x \neq y \implies x - y \neq 0$ oraz ma odwrotność $\mod p$, zatem

$$a \equiv i \cdot m(x-y)^{-1} \mod p$$

z czego wynika, że występuje $\left\lfloor \frac{p-1}{m} \right\rfloor$ możliwości.

$$P_{h \in \mathcal{H}}\left(h(k_i) = h(k_j)\right) \leq \frac{1}{p-1} \cdot \left\lfloor \frac{p-1}{m} \right\rfloor \leq \frac{1}{m}$$

□

10.2.2 Wartość oczekiwana kolizji

Niech n-liczba haszowanych elementów, $n = \Theta(m)$, zefinujmy zmienną losową

$$L = \#\text{kolizji}$$

Do tego dodajmy zmienną losową indykatorową:

$$X_{i,j} = \begin{cases} 1, & \text{jeżeli } h(k_i) = h(k_j) \\ 0, & \text{w przeciwnym przypadku} \end{cases}$$

wtedy

$$L = \sum_{j \geq 1, i \neq j}^{n} X_{i,j}$$

nakładając wartość oczekiwaną na obie strony:

$$\mathbb{E}[L] = \mathbb{E}\left[\sum_{j \geq 1, i \neq j}^{n} X_{i,j}\right] = \sum_{j \geq 1, i \neq j}^{n} \mathbb{E}[X_{i,j}] = \sum_{j \geq 1, i \neq j}^{n} P_{h \in \mathcal{H}}\left(h(k_i) = h(k_j)\right)$$

Kożystając tutaj z universal hashing property:

$$\mathbb{E}[L] \leq \sum_{j \geq 1, i \neq j}^{n} \frac{1}{m} = \frac{n-1}{m} = \Theta(1)$$

10.3 Wzbogacona struktura danych

Rozważmy przykład Dynamicznej statystyki pozycyjnej z zadanymi funkcjami

- dynamizcne struktury danych
- OS-Select(i) – zwraca i-tą statystykę pozycyjną
- OS-Rank(x) – zwraca statystykę pozycyjną elementu x

Zobrazujmy to na przykładzie RB-Tree: Wzbogacamy taką strukturę danych o pole **size**. Zachowuje on kilka własności:

- $size(x)$–rozmiar poddrzewa x
- $size(x) = size(x.left) + size(x.right) + 1$
- $size(null) = 0$
- $OS - Select(i)$–wykonuje się w $\Theta(\log n)$
- $OS - Rank(x)$–wykonuje się w $\Theta(\log n)$

Algorithm 16 OS-Select

```
1: procedure OS-SELECT(x, i)
2:     k = x.left.size + 1
3:     if i = k then
4:         return x
5:     else if i < k then
6:         return OS-Select(x.left, i)
7:     else
8:         return OS-Select(x.right, i − k)
9:     end if
10: end procedure
```

Algorithm 17 OS-Rank

```
1: procedure OS-RANK(x)
2:     r = x.left.size + 1
3:     y = x
4:     while y ≠ root do
5:         if y = (y.parent).right then
6:             r = r + (y.parent).left.size + 1
7:         end if
8:     end while
9: end procedure
```

10.3.1 Algorytm OS-Select

10.3.2 OS-Rank

Należy zauważyć, że dynamiczne operacje przeprowadzane na strukturze danych takiej jak RB-Tree da się również zmodyfikować w taki sposób, aby jednocześnie zmieniać pole size. Na Przykładzie RB-Insert:

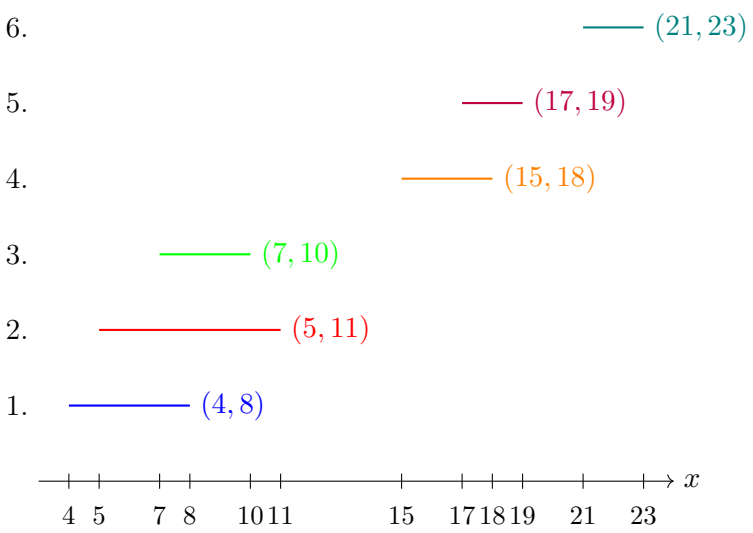
1. BSTinsert – podczas przechodzenia po drzewie zwiększamy size
2. fixup – dalej nie zmieniamy złożonośći poszczególnych funkcji, jedynie zwiększamy stałą w ich złożoności
   - recolor – nie wpływa na size
   - rotate – znowu nie wpływa na size w poddrzewach, jednynie rotowanych nodów, ale da się je zmienić w $\Theta(1)$

10.3.3 Metodologia wzbogacania struktur danych

1. Wybierz strukturę danych, która ma być wzbogacona
2. Wybrać dodatkową informację, która ma być przechowywana w strukturze
3. **Upewnić się**, że dodatkowa informacja nie "pogorszy asymptotycznej złożoności" operacji
4. Zaprojektować dodatkowe operacje na strukturze danych wykorzystując dodatkową informację

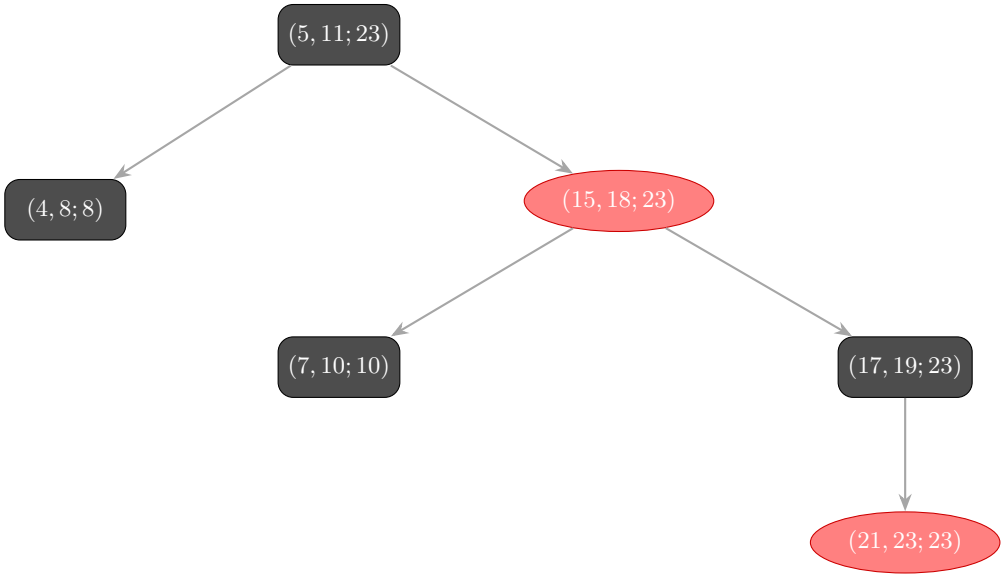
10.3.4 Drzewa Przedziałowe (Interval Trees)

Teraz na innym przykładzie pokażemy metode wzbogacania danych do drzew przedziałowych. Drzewa przedziałowe są to struktury danych, które przechowują zbiory przedziałów.
Rozważmy najpierw przykład. Mamy dane takie odcinki:



1. Wybieramy strukturę danych do przechowywania informacji: **RB-Tree** → kluczem będzie low każdego odcinka
2. dodatkową informacja m

$$x.m = \max \begin{cases} x.high \\ x.left.m \\ x.right.m \end{cases}$$



3. Dodatkowe operacje na drzewie przedziałowym:
   - **Interval-Search** – zwraca wszystkie odcinki, które przecinają się z danym przedziałem $[a, b]$
     Poprawność algorytmu:
     Dla $x \in T$ niech $L = \{j \in x.left\}$ oraz $R = \{j \in x.right\}$.
     Jeśli algorytm "idzie w prawo", wtedy

     $$\{j \in L : j \text{ przecina } i\} = \emptyset.$$

     Jeśli algorytm "idzie w lewo", wtedy:

     $$\{j \in L : i \text{ przecina } j\} \neq \emptyset$$

     lub

     $$\{j \in R : i \text{ przecina } j\} = \emptyset$$

Algorithm 18 Interval-Search

```
1: procedure INTERVAL-SEARCH(i, T)
2:     x = root(T)
3:     while x = null  &  (i.low > x.high | x.low > i.high) do   ▷ if TRUE to i nie przecina się z x
4:         if x.left ≠ null  &  i.low ≤ x.left.m then
5:             x = x.left
6:         else
7:             x = x.right
8:         end if
9:     end while
10:    return x
```

Dowód. załóżmy, że algorytm "idzie w prawo" z x, wtedy

$\to x.left = null \to L = \emptyset$

$\to x.left \neq null, \quad i.low > x.left.m$

$\to \exists j \in L : i.low > j.high (= x.left.m) \implies \{j \in L : i \text{ przecina } j\} = \emptyset$

Załóżmy teraz, że algorytm "idzie w lewo" z x, wtedy

$\to x.right = null \to R = \emptyset$

$\to x.right \neq null, \quad i.low \leq x.right.m$

$\to \exists j \in R : i.low \leq j.high (= x.right.m) \implies \{j \in R : i \text{ przecina } j\} = \emptyset$

□

11 Wykład 2025-05-05

Dzisiaj zastępstwo z MG.

Programowanie Dynaniczne

Do tej pory poznane metody programowania to:

- rekurencyjne
- D&C

Do nich dołączy dzisaj programowanie dynamiczne. Na pierwszy rzut oka wygląda ona podobnie do rekurencji. Zobaczmy to na przykładzie

11.1 Problem znalezienia najdłuższego rosnącego podciągu

- **Input**: $a_1, a_2 \ldots, a_n$–tablica liczb całkowitych
- **Output**: L – ciąg rosnący
- **Problem**: Znaleść L–najdłuższy rosnący podciąg

Rozwiązanie rekurencyjne

Zastnaówmy się jak wyglądałoby rozwiązanie tego problemu w sposób dynamiczny. Niech

$$L(i) = 1 + \max_{1 \leq j \leq i} \{\{L(j) : a_i > a_j\} \cup \{0\}\}$$

$L(i)$–najdłuższy rosnący podciąg kończący się na a_i. Teraz należało by przejść od $i = 1$ do n i policzyć $L(i)$ dla każdego i, zapisać w tablicy i zwrócić $\max_{1 \leq i \leq n}\{L(i)\}$.
Złożoność czasowa tego algorytmu to $O(n^2)$, ponieważ dla każdego i musimy przejść przez wszystkie j i policzyć $L(j)$, natomiast złożoność pamięciowa to $O(n)$.

11.2 Problem wydawania reszty

- **Input**: $c_1 < c_2 < \cdots < c_n$ – zbiór nominałów $\in \mathbb{N}$, R – reszta do wydania
- **Output**: $k \in \mathbb{N}$ – liczba monet do wydania
- **Problem**: Znaleść minimalne k takie, że k monet wystarczy do wydania reszty R

Niech $L(i)$ oznacza minimalną liczbę monet do wydania reszty i.

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

Z warunkami początkowymi $L(0) = 0$. Zobaczmy to na przykładzie, zbiór nominałów to $\{1, 4, 5\}$:

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

Jeżeli trafimy na $L(i)$, którego nie możemy policzyć bo się nie da wydać reszty z danymi nominałami, to zwracamy *inf*.
Złożoność takiego algorytmu to $O(nR)$, natomiast z analizy długości danych wynika, że m–długość w bitach danych to:

$$m = n \cdot \log c_n + \log R \iff n \cdot \log c_n \le \log R$$

Wtedy Złożoność naszego algorytmu jest wykładnicza $O(2^m)$.

Fakt

Jeśli zbiór nominałów zawiera 1, to rozwiązanie istnieje dla każdego $R \in \mathbb{N}$.

11.3 Problem plecakowy – Knapsack

- **Input**: n par waga, wartość (w_i, v_i), ograniczone górnie pezez pojemność plecaka W
- **Output**: $I \subset \{1, \ldots, n\}$ – zbiór przedmiotów zabieranych do plecaka, takie że
  1. $\sum_{i \in I} w_i \le W$
  2. $\sum_{i \in I} v_i$ – maksymalne

Naiwnym algorytmem byłoby przeszukiwane wszystkich podzbiorów zbioru przedmiotów, co daje nam złożoność $O(2^n)$.
Przyjrzyjmy się jednak temu problemowi rekurencyjnie. Niech $V(n, W)$ – maksymalna wartość rozwiązania na n przedmiotach

$$V(i, W) = \max \left\{ \underbrace{V(i-1, W),}_{\text{discard } i^{\text{th}} \text{ item}} \quad \underbrace{V(i-1, W - w_i) + v_i}_{\text{bierzemy } i\text{-ty przedmiot}} \right\}$$

Z pewnymi warunkami początkowymi $V(\cdot, 0) = V(0, \cdot) = 0$. Teraz możemy zacząć budować rozwiązanie rekurencyjne, odrazu rozwiązując rekurencje.
Złożoność tego algorytmu to $O(nW)$, natomiast z podobnego powodu jak poprzednio (analiza długości danych) złożoność algorytmu jest wykładnicza $O(2^m)$, gdzie m to długość danych.

Algorithm 19 Knapsack

```
1: procedure KNAPSACK(n, W)
2:     if n = 0 ∨ W = 0 then
3:         return 0
4:     end if
5:     for i = 1 to n do
6:         for w = 1 to W do
7:             if w_i ≤ w then
8:                 V(i, w) = max{V(i − 1, w), V(i − 1, w − w_i) + v_i}
9:             else
10:                V(i, w) = V(i − 1, w)
11:            end if
12:        end for
13:    end for
14:    return V(n, W)
15: end procedure=0
```

11.4 Optymalne mnożenie macierzy

- **Input**: $A_1, A_2, \ldots, A_n$, gdzie $A_i : m_{i-1} \times m_i$
- **Output**: $P = \prod_{i=1}^{n} A_i$
- **Problem**: Znaleść optymalny sposób mnożenia macierzy

Niech $c(i, j)$ – optymalny koszt przemnożenia macierzy $A_i, A_{i+1}, \ldots, A_j$. Wtedy

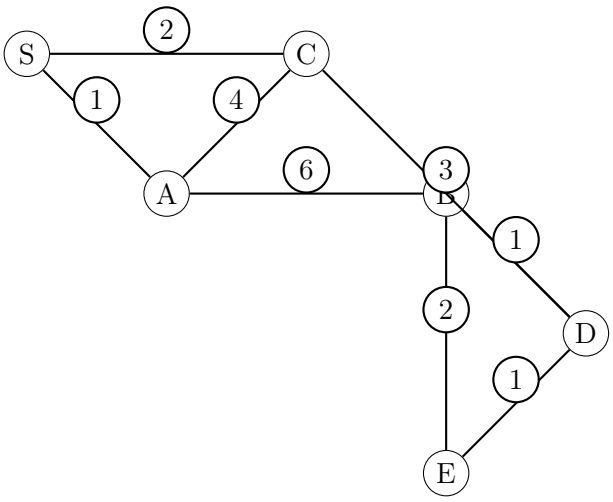
$$c(i, j) = \min_{i \leq k < j} \{c(i, k) + c(k + 1, j) + m_{i-1} m_k m_j\}$$

Z warunkami początkowymi $c(i, i) = 0$.

12 Wykład 2025-05-12

Dzisiaj zaczcynamy używać grafów. Grafy są to struktury danych, które składają się z węzłów i krawędzi. Węzły są to obiekty, które mogą być połączone ze sobą krawędziami. Krawędzie mogą być skierowane lub nieskierowane.

1. Najkrótsza scieżka w DAG



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

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

Natomiast w pseudokodzie:

Algorithm 20 Najkrótsza ścieżka w DAG

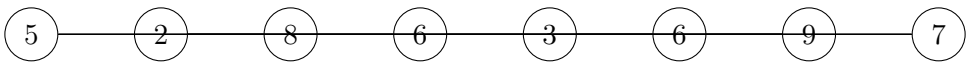
```
1: procedure ShortestPath(G, E)
                                          ▷ Inicjalizacja: Θ(|V|)
2:     for v ∈ V do
3:         L(v) ← ∞
4:     end for
5:     L(S) ← 0
                                          ▷ Meat algorytmu: Θ(|E|)
6:     for v ∈ V \ {S} do
7:         L(v) ← min_{(u,v)∈E}{L(u) + w(u, v)}
8:     end for

9:     return L
10: end procedure
```

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

Rozważmy problem znajdowania nadłuższego rosnącego podciągu. Weźmy na przykład ciag:

$$5, 2, 8, 6, 3, 6, 9, 7$$

W reprezentacji graficznej wygląda to następująco:



12.1 Edit Distance

Przykładem wykożystania tego problemu jest użycie go w spellcheckerach, które sugerują poprawne słowa na podstawie podobieństwa do błędnie napisanego słowa.

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

Zobaczmy to najpierw na **przykładzie**:

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

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

Stąd

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

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

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

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

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

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

A jak wygląda graf?

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

Pseudokod:

13 Wykład 2025-05-19

13.1 Kopiec binarny

Kopiec binarny jest to struktura danych, która jest zbudowana na zasadzie drzewa binarnego. Jest to drzewo, w którym każdy węzeł ma co najwyżej dwóch potomków. Kopiec binarny jest zbudowany na zasadzie porządku, czyli każdy węzeł ma wartość większą lub równą wartości swoich potomków. Kopiec binarny jest używany do implementacji kolejki priorytetowej.
Inaczej można powiedzieć, że jest to drzewo binarne przechowywane w tablicy.

Przykład

Kopiec binarny zapisany w tablicy:

$$[16, 14, 10, 8, 7, 3, 2, 4, 1]$$

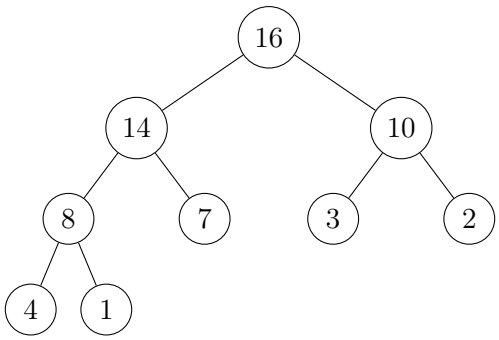
odpowiada następującemu drzewu binarnemu:

Algorithm 21 Edit Distance

procedure EDITDISTANCE(w_1, w_2)
  $n \leftarrow |w_1|$
  $m \leftarrow |w_2|$
  for $i = 0$ to n **do**
    for $j = 0$ to m **do**
      if $i = 0$ **then**
        $d(i, j) = j$
      else if $j = 0$ **then**
        $d(i, j) = i$
      else

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

      end if
    end for
  end for
  return $d(n, m)$
end procedure



W jaki jest algortym przepisania:

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

13.1.1 Własności kopca (maksymalnego)

1. $\forall i : A[parent(i)] > A[i]$ – każdy węzeł jest większy od swoich potomków
2. wysokość węzła = długość najdłuższej prostej ścieżki od tego węzła do liscia

13.1.2 Heapify

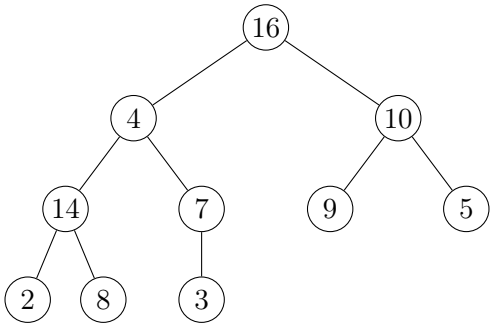
Heapify jest to operacja, która przekształca drzewo binarne w kopiec binarny. Operacja ta jest wykonywana na drzewie binarnym, które nie jest kopcem binarnym. Operacja ta jest wykonywana w czasie $O(n)$.

Przykład

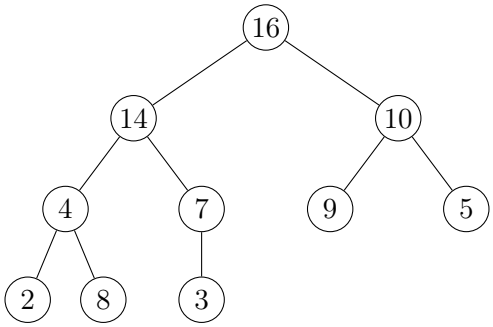
Rozważmy następujący kopiec binarny:

Algorithm 22 Heapify

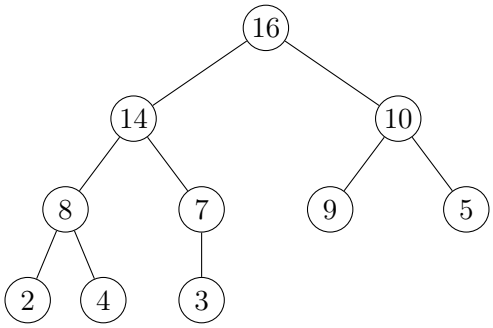
```
1: procedure HEAPIFY(A, i)
2:     l ← left(i)
3:     r ← right(i)
4:     largest ← i
5:     if l < |A| and A[l] > A[largest] then
6:         largest ← l
7:     end if
8:     if r < |A| and A[r] > A[largest] then
9:         largest ← r
10:    end if
11:    if largest ≠ i then
12:        swap(A[i], A[largest])
13:        Heapify(A, largest)
14:    end if
15: end procedure
```



Jeżeli wykonamy operację Heapify na węźle 14, to otrzymamy następujący kopiec binarny:



I ostatecznie



Złożoność obliczeniowa

Algorytm spełnia rekurencję:

$$T(n) = T\left(\frac{2}{3}n\right) + O(1)$$

Z **Master Theorem** otrzymujemy, że złożoność czasowa tego algorytmu to $O(\log n)$, ponieważ $a = 1$, $b = \frac{3}{2}$, $\log_{\frac{3}{2}} 1 = 0$.

Analogicznie wszystko na odwrót dla kopca minimalnego.

13.1.3 Build Heap

Build Heap jest to operacja, która przekształca tablicę w kopiec binarny. Operacja ta jest wykonywana na tablicy, która nie jest kopcem binarnym. Wykożystuje ona wcześniejszy algorytm Heapify.

Algorithm 23 Build Heap

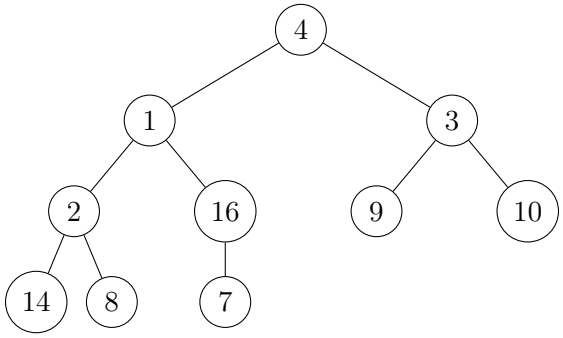
```
1: procedure BuildHeap(A)
2:     for i = ⌊(|A|−1)/2⌋ down to 0 do
3:         Heapify(A, i)
4:     end for
5: end procedure
```

13.1.4 Przykład

ii

Build-Heap na tablicy $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$

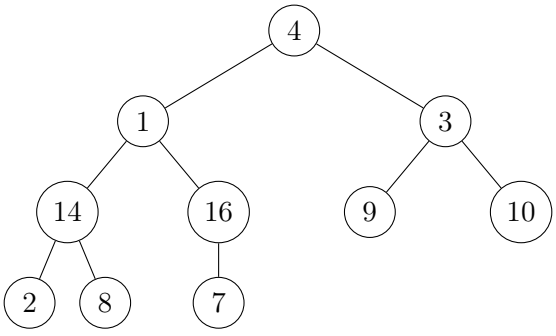
Krok 0: drzewo początkowe



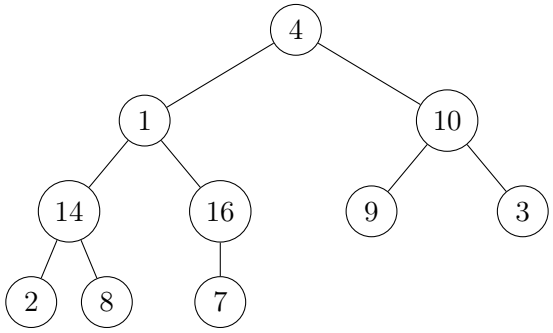
Heapify(@i=5) – brak zmiany

Węzeł o indeksie 5 ma wartość 16, jego jedyny potomek (indeks 10) to 7 – już warunek kopca jest spełniony.

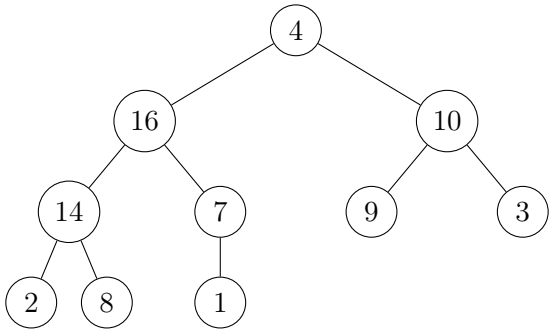
Heapify(@i=4) – zamiana $2 \leftrightarrow 14$



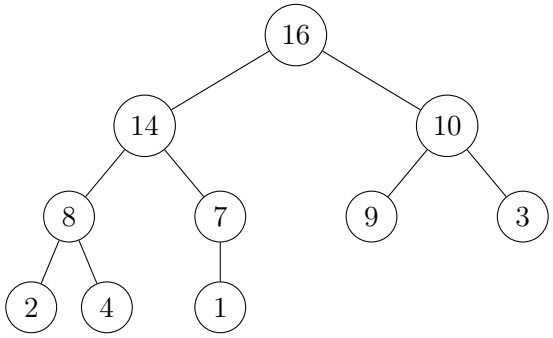
Heapify(@i=3) – zamiana $3 \leftrightarrow 10$



Heapify(@i=2) – zamiana $1 \leftrightarrow 16$**, następnie** $1 \leftrightarrow 7$



Heapify(@i=1) – zamiana $4 \leftrightarrow 16$**, dalej** $4 \leftrightarrow 14$**, dalej** $4 \leftrightarrow 8$



Złożoność obliczeniowa

Niech $|A| = n$ bedzie długością tablicy. musimy wykonać przynajmniej $\frac{n}{2}$ razy operację Heapify, a każda z nich ma złożoność $O(\log n)$. Zatem całkowita złożoność algorytmu to $O(n \log n)$.

Fakt

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

Dowód. jeśli $h = 0$, to są to liście, a zatem jest ich $\lceil \frac{n}{2} \rceil = \frac{n}{2}$.

Założenie indukcyjne:

$$\forall_{k<n} \text{ ilość węzłów na wysokości k } \leq \left\lceil \frac{n}{2^{k+1}} \right\rceil$$

Krok indykcyjny:

węzły o wysokości $h-1$ z zał. ind. jest $\leq \lceil \frac{n}{2^h} \rceil$. Na wysokości h mamy co najwyżej dwa potomki każdego węzła, a zatem liczba węzłów na wysokości h jest $\leq 2 \lceil \frac{n}{2^h} \rceil = \lceil \frac{n}{2^{h+1}} \rceil$. □

13.1.5 Złożoność Obliczeniowa BuildHeap

Algorytm spełnia:

$$T(n) = O\left(\sum_{h=0}^{\log n} (\text{liczba węzłów o wysokości } h) - h\right)$$

Natomiast z Faktu wynika, że

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

13.2 Kolejka priorytetowa

Kolejka priorytetowa jest to struktura danych, która przechowuje elementy w taki sposób, że możemy szybko znaleźć element o najwyższym priorytecie. Kolejka priorytetowa jest implementowana za pomocą kopca binarnego. Operacje, które możemy wykonać na kolejce priorytetowej to:

- $Insert(Q, x)$
- $Maximum(Q)$
- $ExtractMax(Q)$
- $IncreaseKey(Q, x, y)$
- $Delete(Q, x)$
- $Union(Q_1, Q_2)$

Jak będą wyglądały te operacje:

```
1: procedure MAXIMUM(Q)
2:     return Q[1]
3: end procedure
```

```
1: procedure UNION(Q_1, Q_2)
2:     BuildHeap(Q_1 ∪ Q_2)                ▷ Złożoność: O(|Q_1| + |Q_2|)
3: end procedure
```

```
1: procedure DELETE(Q, i)
2:     Q[i] ← Q[size(Q)]                   ▷ Zamiana z ostatnim elementem
3:     size(Q) ← size(Q) − 1               ▷ Zmniejszenie rozmiaru kolejki
4:     if Q[i] > Q[parent(i)] then
5:         Heapify(Q, i)                   ▷ Przywrócenie własności kopca
6:     else
7:         while i > 1 and Q[i] < Q[parent(i)] do
8:             swap(Q[i], Q[parent(i)])    ▷ Przeniesienie węzła w górę
9:             i ← parent(i)
10:        end while
11:    end if
12: end procedure
```

```
1: procedure INSERT(Q, key)
2:     size(Q) ← size(Q) + 1                          ▷ Zwiększenie rozmiaru kolejki
3:     i = size(Q)                                    ▷ Indeks nowego elementu
4:     while i > 1 and key > Q[parent(i)] do
5:         Q[i] ← Q[parent(i)]                        ▷ Przeniesienie węzła w górę
6:         i ← parent(i)
7:     end while
8:     Q[i] ← key                                     ▷ Wstawienie nowego elementu
9:                                                    ▷ Złożoność: O(log n)
10: end procedure
```

```
1: procedure EXTRACTMAX(Q)
2:     if  thensize(Q) < 1
3:         error "Kolejka jest pusta"
4:     else
5:         max ← Q[1]                                 ▷ Zapisanie maksymalnego elementu
6:         Q[1] ← Q[size(Q)]                          ▷ Zamiana z ostatnim elementem
7:         size(Q) ← size(Q) − 1                      ▷ Zmniejszenie rozmiaru kolejki
8:         Heapify(Q, 1)                              ▷ Przywrócenie własności kopca
9:     end if
10:    return max                                     ▷ Zwrócenie maksymalnego elementu
11: end procedure                                     ▷ Złożoność: O(log n)
```

```
1: procedure INCREASEKEY(Q, i, newKey)
2:     if Q[i] > newKey then
3:         Q[i] ← newKey                              ▷ Zmniejszenie klucza
4:         Heapify(Q, i)                              ▷ Przywrócenie własności kopca
5:     else
6:         while i > 1 and newKey > Q[parent(i)] do
7:             Q[i] ← Q[parent(i)]                    ▷ Przeniesienie węzła w górę
8:             i ← parent(i)
9:         end while
10:        Q[i] ← newKey                              ▷ Wstawienie nowego klucza
11:    end if
12: end procedure                                     ▷ Złożoność: O(log n)
```

polecenie	kopiec binarny	kopiec dwumianowy	kopiec Fibonacciego
Insert	$O(\log n)$	$O(\log n)$	$O(1)$ amortyzowane
Maximum	$O(1)$	$O(\log n)$	$O(1)$
ExtractMax	$O(\log n)$	$O(\log n)$	$O(\log n)$ amortyzowane
Decrease/Increase Key	$O(\log n)$	$O(\log n)$	$O(1)$ amortyzowane
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$ amortyzowane
Union	$O(n_1 + n_2)$	$O(\log n)$	$O(1)$ amortyzowane

Tabela 1: Złożoności obliczeniowe operacji na różnych kopcach

13.3 Podsumowanie

14 Wykład 2025-05-20

14.1 Grafy skierowane

Graf skierowany jest to graf, w którym krawędzie mają kierunek. Oznacza to, że krawędź z wierzchołka u do wierzchołka v jest inna niż krawędź z wierzchołka v do wierzchołka u. Graf skierowany jest często używany do modelowania relacji między obiektami, gdzie kierunek relacji ma znaczenie.

Przyjmujemy następujące oznaczenia:

- V – zbiór wierzchołków
- E – zbiór krawędzi
- $|V|$ – liczba wierzchołków
- $|E|$ – liczba krawędzi

Przy czym, jeżeli graf jest skierowany to:

$$E \subseteq \{(i,j) : i,j \in V, i \neq j\}$$

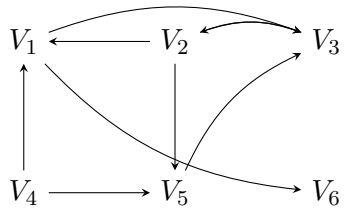
A jeżeli nie skierowany to:

$$E \subseteq \{\{i,j\} : i,j \in V, i \neq j\}$$

Głownie skupimy się na skierowanych grafach.

14.2 Lista sąsiedztwa

Lista sąsiedztwa jest to struktura danych, która przechowuje graf w postaci listy wierzchołków i ich sąsiadów. Jest to efektywny sposób reprezentacji grafu, szczególnie dla grafów rzadkich, gdzie liczba krawędzi jest znacznie mniejsza niż liczba wierzchołków kwadratowa.



Taki DAG jest tożsamy z następującą listą sąsiedztwa:

- $V_1 \to V_3 \to V_6$
- $V_2 \to V_1 \to V_3 \to V_5$
- $V_3 \to V_2$

- $V_4 \to V_1 \to V_5$
- $V_5 \to V_3$
- V_6 (brak krawędzi wychodzących)

Złożoność pamięciowa wynosi $O(n+m) = O(|V| + |E|)$

14.2.1 Macierz sąsiedztwa

Możemy reprezentować sąsiedztwo za pomocą macierzy zdefinowanej w następujący sposób, niech $A = (a_{ij})_{i,j \in [1..n]}$ będzie macierzą sąsiedztwa grafu skierowanego, gdzie:

$$a_{ij} = \begin{cases} 1 & \text{jeśli istnieje krawędź } (i,j) \in E \\ 0 & \text{w przeciwnym wypadku} \end{cases}$$

Złożoność pamięciowa wynosi $O(n^2)$, ponieważ mamy macierz o wymiarach $n \times n$, ale zyskujemy na czasie dostępu do krawędzi, ponieważ możemy sprawdzić istnienie krawędzi w czasie $O(1)$.

14.3 Depth First Search

Rozważmy najpierw składową algorytmu DFS, czyli **explore**:

Algorithm 24 DFS Explore

```
1:  procedure EXPLORE(G, v)
2:      visited(v) = true
3:      previsit(v)                          ▷ operacja przed odwiedzeniem wierzchołka
4:      for each edge (v, u) ∈ E do
5:          if visited(u) = false then
6:              explore(G, u)                ▷ rekurencyjne wywołanie dla sąsiada
7:          end if
8:      end for
9:      postvisit(v)                         ▷ operacja po odwiedzeniu wierzchołka
10: end procedure
11: procedure PREVISIT(v)
12:     pre[v] = clock
13:     clock ← clock + 1
14: end procedure
15: procedure POSTVISIT(v)
16:     post[v] = clock
17:     clock ← clock + 1
18: end procedure
```

Kożystając z tego algorytm DFS wygląda następująco:

Algorithm 25 DFS

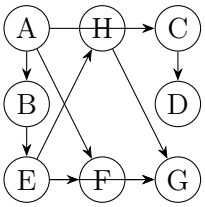
```
1: procedure DFS(G)
2:     visited(v) = false dla każdego v ∈ V
3:     for each vertex v ∈ V do
4:         if visited(v) = false then
5:             explore(G, v)
6:         end if
7:     end for
8: end procedure
```

14.3.1 Złożoność obliczeniowa

Złożoność czasowa algorytmu DFS wynosi $O(n + m)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi. Złożoność pamięciowa wynosi $O(n)$, ponieważ musimy przechowywać informacje o odwiedzonych wierzchołkach.

15 Wykład 2025-05-26

15.1 DFS i porządek topologiczny

Rozważmy graf skierowany 

15.1.1 Własność 1

Dla każdego $\forall v, u \in V$:

1. $pre(v) < pre(u) < post(u) < post(v)$
2. $pre(v) < post(v) < pre(u) < post(u)$

15.1.2 Własność 2

Dla każdej krawędzi $(u, v) \in V$ mamy nazewnictwo:

1. Tree/forward edge: $pre(u) < pre(v) < post(v) < post(u)$
2. Back edge: $pre(v) < pre(u) < post(u) < post(v)$
3. Cross edge: $pre(v) < post(v) < pre(u) < post(u)$

15.1.3 Własność 3

W grafie skierowanym istnieje cykl wtedy i tylko wtedy, gdy DFS wykryje back edge

Dowód. $\implies$
Rozważmy następującą sytłację:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to \cdots \to v_0$$

powiedzmy, że DFS odwiedzi jako pierwszy w tym cyklu wieżchołek v_i. Dalej eksploruje, aż napotka się na v_{i-1}, wtedy krawędź $v_{i-1} \to v_i$ musi być back edge. $\impliedby$

Do własnego zastanowienia się □

15.2 Sortowanie topologiczne DAG'ów

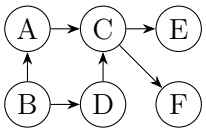
Rozważmy następujący problem:

- **Input**: $G = (V, E)$ – acykliczny graf skierowany
- **Output**: $(V, \prec)$ – porządek topologiczny

Algorytm musi wykonywać następujące operacje:

1. wykonujemy DFS, zapisuje pre i post
2. zwrócić wierzchołki w malejącej kolejności po post

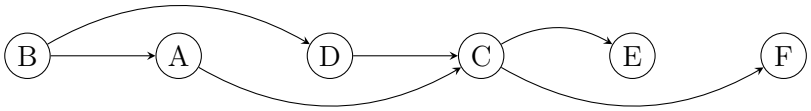
Przykład



Algorytm DFS bedzię odwiedzał odpowiednio wierzchołki w kolejności, gdzie ${}^{pre}X^{post}$

- ${}^{1}F^{2}$
- ${}^{3}C^{6} \rightarrow {}^{4}E^{5}$
- ${}^{7}D^{8}$
- ${}^{9}B^{12} \rightarrow {}^{10}A^{11}$

co wraca następujący porządek topologiczny po post:



15.3 costam

Definicja

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

Definicja

Będziemy mówić, że $v_i, v_2 \in V$ tworzą silnie spójną komponentę w grafie G, jeśli:

- $\forall 1 \leq j \leq k : v_i$ jest połączony z v_j
- nie istnije wieżchołek $u \in V$ taki, że u jest połączony z $v_i \forall 1 \leq i \leq k$

15.3.1 Własność 4

wieżchołek z największą wartością post będzie znajdował się w źródłowym silnie spójnym komponencie grafu skierowanego.

15.3.2 Własność 5

Niech C i C' będą silnie spójnymi składowymi grafu skierowanego, G oraz $\exists (u, v) \in G : u \in C, c \in C'$. Wtedy maksymalna wartość post wierzchołka z C będzie większa niż maksymalna wartość z C'

Dowód. Rozważmy możliwe przypadki

1. DFS najpiwer odwiedzi wierzchołek $u \in C$, przed wierzchołkami z C' (DFS wróci i ustawi wiekszego posta, prosta idea)
2. DFS najpierw odwiedzi wierzchołek $v \in C'$ przed wierzchołkami z C (eksploracja C odbędzie się ostatnia, bo musi do niej wrócić w poźniejszych eksplorach)

□

15.3.3 Własnośc 6

Niech $G^R = (V, E^R)$, gdzie

$$E^R = \{(v, u) : (u, v) \in E\}$$

wtedy źródło metagrafu G^R jest ujściem w metagrafie G

15.4 Strongly Connected Componets algorytm

- **Input**: G – graf skierowany
- **Output**: metagraf silnie spójnych składowych G

16 Wykład 2025-06-02

Na dzisiejszym wykładzie bedziemy zajmować się problemem najkrótszej ścieżki w grafie skierowanym. Używając algorytmu DFS dostajemy kiepskie wyniki, ponieważ DFS nie gwarantuje odwiedzenia wierzchołków w kolejności od najkrótszej ścieżki do najdłuższej. Dlatego będziemy używać algorytmu BFS, który jest bardziej odpowiedni do tego typu problemów.

Porównanie DFS i BFS w kontekcie najkrótszej ścieżki:

16.1 Przeszukiwanie w szerz (Breath First Search)

- **Input**: $G = (V, E)$, G–grafm $s \in V$ – wierzchołek startowy
- **Output**: v do którego istnieje ścieżka z s oraz $dist(v)$ – odległość od s do v

W pseudokodzie:

Algorithm 26 BFS

```
1:  procedure BFS(G, s)
2:      for each vertex v ∈ V do                          ▷ O(|V|)
3:          dist(v) = ∞
4:      end for
5:      dist(s) = 0
6:      Q = [s]                                           ▷ kolejka FIFO
7:      while Q nie jest pusta do
8:          v = Q.pop()
9:          for each edge (v, u) ∈ E do                   ▷ koniec konców O(|E|)
10:             if dist(u) = ∞ then
11:                 dist(u) = dist(v) + 1
12:                 Q.push(u)
13:             end if
14:         end for
15:     end while
16: end procedure
```

Dowód Poprawności

Dowód. **Założenie indukcyjne:**

1. $\forall k \leq d : dist(v)$, który jest równy k będzie poprawnie ustawiony
2. kolejka Q zawiera tylko elementy oddalone o najkrótszą ścieżkę od s do v

Krok indukcyjny:

1. Załóżmy, że $dist(v) = k$ jest poprawnie ustawiony, wtedy $dist(u) = k + 1$ będzie poprawnie ustawiony, ponieważ u jest sąsiadem v i nie był jeszcze odwiedzony.
2. Załóżmy, że kolejka Q zawiera elementy oddalone o najkrótszą ścieżkę od s do v, wtedy po dodaniu nowego elementu u, który jest sąsiadem v, kolejka będzie zawierała elementy oddalone o najkrótszą ścieżkę od s do u.

□

Złożoność obliczeniowa

Złożoność czasowa algorytmu BFS wynosi $O(n+m)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi. Złożoność pamięciowa wynosi $O(n)$, ponieważ musimy przechowywać informacje o odległościach i kolejce.

16.2 Algorytm Dijkstry

Kiedy [11] w grafie wysępuje waga krawędzi pojawia sie problem przy wyznaczaniu najkrótszej ścieżki. Wtedy algorytm BFS nie jest wystarczający, ponieważ nie uwzględnia wag krawędzi. Dlatego używamy algorytmu Dijkstry, który jest bardziej odpowiedni do tego typu problemów.

Pierwszym intuicyjnym pomysłem byłoby dodanie *dummy* wierzchołków do grafu, które będą reprezentować wagi krawędzi, to znaczy jeżeli waga krawędzi między A i B to 4 to dodajemy 3 wierzchołki, które będą reprezentować krawędzie $A \to A_1 \to A_2 \to B$, gdzie A_1 i A_2 będą wierzchołkami o wadze 1, nastepnie odpalamy BFS na nowym grafie. Jednakże jest to bardzo nieefektywne, ponieważ wprowadza dodatkowe wierzchołki i krawędzie, co zwiększa złożoność obliczeniową algorytmu, która będzie zależeć od samej wagi krawędzi grafu.

Zastanówmy się nad innym podejściem:
Alarm Clock Algorithm – algorytm zegara alarmowego, który będzie działał na zasadzie podobnej do zegara, gdzie w każdej iteracji będziemy sprawdzać sąsiadów aktualnego wierzchołka i aktualizować odległości do nich.

- ustaw alarm s na 0
- *Repeat until* no more alarms
  powiedzmy, że następny ustawiony alarm zadzwoni w momencie T w wierzchołku v
  - ustaw $dist(v) = T$
  - dla każdego siąsiada v, niech nazywa się u:
    - jeśli u nie ma ustawionego alarmu to

$$alarm(u) = T + w(v,u)$$

    - jeśli $alarm(u) > T + w(v,u)$

$$alarm(u) = T + w(v,u) + alarm(u)$$

Z jeżeli do implementacji alarmów użyjemy kolejki priorytetowej, to rodzi się **algorytm Dijkstry**:

- **Input**: $G = (V, E)$, $s \in V$ – wierzchołek startowy
- **Output**: $dist(v)$ – odległość od s do v

Pseudokod: **Złożoność obliczeniowa**:
Zgodnie z tym co jest napisane w pseudokodzie złożoność obliczeniowa algorytmu Dijkstry wynosi $O(n + m \log n)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi. Złożoność pamięciowa wynosi $O(n)$, ponieważ musimy przechowywać informacje o odległościach i kolejce priorytetowej.

Dijkstra	$O(\|V\|) \times ExtractMin$[12] $+ O(\|E\|) \times DecreaseKey$[13]
Array	$O(\|V\|^2 \log \|V\|)$
Kopiec binarny	$O(\|V\| \log \|V\| + \|E\| \log \|V\|)$
d–arny Kopiec	$O(\frac{\log \|V\|}{\log d} \cdot (\|V\| d + \frac{\log \|E\|}{d}$
Fibbonacci Heap	$O(\|V\| \log \|V\| + \|E\|)$

[11] wojtas ssie pęto bo nie przyszedł na wykład

Algorithm 27 Algorytm Dijkstry

```
1: procedure DIJKSTRA(G, s)
2:     for each vertex v ∈ V do                                   ▷ O(|V|)
3:         dist(v) = ∞
4:         Q.push(v)
5:     end for
6:     Q = PriorityQueue()                                        ▷ kolejka priorytetowa – dist(·)
7:     dist(s) = 0
8:     while Q nie jest pusta do                                  ▷ O(|V|) × ExtractMin
9:         v = Q.pop()                                            ▷ element z najmniejszą odległością
10:        for each edge (v, u) ∈ E do                            ▷ O(|E|)
11:            if dist(u) > dist(v) + w(v, u) then
12:                dist(u) = dist(v) + w(v, u)
13:                Q.update(u)                                    ▷ O(|E|) × DecreaseKey
14:            end if
15:        end for
16:    end while
17: end procedure
```

17 Wykład 2025-06-03

Wracając do algorytmu Dijkstry jeżeli w grafie pojawią się ujemne wagi krawędzi, to algorytm Dijkstry może przestać działać poprawnie. Rozważmy przypadek cyklu w grafie, którego waga jest ujemna. Możemy po nim chodzić w nieskończoność, co spowoduje, że algorytm Dijkstry nigdy nie zakończy się. Dlatego w grafach z ujemnymi wagami krawędzi należy użyć innego algorytmu, takiego jak Bellman-Ford.

17.1 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda jest algorytmem, który znajduje najkrótsze ścieżki z wierzchołka źródłowego do wszystkich innych wierzchołków w grafie skierowanym, nawet jeśli graf zawiera krawędzie o ujemnych wagach. Algorytm ten działa na zasadzie relaksacji krawędzi, czyli stopniowego aktualizowania odległości do wierzchołków.

- **Input**: $G = (V, E), e \in E w_e, w_e \in \mathbb{R}$, bez ujemnych cykli, $s \in V$ – wierzchołek startowy
- **Output**: $\forall v \in V$ do którego da sie dojśc z s mamy wyznaczyć $dist(v)$ – odległość od s do v

Pseudokod:

Dowód poprawności:

Dowód. **Założenie indukcyjne:** $\forall k \leq |V| - 1 : \forall v \in V$ do którego da się dojść z s mamy $dist(v)$ poprawnie ustawione. **Krok indukcyjny:** Załóżmy, że dla k iteracji mamy poprawnie ustawione odległości, wtedy w $k + 1$ iteracji sprawdzamy każdą krawędź i aktualizujemy odległości do wierzchołków, które sąsiadują z wierzchołkiem u, który ma poprawnie ustawioną odległość. □

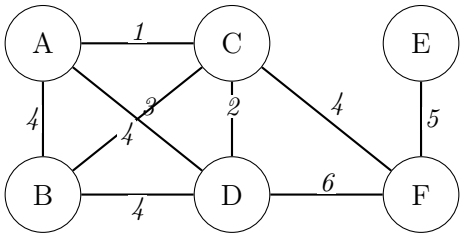
Złożoność obliczeniowa:
Złożoność obliczeniowa algorytmu Bellmana-Forda wynosi $O(nm)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi. Złożoność pamięciowa wynosi $O(n)$, ponieważ musimy przechowywać informacje o odległościach i poprzednikach wierzchołków.

17.2 Algorytmy zachłanne

Problem: Minimalne drzewo rozpinające/Minimum Spanning Tree (MST)

Algorithm 28 Algorytm Bellmana-Forda

```
1:  procedure BELLMAN-FORD(G, s)
2:      for each vertex v ∈ V do                                  ▷ O(|V|)
3:          dist(v) = ∞
4:          prev(v) = null
5:      end for
6:      dist(s) = 0
7:      for i = 1 to |V| − 1 do                                   ▷ O(|V|)
8:          for each edge (u, v) ∈ E do                           ▷ O(|E|)
9:              if dist(v) > dist(u) + w(u, v) then
10:                 dist(v) = dist(u) + w(u, v)
11:                 prev(v) = u
12:             end if
13:         end for
14:     end for
15: end procedure
```



- **Input**: $G = (V, E), e \in E, w_e \in \mathbb{R}$ – graf nieskierowany z wagami krawędzi
- **Output**: $T = (V, \tilde{E})$ – drzewo rozpinające o minimalnej wadze, takie, że $\tilde{E} \subseteq E$, $weight(T) = \sum_{e \in \tilde{E}} w_e$ jest minimalne

Definijca:

Drzewo ≡ acykliczny spójny graf nieskierowany

Wlasności:

- usunięcie krawędzi należącej do cylku nie rospójni grafu
- drzewo o n wierzchołkach ma $n - 1$ krawędzi

  Dowód. Załóżmy, że drzewo ma k krawędzi, wtedy ma $k + 1$ wierzchołków. Jeśli usuniemy jedną krawędź, to otrzymamy dwa drzewa, które mają k krawędzi i $k + 2$ wierzchołków. Zatem drzewo o n wierzchołkach ma $n - 1$ krawędzi. □

- Każdy spójny nieskierowany graf $G = (V, E)$ taki, że $|E| = |V| - 1$ jest drzewem[14]

  Dowód. Musimy pokazać acykliczność.
  Załóżmy, że graf G ma cykl i $e \in E$ należy do tego cyklu. Wtedy usunięcie krawędzi e spowoduje, kożystając z wlasności 1. spowoduje otrzymanie nowego grafu

  $$\hat{G} = (V, \hat{E}), \hat{E} = E \setminus \{e\}$$

  Zatem $\hat{G}$ jest drzewem jeśli $\hat{|}E| = |V| - 1$, ale G miał $|E| = |V| - 1$, więc $|\hat{E}| = |V| - 2$, co jest sprzeczne z założeniem, że G ma cykl. □

[14] Równoważna do definicji znajdującej się powyżej

- **Cut property**
  Niech X będzie podzbiorem krawędzi MST grafu $G = (V, E)$.Wybieramy podzbiór wierzchołków $S \subset V$ taki, że każda krawędz z X nie przechodzi między wierzchołkami z S i $V \setminus S$.
  Niech $e \in E$ będzie krawędzią o najmniejszej wadze pomiędzy S i $V \setminus S$. Wtedy $X \cup \{e\}$ należy do MST grafu G.

  Dowód. Niech T będzie MST. Rozważmy dwa przypadki:

  1. Jeśli $e \in T$, to $T \cup \{e\}$ jest drzewem rozpinającym, ponieważ dodanie krawędzi do drzewa nie tworzy cyklu. □
  2. Jeśli $e \notin T$, zmodyfikujmy T poprzez usunięcie krawędzi $\hat{T} = T \setminus \{e'\} \cup \{e\}$. Wtedy $\hat{T}$ jest drzewem ponieważ:
     - jest nieskierowane
     - jest acykliczne
     - z wlasności 1. jest spójne
     - ma $|V| - 1$ krawędzi

     A więc $\hat{T}$ jest drzewem z własności 3.

     $$weight(\hat{T}) = weight(T) - w(e') + w(e)$$

     natomiast z tego, że T jest MST $weight(T) \leq weight(\hat{T})$, oraz z założenia $w(e) \leq w(e')$, więc

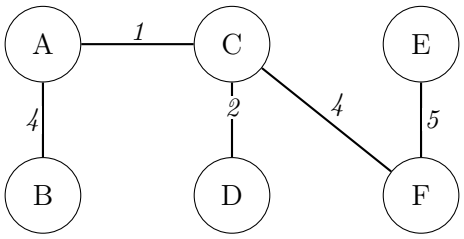
     $$weight(T) = weight(\hat{T})$$

     zatem $\hat{T}$ jest MST, ponieważ ma tę samą wagę co T.

  □

Pierwszy pomysł Rozważmy przykład intuicyjnego algorytmu zachłannego: wybieranie krawędzi o najmniejszej wadze, która nie tworzy cyklu. Jednakże taki algorytm nie gwarantuje znalezienia minimalnego drzewa rozpinającego, ponieważ może wybrać krawędź, która nie jest częścią minimalnego drzewa rozpinającego.

1. Zaczynamy od najmniejszej wadze, wartość $(A, C, 1)$ krawędzi w grafie, dodajemy ją do drzewa.
2. Następnie wybieramy kolejną najmniejszą krawędź, wartość $(C, D, 2)$, dodajemy ją do drzewa.
3. Jedyną krawędzą o wadze 3 jest $(A, D, 3)$, ale nie dodajemy jej, ponieważ tworzy cykl z $(A, C, 1)$ i $(C, D, 2)$.
4. Teraz rozważamy krawędzie o wadze 4
   - $(A, B, 4)$ – nie dodajemy, ponieważ tworzy cykl z $(A, C, 1)$ i $(C, D, 2)$
   - $(B, C, 4)$ – nie dodajemy, ponieważ tworzy cykl z $(A, C, 1)$ i $(C, D, 2)$
   - $(B, D, 4)$ – dodajemy do drzewa
   - $(C, F, 4)$ – nie dodajemy, ponieważ tworzy cykl z $(A, C, 1)$ i $(C, D, 2)$
5. Następnie rozważamy krawędzie o wadze 5. Jest ona jedyna i nie tworzy cyklu, więc dodajemy ją do drzewa.
6. Teraz rozważamy krawędzie o wadze 6. Jest ona jedyna i nie tworzy cyklu, więc dodajemy ją do drzewa.



18 Wykład 2025-06-16

Na dziejszym wykładzie będziemy zajmować się algorytmem Kruskala, który jest jednym z algorytmów do znajdowania minimalnego drzewa rozpinającego w grafie nieskierowanym. Algorytm ten działa na zasadzie sortowania krawędzi według ich wag i dodawania ich do drzewa, jeśli nie tworzą cyklu.

18.1 Algorytm Kruskala

Do implementacji tego algorytmu bedziemy potrzebować pewnej struktury, od której będziemy wymagać pewnych właściwości:
Struktura Danych zbiorów rozłącznych (Disjoint Set Union, DSU)

- $makeset(x)$ – tworzy nowy zbiór zawierający tylko element x.
- $find(x)$ – zwraca rozłączny zbiór do którego należy element x.
- $union(x, y)$ – łączy dwa zbiory rozłączne, do których należą elementy x i y.

Pseudokod algorytmu Kruskala:

Algorithm 29 Algorytm Kruskala

```
1:  procedure KRUSKAL(G = (V, E), {w_i : i ∈ {1, ..., n}})
2:      T = ∅                                          ▷ początkowe drzewo rozpinające
3:      DSU = DisjointSetUnion()                       ▷ inicjalizacja struktury DSU
4:      for each vertex v ∈ V do                       ▷ O(|V|)
5:          DSU.makeset(v)
6:      end for
7:      E.sort()                                       ▷ sortowanie krawędzi według wag – O(|E| log |E|)
8:      for each edge (u, v) ∈ E do                    ▷ O(|E|) × find
9:          if DSU.find(u) ≠ DSU.find(v) then          ▷ sprawdzenie czy krawędź tworzy cykl
10:             T = T ∪ (u, v)
11:             DSU.union(u, v)                        ▷ O(|V|) × union
12:         end if
13:     end for
14: end procedure
```

Złożoność algorytmu:

$$O(|V|) \times makeset + O(|E| \log |E|) + O(|E|) \times find + O(|V|) \times union$$

Struktura danych:
Bedzie ona miała następujące pola:

- π – wskaźnik na ojca
- $rank$ – wyskość poddrzewa zaczepionegi w tym węźle
- $nazwa$

Trochę pseudokodów

19 Wykład 2025-06-17

19.1 Min Cut Problem

Zaczynamy dziejszy wykład od opisu problemu Min Cut Problem. Dysponując nie ważonym grafem nieskierowanym $G = (V, E)$, gdzie $|V| = n$ i $|E| = m$, chcemy znaleźć taki podzbiór krawędzi $E' \subseteq E$, że:

Algorithm 30 Struktura danych zbiorów rozłącznych

```
1: procedure MAKESET(x)
2:     π[x] = x                                   ▷ ustawiamy ojca na siebie
3:     rank[x] = 0                                ▷ wysokość poddrzewa to 0
4: end procedure
5: procedure FIND(x)
6:     if π[x] ≠ x then                           ▷ jeśli nie jesteśmy korzeniem
7:         π[x] = find(π[x])                      ▷ idź do ojca i ustaw go jako ojca dla x
8:     end if
9:     return π[x]                                ▷ zwróć ojca
10:    procedure UNION(x, y)
11:        root_x = find(x)
12:        root_y = find(y)
13:        if root_x ≠ root_y then                ▷ jeśli są w różnych zbiorach rozłącznych
14:            if rank[root_x] < rank[root_y] then
15:                π[root_x] = root_y
16:            else if rank[root_x] > rank[root_y] then
17:                π[root_y] = root_x
18:            else
19:                π[root_y] = root_x
20:                rank[root_x] += 1
21:            end if
22:        end if
23:    end procedure
```

- E' jest minimalny, tzn. $|E'|$ jest minimalne
- $G' = (V, E')$ jest nie jest spójny
- *Dodatkowo* jeżeli G jest ważony, to E' jest taki, że suma wag krawędzi w E' jest minimalna.

Do rozwiązania tego problemu możemy użyć algorytmu kruskala. Załóżmym, że mamy dwa podzbiory w grafie S i $V \setminus S$. Między nimi nie występuje żadna krawędź, natomiast w samych subsetach istnieją już zbudowane początki MST. Rozważmy teraz wszytkie krawędzie takie, że

$$A = \{(u, v) \in E : u \in S, v \in V \setminus S\}$$

Krawędzie z zbioru A łączą dwa podzbiory S i $V \setminus S$. Rozważmy teraz prawdopodobieństwo tego, że wybrana krawędź jest tą odpowiadająca *MinCut*.

$$P(A \in MinCut) \geq \frac{1}{n(n-1)}$$

oraz jeżeli powtórzymy $\Theta(n^2)$ razy algorytm Kruskala na tej krawędzi, to z prawdopodobieństwem $n \to \infty \implies ppb \to 1$, znajdziemy *MinCut*.

Złożoność obliczeniowa: Wynosi ona $O(n^2 m \log n)$, ponieważ musimy wykonać algorytm Kruskala $\Theta(n^2)$ razy, a jego złożoność wynosi $O(m \log n)$, gdzie m to liczba krawędzi w grafie.

Lepsze algorytmy wyszukiwania *MinCut* ulepszają złożoność do $O(n^2 \log n)$, ale nie będziemy się nimi zajmować.

Rozważania Zastanówmy się na działaniem algorytmu. Załóżmy, że mamy:

- k komponentów spójnych w grafie (jak S)
- $|MinCut| = C$
- liczba krawędzi które możemy wybrać to conajmneij $\frac{k \cdot C}{2}$ (gdyby istnijała komponęta mająca mniej wychodzacych krawędzi niż C, to była by to wartość *MinCut*)

Wtedy prawdopodobienstwo wyboru krawędzi z $MinCut$ wynosi:

$$P(A \in MinCut) \geq \frac{C}{\frac{k \cdot C}{2}} = \frac{2}{k}$$

A więc

$$P(A \in MinCut) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \dots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}$$

Pseudokod:

Algorithm 31 Algorytm MinCut

```
1: procedure MinCut(G = (V, E))
2:     X = {}                                   ▷ początkowy zbiór wierzchołków
3:     while |X| < |V| − 1 do                   ▷ dopóki nie zostanie dodany ostatni wierzchołek
4:         wybierz S ⊂ V dla, którego w X nie ma krawędzi pomiędzy S i V \ S
5:         znajdź e ∈ E krawędź o najmniejszej wadze, która łączy S i V \ S
6:         X = X ∪ {e}                          ▷ dodaj krawędź do zbioru
7:     end while
8: end procedure=0
```

19.2 Algorytm Prim'a

Algorytm Prim'a jest algorytmem zachłannym, który znajduje minimalne drzewo rozpinające w grafie nieskierowanym. Działa on na zasadzie stopniowego dodawania krawędzi do drzewa, zaczynając od jednego wierzchołka i rozszerzając drzewo o krawędź o najmniejszej wadze, która łączy drzewo z wierzchołkiem spoza drzewa.

Pseudokod algorytmu Prim'a:

Algorithm 32 Algorytm Prim'a

```
1: procedure Prim(G = (V, E), s, (w_i)_{i=1...|E|})   ▷ s - wierzchołek startowy, w_i - wagi krawędzi
2:     for v ∈ V do
3:         for  docost(v) = ∞                          ▷ koszt dojścia do wierzchołka
4:             prev(v) = null                          ▷ poprzednik wierzchołka
5:         end for
6:         cost(s) = 0                                 ▷ koszt dojścia do wierzchołka startowego
7:         Q = PriorityQueue()                         ▷ kolejka priorytetowa na cost
8:         while H is not empty do
9:             v = Q.extractMin()                      ▷ wyciągamy wierzchołek o najmniejszym koszcie
10:            for each edge (v, u) ∈ E do             ▷ przechodzimy po wszystkich krawędziach wychodzących
    z v
11:                if cost(u) > w(v, u) then           ▷ jeśli koszt dojścia do u jest większy niż waga krawędzi
12:                    cost(u) = w(v, u)               ▷ aktualizujemy koszt dojścia do u
13:                    prev(u) = v                     ▷ ustawiamy poprzednika u na v
14:                    Q.update(u)                     ▷ aktualizujemy kolejkę priorytetową
15:                end if
16:            end for
17:        end while
18:
```

19.3 Kodoowanie Huffmana

Kodoanie Huffmana jest algorytmem kompresji danych, który wykorzystuje drzewo binarne do reprezentacji znaków w taki sposób, że znaki o większej częstotliwości występowania mają krótsze kody, a

znaki o mniejszej częstotliwości mają dłuższe kody. Algorytm ten jest oparty na zasadzie budowania drzewa binarnego, gdzie każdy węzeł reprezentuje znak i jego częstotliwość występowania.

Rozważmy sobie przykład kompresji pliku MP3. Musimy napocząktku wykonać kilka kroków

- digitalizacja analogowego dzwięku: próbkowanie 44,1 KHz $\rightarrow$ 44100 próbek/sekundę
- Γ-alfabet na który zmapowujemy próbki: $\Gamma = \{A, B, C, D\}$
- zmapuj elementy alfabetu na ciągi bitowe

Jeżeli mamy dany jakiś ciąg znaków z naszego alfabetu (zdanie), na przykład:

$$ABACCD\ldots$$

możemy policzyć ilość wystąpień każdego znaku w tym ciągu:

znak	ilość wystąpień
A	70 000 000
B	3 000 000
C	20 000 000
D	37 000 000

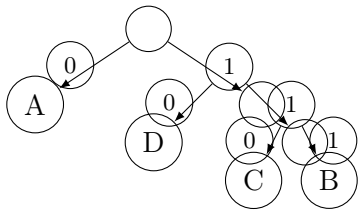
Na podstawie częstotliwości wystąpień znaków możemy stworzyć *prefix-free codes*, które będą za pomocą mniejszej ilości bitów dla najczęściej występujących znaków, a dla rzadziej występujących znaków będą używać większej ilości bitów w jednoznaczy sposób zakodować zdanie. W tym celu możemy użyć algorytmu Huffmana.

$$A \rightarrow 0, D \rightarrow 10, C \rightarrow 110, B \rightarrow 111$$

19.3.1 W jaki sposób tworzyć prefix-free codes?

Prefix-free codes możemy opisać przy pomocy drzewa binarego, gdzie każdy węzeł liściowy reprezentuje znak, a krawędzie reprezentują bity. Idać w drzewie w prawo oznacza dopisanie 0, a w lewo 1. W ten sposób możemy zakodować każdy znak w taki sposób, że żaden kod nie jest prefiksem innego kodu.

W naszym przykładzie drzewo kodowe będzie wyglądać następująco:



20 Ćwiczenia

tu beda pojawialy sie notatki z cwiczen do przedmiotu Algorytmy i struktury danych na Politechnice Wrocławskiej na kierunku Informatyka Algorytmiczna rok 2025 semestr letni.

20.1 Lista 2

robiona na zajęciach 2025-03-10

20.1.1 zadanie 1

Wylicz ile linijek wypisze poniższy program (podaj wynik będacy funkcją od n w postaci asymptotycznej $\Theta(\cdot)$). Można założyć, że n jest potęgą liczby 3. w pseudo kodzie pojawia sie nastepujaca

```
1: function f(n)
2: if n > 1 then
3:     print_line('still going')
4:     f(n/3)
5:     f(n/3)
6: end if=0
```

rekurencja:

$$T(n) = 2T(\frac{n}{3}) + 1$$

rozwiąże ją używając metody podstawienia. Niech $n = 3^k, k = \log_3 n$, wtedy:

$$T(3^k) = 2T(3^{k-1}) + 1$$

Zatem przyjmując $S(k) = T(3^k)$ mamy:

$$S(k) = 2S(k-1) + 1$$

rozwiązując rekurencję otrzymujemy:

$$S(k) = 2^k - 1$$

zatem

$$T(n) = 2^{\log_3 n} - 1 = n^{\log_3 2} - 1 = \Theta(n^{\log_3 2})$$

analogicznie liczmy jaka jest wykonana "praca" wykonana przez program w drzweie rekursji.

20.1.2 zadanie 2

Niech $f(n)$ i $g(n)$ będą funkcjami asymptotycznie nieujemnymi (tzn. nieujemnymi dla dostatecznie dużego n). Korzystając z definicji notacji Θ, udowodnij, że:

$$\max\{f(n), g(n)\} = \Theta(f(n) + g(n)).$$

Dowód. Z definicji notacji Θ mamy:

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

skoro $f(n)$ i $g(n)$ są asymptotycznie nieujemne to:

$$\exists n_f : \forall n \geq n_f, f(n) \geq 0$$

$$\exists n_g : \forall n \geq n_g, g(n) \geq 0$$

zatem

$$n_0 = \max\{n_f, n_g\}$$

a więc

$$f(n) \leq \max\{f(n), g(n)\}$$
$$g(n) \leq \max\{f(n), g(n)\}$$

dodając obie nierówności otrzymujemy:

$$f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$$

zatem

$$\forall n \geq n_0 : \max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$$

a więc z definicji mamy

$$\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$$

□

20.1.3 zadanie 3

Wylicz asymptotyczną złożoność (używając notacji Θ) poniższych fragmentów programów:

Algorithm 33 Pierwszy fragment kodu

```
1: for i = 1 to n do
2:     j = i
3:     while j < n do
4:         sum = P(i, j)
5:         j = j + 1
6:     end while
7: end for
```

Algorithm 34 Drugi fragment kodu

```
1: for i = 1 to n do
2:     j = i
3:     while j < n do
4:         sum = R(i, j)
5:         j = j + j
6:     end while
7: end for
```

Gdzie:

- koszt wykonania procedury $P(i, j)$ wynosi $\Theta(1)$,
- koszt wykonania procedury $R(i, j)$ wynosi $\Theta(j)$.

Dowód.

- Pierwszy fragment kodu
  - Wewnętrzna pętla wykonuje się $n - i$ razy
  - Koszt wykonania procedury $P(i, j)$ wynosi $\Theta(1)$
  - Zatem koszt wykonania wewnętrznej pętli wynosi $\Theta(n - i)$
  - Zatem koszt wykonania całego fragmentu wynosi

$$\sum_{i=1}^{n} \Theta(n - i) = \Theta(n^2)$$

- Drugi fragment kodu

- Wewnętrzna pętla wykonuje się $\log_2 n$ razy
- Koszt wykonania procedury $R(i, j)$ wynosi $\Theta(j)$
- Zatem koszt wykonania wewnętrznej pętli wynosi $\Theta(\log_2 n)$
- Zatem koszt wykonania całego fragmentu wynosi

$$\sum_{i=1}^{n} \Theta(\log_2 n) = \Theta(n \log_2 n)$$

□

Dla pewnosci sprawdzone empirycznie:

20.1.4 zadanie 4

Wyznacz asymptotyczne oszacowanie górne dla następujących rekurencji:

- $T(n) = 2T(n/2) + 1$
- $T(n) = 2T(n/2) + n$
- $T(n) = 3T(n/2) + n \log n$

Kożystając z **Master Theorem** możemy wyznaczyć ograniczenie dla tych rekurencji.

- $T(n) = 2T(n/2) + 1$

  Dowód.

$$a = 2, b = 2, d = 0$$
$$\log_b a = \log_2 2 = 1 > 0 = d$$
$$T(n) = \Theta(n)$$

□

- $T(n) = 2T(n/2) + n$

  Dowód.

$$a = 2, b = 2, d = 1$$
$$\log_b a = \log_2 2 = 1 = d$$
$$T(n) = \Theta(n \log n)$$

□

- $T(n) = 3T(n/2) + n \log n$

  Dowód. Dolne ograniczenie

$$T(n) = 3T(n/2) + n \implies {}^{\text{Master Theorem}} T(n) = \Theta(n^{\log_2 3})$$

  Górne ograniczenie

$$T(n) = 3T(n/2) + n^{1.1} \implies {}^{\text{Master Theorem}} T(n) = \Theta(n^{1.1})$$

□

20.1.5 zadanie 5

Zaprojektuj algorytm wczytujący z wejścia tablicę liczb $A[1], \ldots, A[N]$ i przygotowujący tablicę B tak, że na jej podstawie będzie potrafił odpowiadać na pytania:

1. ile wynosi suma elementów tablicy A od miejsca i do miejsca j włącznie, dla $i < j$.
2. Jaka jest złożoność czasowa Twojego algorytmu? Ile pamięci zajmuje tablica B?
3. Ile zajmuje odpowiedź na jedno pytanie?

Przykładowy algorytm mógłby wyglądać następująco:

Algorithm 35 Algorytm do zadania 5.

```
1: B[1] = A[1]
2: for i = 2 to N do
3:     B[i] = B[i − 1] + A[i]
4: end for
5: procedure SUM(i, j)
6:     if i = 1 then
7:         return B[j]
8:     else
9:         return B[j] − B[i − 1]
10:    end if
11: end procedure
```

Co tu się dzieje?

- W pierwszej pętli obliczamy sumy prefiksowe tablicy A i zapisujemy je w tablicy B.
- W procedurze `Sum` zwracamy różnicę między dwoma elementami tablicy B.
- Złożoność czasowa algorytmu wynosi $\Theta(n)$.
- Tablica B zajmuje $\Theta(n)$ pamięci.
- Odpowiedź na jedno pytanie zajmuje $\Theta(1)$ czasu.

20.1.6 zadanie 6

Pokaż, jak grać w grę w "10 pytań", w której wiadomo, że wybrana liczba jest dodatnia, ale nie jest na początku znane górne ograniczenie jej wartości. Ile pytań potrzebujesz, żeby zgadnąć dowolną liczbę (liczba pytań może zależeć od wielkości liczby)?

W grze "10 pytań"możemy zadać pytania w stylu "czy liczba jest większa od x?". W ten sposób możemy zredukować przestrzeń poszukiwań. W pierwszym pytaniu zapytajmy, czy liczba jest większa od 1. Jeśli tak, to zapytajmy, czy liczba jest większa od 2. W ten sposób możemy zredukować przestrzeń poszukiwań do 2^k dla pewnego k. W ten sposób możemy znaleźć dowolną liczbę w k pytaniach.

20.1.7 zadanie 7

Używając algorytmu **divide-and-conquer** do mnożenia liczb wykonaj mnożenie dwóch liczb binarnych 11011, 1010.

Algorytm **divide-and-conquer** do mnożenia liczb działa w następujący sposób:

Algorithm 36 Algorytm do zadania 6.

```
k = 0
while 2^k < x do
    k = k + 1
end while
p = 2^(k-1)
q = 2^k
procedure BinarySearch(p, q)
```

1. Podziel liczby na dwie równe części.
2. Rekurencyjnie pomnóż te części.
3. Połącz wyniki.

Mnożenie dwóch liczb binarnych 11011 i 1010 możemy zrealizować w następujący sposób:

1. Podziel liczby na dwie równe części: 1101, 1 oraz 10, 10.
2. Rekurencyjnie pomnóż te części: $1101 \cdot 10 = 11010$.
3. Połącz wyniki: $11010 + 110100 = 1000000$.

Algorithm 37 Algorytm do zadania 7 (pokazany na wykładzie)

```
procedure Mul(x, y)
    n = max{|x|, |y|}
    if n = 1 then
        return x · y
    end if
    x_L, x_R = LetfMost ⌈n/2⌉ , RightMost ⌈n/2⌉ bits of x
    y_L, y_R = LetfMost ⌈n/2⌉ , RightMost ⌈n/2⌉ bits of y
    p_1 = Mul(x_L, y_L)
    p_2 = Mul(x_R, y_R)
    p_3 = Mul(x_L + x_R, y_L + y_R)
    return p_1 · 2^(2n) + (p_3 − p_1 − p_2) · 2^n + p_2
```

20.2 Lista 3

robiona na zajęciach 2025-03-24

20.2.1 zadanie 1

Podaj algorytm scalający k posortowanych list tak aby powstała jedna posortowana lista nb (liczba wszystkich elementów na listach to n) działający w czasie $O(n \log k)$.

Algorytm ten można zrealizować w następujący sposób:

20.2.2 zadanie 2

Zdefiniujmy algorytm *k-MergeSort* jako uogólnienie algorytmu sortowania przez scalanie. Różni się od omawianego na wykładzie algorytmu sortowania przez scalanie tym, że dzieli sortowana tablice rekurencyjnie na k równych części (zakładamy, że liczba elementów w tablicy jest potęgą k ($n = k^l$)). Używając wyniku z zadania 1 proszę wykazać dla jakiego k algorytm ma najmniejsza asymptotyczną złożoność obliczeniową liczby porównań (górne ograniczenie $O(\cdot)$).

Algorithm 38 Algorytm do zadania 1.

```
1: procedure MergeLists(L_1, L_2, ..., L_k)
2:     n = sum_{i=1}^{k} |L_i|
3:     B = tablica[1 ... n]
4:     heap = MinHeap
5:     for i = 1 to k do
6:         heap.insert(L_i.pop())
7:     end for
8:     for i = 1 to n do
9:         B[i] = heap.pop()
10:        heap.insert(L_i.pop())
11:    end for
12: end procedure=0
```

Algorytm *k-MergeSort* spełnia rekurencję:

$$T(n) = kT(\frac{n}{k}) + \Theta(n \log k)$$

gdzie $\Theta(n \log k)$ to koszt scalania k posortowanych list (zadanie 1). Z **Master Theorem** otrzymujemy, że algorytm ma złożoność:

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

20.2.3 zadanie 3

Załóżmy że tablica $A = [a_1, \ldots, a_n]$ jest do pewnego momentu k posortowana malejąco i dalej rosnąco (tzn. dla $\forall i < k : a_i > a_{i+1}$ oraz $\forall i \geq k : a_i < a_{i+1}$). Zaprojektuj algorytm znajdujący minimalny element w tablicy A, którego złożoność obliczeniowa będzie wynosić $O(\log n)$. Udowodnij poprawność działania zaproponowanego algorytmu.

- Algorytm ten można zrealizować w następujący sposób:

Algorithm 39 Algorytm do zadania 3.

```
1: procedure FindMin(A, p, q)
2:     if p = q then
3:         return A[p]
4:     end if
5:     m = ⌊(p+q)/2⌋
6:     if A[m] > A[m + 1] then
7:         return FindMin(A, m + 1, q)
8:     else
9:         return FindMin(A, p, m)
10:    end if
11: end procedure
```

- Przykład:
  - $A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$
  - $m = 5$
  - $A[m] = 5,\ A[m + 1] = 4$
  - FindMin$(A, 6, 10)$

- $m = 8$
- $A[m] = 2,\ A[m+1] = 1$
- $\text{FindMin}(A, 9, 10)$
- $m = 9$
- $A[m] = 1,\ A[m+1] = 1$
- **Zwracamy** $A[m] = 1$

- Złożoność obliczeniowa: Algorytm spełnia rekurencję:

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

Z **Master Theorem** otrzymujemy, że algorytm ma złożoność:

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

□

20.2.4 zadanie 4

Zastąpienie użycia QuickSort'a, dla tablic małych rozmiarów, algorytmem InsertionSort jest częstym sposobem na zwiększenie efektywności algorytmu rozwiązującego problem sortowania. Pokaż, że jeśli zmodyfikujesz bazowy przypadek rekurencji w algorytmie QuickSort w taki sposób, że dla tablic o $\leq k$ elementach wywoływany będzie InsertionSort (zamiast rekurencyjnego wywołania QuickSort'a), to wartość oczekiwana liczby porównań będzie wynosić $\Theta\left(nk + n \log \frac{n}{k}\right)$. Jakie k należy wybrać, aby zminimalizować tę złożoność?

Należy rozważyć wartość oczekiwaną zmiennej losowej $\text{compInsertSort}_n \equiv R_n$

$$\mathbb{E}[R_n] = \frac{n(n-1)}{4}$$

Niech T_n – liczba porównan w *Hybrid Sorcie* oraz

$$X_k^n = \begin{cases} 1 \iff \text{partition podzielił tablice na} \mid k \mid n-k \text{split}(k, n-k) \\ 0 \iff \text{w przeciwnym przypadku} \end{cases}$$

bedzie funkcja indykatorową. Wtedy

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

Przyjmimy oznaczenie $\mathbb{E}[T_n] = t_n$, wtedy

$$\begin{aligned} t_n &= \sum_{k=0}^{n-1} \mathbb{E}[X_k] \cdot (t_k + t_{n-k} + n - 1) \\ &= \sum_{k=0}^{n-1} \frac{1}{n} \cdot (t_k + t_{n-k} + n - 1) \\ &= \frac{1}{n} \sum_{k=0}^{n-1} (t_k + t_{n-k} + n - 1) \\ &= \frac{1}{n} \sum_{k=0}^{n-1} t_k + \frac{1}{n} \sum_{k=0}^{n-1} t_{n-k} + \frac{1}{n} \sum_{k=0}^{n-1} n - 1 \\ &= \frac{2}{n} \sum_{k=0}^{n-1} t_k + n - 1 \end{aligned}$$

Mnożąc obie strony równania przez n otrzymujemy

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

Zamieniając $n \to n-1$ otrzymujemy

$$(n-1)t_{n-1} = 2\sum_{k=0}^{n-2} t_k + n^2 - 3n + 2$$

Odejmując stronami otrzymujemy

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

Upraszczając:

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

Dzieląc obie strony przez $n(n+1)$ otrzymujemy

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

teraz przyjmujemy oznaczenie $\varphi_n = \frac{t_n}{n+1}$, wtedy

$$\varphi_n = \varphi_{n-1} + \frac{2(n-1)}{n(n+1)}$$

należy rozwiązać powyższą rekurencję itracyjnie, biorąc pod uwagę, że w pewnym momencie następuje zmiana z algorytmu *QuickSort* na *InsertionSort*, niech k tym momentem, wtedy

$$\begin{aligned}
\varphi_n &= \varphi_{n-1} + \frac{2(n-1)}{n(n+1)} = \\
&= \sum_{i=k+1}^{n} \frac{2(i-1)}{i(i+1)} + \frac{k(k-1)}{4(k+1)} \\
&= \sum_{i=k+1}^{n} \left(\frac{2}{i+1} - \frac{1}{i}\right) + \frac{k(k-1)}{4(k+1)} = \\
&= 2\sum_{i=k+1}^{n} \frac{1}{i+1} - \sum_{i=k+1}^{n} \frac{1}{i} + \frac{k(k-1)}{4(k+1)} = \\
&= 2\left(H_{n+1} - H_k\right) - \left(H_n - H_k\right) + \frac{k(k-1)}{4(k+1)} = \\
&= H_n + \frac{2}{n+1} - H_k - \frac{k(k-1)}{4(k+1)}
\end{aligned}$$

Podstawiając $t_n = (n+1)\varphi_n$ otrzymujemy

$$\begin{aligned}
t_n &= (n+1)\left(H_n + \frac{2}{n+1} - H_k - \frac{k(k-1)}{4(k+1)}\right) = \\
&= (n+1)H_n + 2 - (n+1)H_k - \frac{k(k-1)}{4(k+1)}(n+1)
\end{aligned}$$

Ostatecznie prowadzi to do asymptotycznego wzoru na liczbę porównań

$$\mathbb{E}[T_n] = \Theta\left(nk + n\log\frac{n}{k}\right)$$

□

20.2.5 zadanie 5

Załóżmy, że masz do wyboru jeden z trzech algorytmów rozwiązujących postawiony Ci problem wielkości n:

1. **Algorytm A**: rozwiązuje problem dzieląc go rekurencyjnie na 5 pod-problemów o połowę mniejszych i scalając ich rozwiązania w czasie $\Theta(n \log n)$.
2. **Algorytm B**: rozwiązuje problem dzieląc go rekurencyjnie na 2 pod-problemy rozmiaru $n-1$ i scala ich rozwiązania w czasie stałym.
3. **Algorytm C**: rozwiązuje problem dzieląc go rekurencyjnie na 9 pod-problemów rozmiaru $n/3$ i scalając ich rozwiązania w czasie $\Theta(n^2)$.

Jaka jest złożoność obliczeniowa tych algorytmów? Który z nich byś wybrał? Odpowiedź uzasadnij.

---

1. Spełniona zostaje rekurencja:

   $$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n \log n)$$

   Ograniczmy ją sobie z dołu i z góry:

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

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

   Zatem z **Master Theorem** otrzymujemy, że złożoność obliczeniowa algorytmu A wynosi $\Theta(n \log n)$.

2. Spełniona zostaje rekurencja:

   $$T(n) = 2T(n-1) + \Theta(1)$$

   Ograniczmy ją sobie z dołu i z góry:

   $$T(n) = \Omega(n)$$

   $$T(n) = O(n)$$

   Zatem z **Master Theorem** otrzymujemy, że złożoność obliczeniowa algorytmu B wynosi $\Theta(n)$.

3. Spełniona zostaje rekurencja:

   $$T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2)$$

   Ograniczmy ją sobie z dołu i z góry:

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

   $$T(n) = O(n^2)$$

   Zatem z **Master Theorem** otrzymujemy, że złożoność obliczeniowa algorytmu C wynosi $\Theta(n^2)$.

20.2.6 zadanie 6

Powiedzmy, że masz do wykonania n zadań, gdzie każde z nich wymaga t_j minut pracy. Chcesz wykonać wszystkie zadania maksymalizując zadowolenie przełożonego poprzez minimalizację średniego czasu zakończenia każdego zadania. Uzasadnij, w jakiej kolejności powinieneś wykonywać zadania.

Możemy przyjąć, że mamy tablicę $T = [t_1, t_2, \ldots, t_n]$ z czasami trwania zadań. Możemy zrealizować algorytm w następujący sposób:

- Zadania powinny być wykonywane w kolejności rosnącej czasu pracy. Posortować tablicę T rosnąco i wykonywać zadania pokoleji.

- Dla każdego zadania j zakończenie zadania j w czasie t_j minimalizuje średni czas zakończenia każdego zadania. Dlaczego się tak dzieje?
  Rozpatrzmy najpierw mały przykład:
  Mamy 3 zadania
  $$T = [t_1 = 3, t_2 = 5, t_3 = 4].$$
  Posortowane zadania to
  $$T = [t_1 = 3, t_3 = 4, t_2 = 5].$$
  teraz czasy zakonczenia wykonywania zadań, to
  $$t = [t_1, t_1 + t_3, t_1 + t_3 + t_2]$$
  $$t = [3, 3 + 4, 3 + 4 + 5]$$
  Średni czas zakończenia zadania to
  $$\frac{3 + 7 + 12}{3} = 7\frac{1}{3}$$
  Rozważmy teraz ogólny problem:
  - **Input**: tablica z długościami trwania wykonywania zadań $T = [t_1, t_2, \ldots, t_n]$
  - **Output**: pewna permutacja tablicy T, która minimalizuje średni czas zakończenia zadania – $\sigma(T)$
  - **Problem**: należy znaleść pewne minimum:
    $$\min \{f(\sigma(T)) : \sigma \in S_n\}$$
    gdzie $f : \mathcal{P}(\mathbb{N}) \to \mathbb{R}$ jest funkcją postaci:
    $$f(T) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} t_j$$
    można to zapisać jako:
    $$f(T) = \frac{nt_1 + (n-1)t_2 + \cdots + 2t_{n-1} + t_n}{n}$$
  - **Rozwiązanie**: minimalizacja funkcji f polega na posortowaniu tablicy T rosnąco, nalezy jednak udowodnić, że permutacja σ sortująca tablicę T jest optymalna. **Dowód:**
    Załóżmy, że istnieje optymalna permutacja τ, w której występuje para indeksów $i < j$ taka, że $t_i > t_j$. Rozważmy nową permutację τ', otrzymaną przez zamianę miejscami zadań i oraz j, czyli:
    $$\tau' = (\ldots, t_j, \ldots, t_i, \ldots)$$
    Przyjrzyjmy się wpływowi tej zamiany na funkcję celu
    $$f(T) = \frac{1}{n} \sum_{k=1}^{n} \sum_{l=1}^{k} t_{\tau(l)}.$$
    W wyniku zamiany, zadania t_i i t_j pojawiają się na pozycjach i oraz j, odpowiednio, co wpływa na sumy częściowe w następujący sposób:
    $$\Delta = f(\tau') - f(\tau) = (t_i - t_j)(j - i).$$
    Skoro $j - i > 0$ oraz przy założeniu $t_i > t_j$, mamy
    $$\Delta > 0.$$

Oznacza to, że funkcja celu f jest mniejsza dla permutacji τ' niż dla τ, czyli:

$$f(\tau') < f(\tau).$$

Sprzeczność z założeniem, że τ była optymalna, wynika z faktu, iż zawsze można dokonać zamiany pary, gdzie wcześniejsze zadanie trwa dłużej niż późniejsze, co zmniejsza średni czas zakończenia.

W związku z tym, aby nie istniały żadne takie "inwersje", permutacja optymalna musi spełniać warunek

$$t_{\tau(1)} \leq t_{\tau(2)} \leq \cdots \leq t_{\tau(n)},$$

czyli musi być uporządkowana rosnąco.

Wniosek: Minimalizacja średniego czasu zakończenia zadań jest osiągana przez sortowanie tablicy T w porządku rosnącym.

20.2.7 zadanie 7

Stwórz algorytm znajdujący najczęściej powtarzający się element w n elementowej tablicy (unikając sortowania tablicy), mający złożoność $O(n \log n)$ (zakładamy, że ten element powtarza się ponad $\frac{n}{2}$ razy).

Algorithm 40 Algorytm do zadania 7.

```
1:  procedure DOMINAT(A, p, q)
2:      if p = q then
3:          return A[p]
4:      end if
5:      dom_L = Dominat(A, p, ⌊(p+q)/2⌋)
6:      dom_R = Dominat(A, ⌊(p+q)/2⌋ + 1, q)
7:      if dom_L = dom_R then
8:          return dom_L
9:      end if
10:     count_L = count(A, p, q, dom_L)
11:     count_R = count(A, p, q, dom_R)
12:     if count_L > count_R then
13:         return dom_L
14:     else
15:         return dom_R
16:     end if
17: end procedure
```

Złożoność obliczeniowa algorytmu:

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

Z **Master Theorem** otrzymujemy, że złożoność obliczeniowa algorytmu wynosi $\Theta(n \log n)$.
Jest to problem z kategorii *Majority element*.

Algorytm działający w czasie liniowym:

Algorithm 41 Algorytm do zadania 7. działajcy w czasie liniowym.

```
1: procedure MAJORITY(A)
2:     count = 0
3:     candidate = None
4:     for i = 0 to n − 1 do
5:         if count = 0 then
6:             candidate = A[i]
7:         end if
8:         if A[i] = candidate then
9:             count = count + 1
10:        else
11:            count = count − 1
12:        end if
13:    end for
14:    return candidate
15: end procedure
```

20.2.8 zadanie 8

Doktor Freud ma wahania nastrojów, które zapisuje sobie ilustrując nastrój danego dnia nieujemną liczbą całkowitą. Po n dniach zgromadził tablice n liczb opisujących swój nastrój i postanowił znaleść (spójny) przedział czasu (dni), w których był najszczęśliwszy. Doktor Freud zdefiniował sobie szczęśliwość przedziału czasu jako sumę wartości występujących w dniach tego przedziału przemnożony przez najmniejszą wartość występującą w zadanym przedziale. Stwórz algorytm D&C znajdujący najszczęśliwszy przedział w złożoności $O(n \log n)$

20.2.9 zadanie 9

Wykaż, że nie istnieje algorytm sortujący, który działa w czasie liniowym dla co najmniej połowy z $n!$ możliwych danych wejściowych długości n. Czy odpowiedź ulegnie zmianie jeśli zapytamy o ułamek $\frac{1}{n}$ lub $\frac{1}{n^2}$ wszystkich permutacji?

Do pokazania tego faktu skożystamy z drzewa decyzyjnego, podobnie jak na wykładzie. Rozważmy nierówność gdzie h-wysokość drzewa, l- liści

1. $$\frac{n!}{2} \leq l \leq 2^h \implies 2^{h+1} \geq n! \implies h \geq \log_2 n! - 1$$
   zatem $h = \Omega(n \log n)$
2. $$\frac{n!}{n} \leq 2^h \implies h \geq \log_2 n! - \log_2 n \implies h = \Omega(n \log n)$$
3. $$\frac{n!}{2^n} \leq 2^h \implies h + n \geq \log_2 n! \implies h \geq log_2 n! - n \implies h = \Omega(n \log n)$$

20.2.10 zadanie 10

Zaprojektuj algorytm, który sortuje n liczb całkowitych z przedziału od 1 do n^2 w czasie $O(n)$.

20.3 lista 5

20.3.1 zadanie 6

Zaproponuj strukturę danych $\mathcal{Q}$ dla dynamicznych zbiorów liczb, w której można wykonywać operację Min – Luka wyznaczającą odległość między dwoma najbliższymi sobie liczbami w $\mathcal{Q}$. Jeśli np. $\mathcal{Q} = \{1, 5, 9, 15, 18, 22\}$, to Min – Luka($\mathcal{Q}$) daje w wyniku $18 - -15 = 3$. Zaimplementuj jak najefektywniej operacje Insert, Delete, Search oraz Min – Luka i wykonaj analizę ich złożoności czasowej.

Do implementacji struktury użyjemy wzbogaconych drzew RBT o mdb i mdt, gdzie

$$mdt = \min\{key - Parent.key, S.key - key\}$$

oraz

$$mdb = \min\{mdt, x.left.mdb, x.right.mdb\}$$

20.4 lista 6

20.4.1 zadanie 2

- **INPUT:** Zbiór n odcinków $[s_i, f_i, w_i]$ dla $i = \{1, \ldots, n\}$, gdzie s_i to początek i–tego odcinka, f_i koniec i–tego odcinka, w_i to waga i–tego odcinka.
- **OUTPUT:** Podzbiór odcinków nie przecinających się o największej wadze.

Zaprojektuj efektywny algorytm rozwiązujący postawiony problem. Przeanalizuj poprawność i złożoność obliczeniową swojego algorytmu.

Podejdźmy do problemu w następujący sposób:

- Posortuj odcinki rosnąco względem końca f_i.
- Niech dp będzie n–elementową tablicą o wartościach -1[15], gdzie $dp[1] = w_1$.
- $trace[1] = [T_1]$
- dla każdego i od 2 do n wykonujemy:
  - wyszukujemy największe j takie, że $f_j < s_i$ (można to zrobić binarnie, ponieważ odcinki są posortowane względem końca).
  - $if dp[j] + w_i > dp[i-1]$
    $dp[i] = dp[j] + w_i$
    $trace[i] = insert(trace[j], T_i)$
  - $else$
    $dp[i] = dp[i-1]$
    $trace[i] = trace[i-1]$
  - return $trace[n]$

20.4.2 zadanie 3

- **INNPUT:** Ciąg $[a_1, \ldots, a_n]$.
- **OUTPUT:** Ciąg spójny $a_i, a_{i+1}, \ldots, a_{i+k}$, którego suma będzie największa.

Zaprojektuj liniowy algorytm znajdowania ciągu opisanego w **OUTPUT**, zbadaj jego poprawność. **Przykład:** dla ciągu $5, 15, -30, 10, -5, 40, 10$ ciągu spójny o największej sumie to: $10, -5, 40, 10$.

Pseudo kod programu dynamicznego dla zadania 3:

[15] inicjalizacja całej tablicy na -1

Algorithm 42 Algorytm do zadania 3.

```
1: procedure MaxSubarray(A)
2:     max_sum = A[0]
3:     current_sum = A[0]
4:     start = 0
5:     end = 0
6:     for i = 1 to n − 1 do
7:         if current_sum < 0 then
8:             current_sum = A[i]
9:             start = i
10:        else
11:            current_sum+ = A[i]
12:        end if
13:        if current_sum > max_sum then
14:            max_sum = current_sum
15:            end = i
16:        end if
17:    end for
18:    return A[start : end + 1]
19: end procedure
```

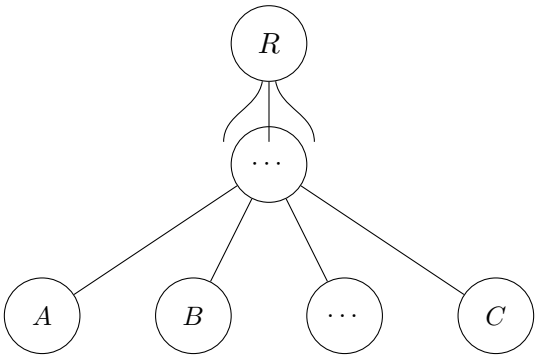
20.4.3 zadanie 4

Jedziemy przez paliwową pustynię pojazdem palącym 1 litr paliwa na 1 km. Pojemność baku wynosi W. Znamy rozkład stacji benzynowych wzdłuż drogi, którą jedziemy i wszystkie one są położone na kilometrach będących liczbami całkowitymi. Na stacji numer i, cena paliwa wynosi w_i. Pokaż algorytm obliczający jak najtaniej można dojechać do końca drogi (czyli do stacji numer n). Przeanalizuj poprawność i złożoność obliczeniową swojego algorytmu.

20.4.4 zadanie 5

Rozważmy ukorzenione drzewo, w którego korzeniu pojawia się pewna informacja. W każdej rundzie, wierzchołek posiadający informację, może poinformować jedno swoje dziecko. Pokaż algorytm, który na podstawie struktury drzewa obliczy dla każdego wierzchołka w jakiej kolejności ma on informować dzieci tak, żeby czas dotarcia informacji do wszystkich wierzchołków drzewa był jak najkrótszy. Przeanalizuj poprawności złożoność obliczeniową swojego algorytmu.

Rozważmy drzewo T jako DAG z wierzchołkiem R jako korzeniem. Jeżeli ustawimy węzły w porządku np. topologicznym, i krawędzie będą wskazywały relacje (parent, child) na przykład:



Niech $T(N)$ bedzie minimalnym czasem propagacji z noda N do jego całego poddrzewa. Zachodzi wtedy taka relacja:

$$T(N) = \max_{M \in N.children} \{T(M) + \text{kolejność wywołania}\}$$

Aby zminimalizować tę wartość to odpalamy M w kolejności malejącej wartości $T(M)$. **Przykład:**