Notatki z Algorytmów i Struktur Danych

Jakub Kogut

13 kwietnia 2025

Spis treści

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) $\leftarrow$ 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

```
procedure MERGESORT(A, 1, n)
    if |A[1..n]| == 1 then
        return A[1..n]
    else
        B = MergeSort(A, 1, ⌊n/2⌋)
        C = MergeSort(A, ⌊n/2⌋, n)
        return Merge(B, C)
    end if
end procedure
```

Algorithm 2 Merge

```
procedure MERGE(X[1..k], Y[1..n])
    if X = ∅ then
        return Y
    else if Y = ∅ then
        return X
    else if X[1] ≤ Y[1] then
        return [X[1]] × Merge(X[2..k], Y[1..n])
    else
        return [Y[1]] × Merge(X[1..k], Y[2..n])
    end if
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 $T(n) = O(n^2)$ $\square$

  – 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

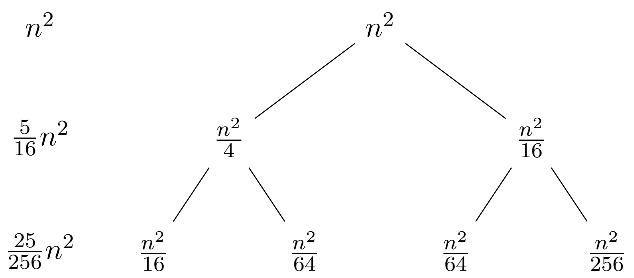
$$T(n) = \log n \log \log n$$

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:

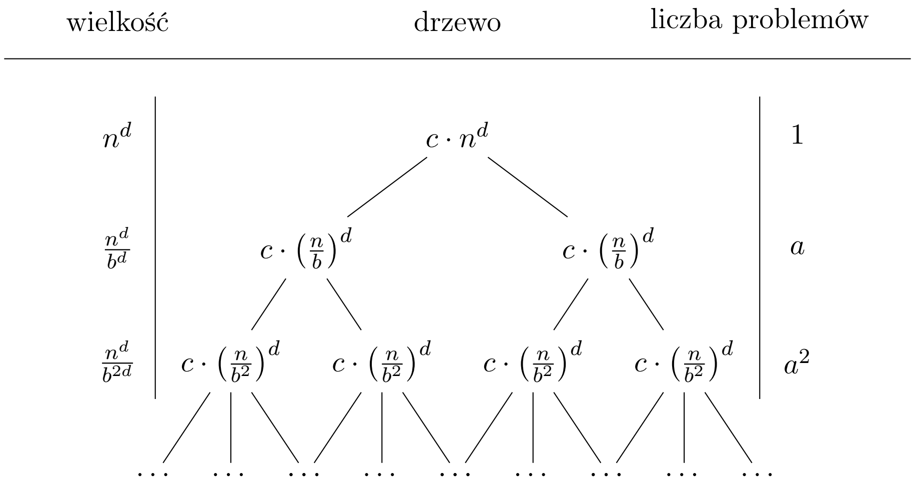
1. $\Theta(n^d)$ jeśli $d > \log_b a$
2. $\Theta(n^d \log n)$ jeśli $d = \log_b a$
3. $\Theta(n^{\log_b a})$ jeśli $d < \log_b a$

[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) $d > \log_b a$
   $$T(n) = \Theta(n^d)$$
   root – heavy

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

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

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

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

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

[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:

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

  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

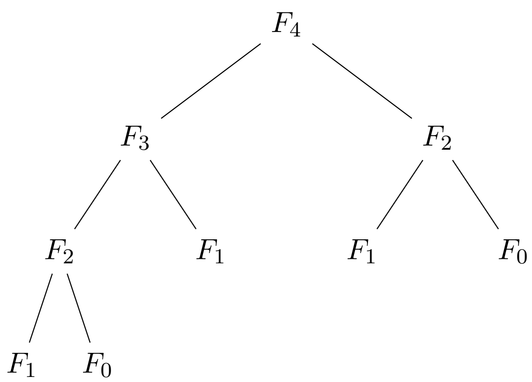
- **Problem**:

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

- **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 przybilżeniu 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

```
procedure MULTIPLY(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 = Multiply(x_L, y_L)
    p_2 = Multiply(x_R, y_R)
    p_3 = Multiply(x_L + x_R, y_L + y_R)
    return p_1 · 2^{2n} + (p_3 − p_1 − p_2) · 2^n + p_2
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:

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

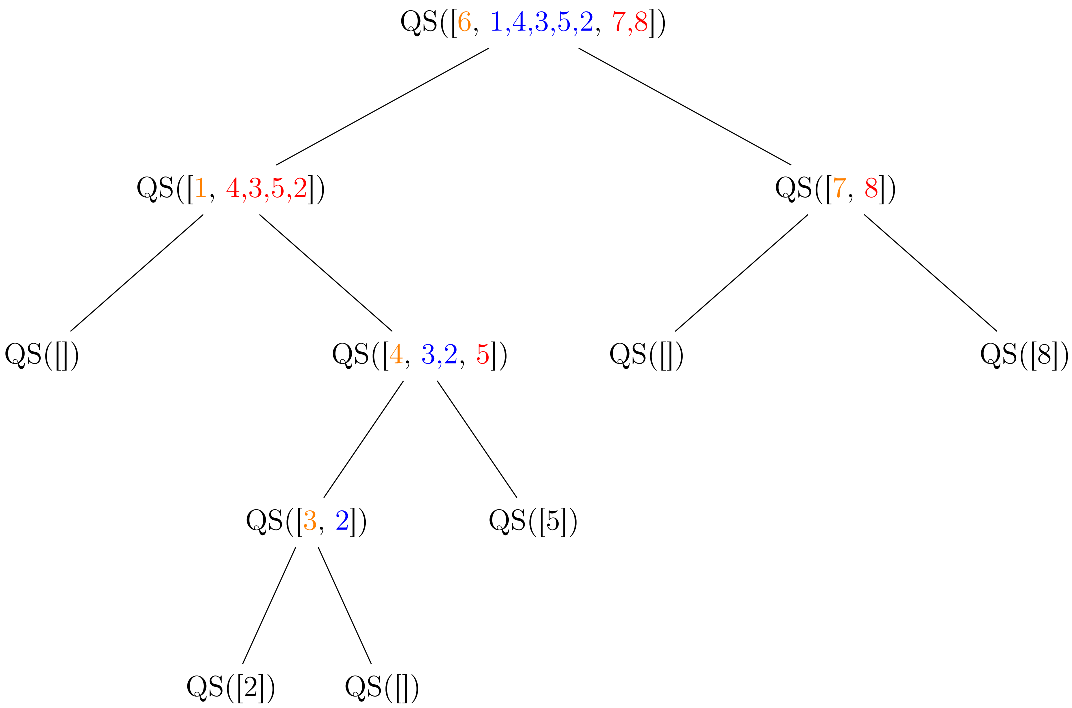
     Po dalszym sortowaniu otrzymamy $[1, 2, 3, 4, 5]$
     – `QuickSort(A, 7, 8)` dla $[7, 8]$, który już jest posortowany.

[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

procedure LEMUTO(A, p, q)
    pivot $= A[p]$
    $i = p$
    for $j = p + 1$ to q **do**
        if $A[j] <$ pivot **then**
            $i = i + 1$
            swap $A[i] \leftrightarrow A[j]$
        end if
    end for
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

```
procedure HOARE(A, p, q)
    pivot = A[(p+q)/2]
    i = p − 1
    j = q + 1
    while True do
        i = i + 1
        while A[j] > pivot do
            j = j − 1
            if ≥ j then
                break
            end if
        end while
        swapA[i] ↔ A[j]
    end while
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

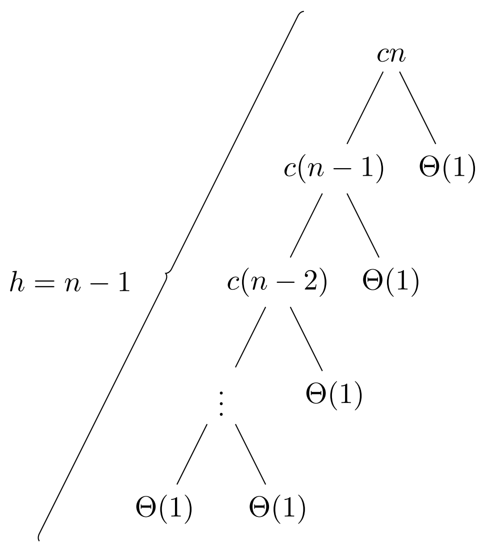
- **Asymptotyka**: *Hoare Partition* wykonue $n \pm c$ porównań – o stałą więcej niżeli *Lemuto Partition*, ale za to wykonuje mniej zamian elementów. W praktyce *Hoare Partition* jest szybszy. Całościowa Asymptotyka wynosi $\Theta(n)$

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:

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

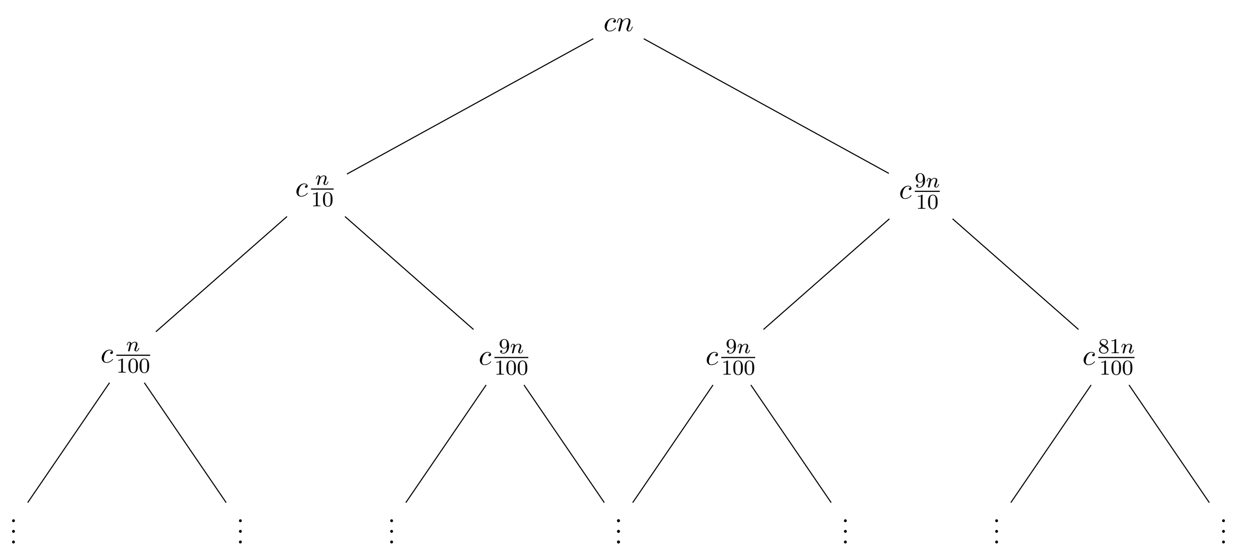
Można zauważyć, z **Master Theorem**, że asymptotyka wynosi:

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

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 bedziędziemy 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:

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

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:

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

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

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 $p < q$

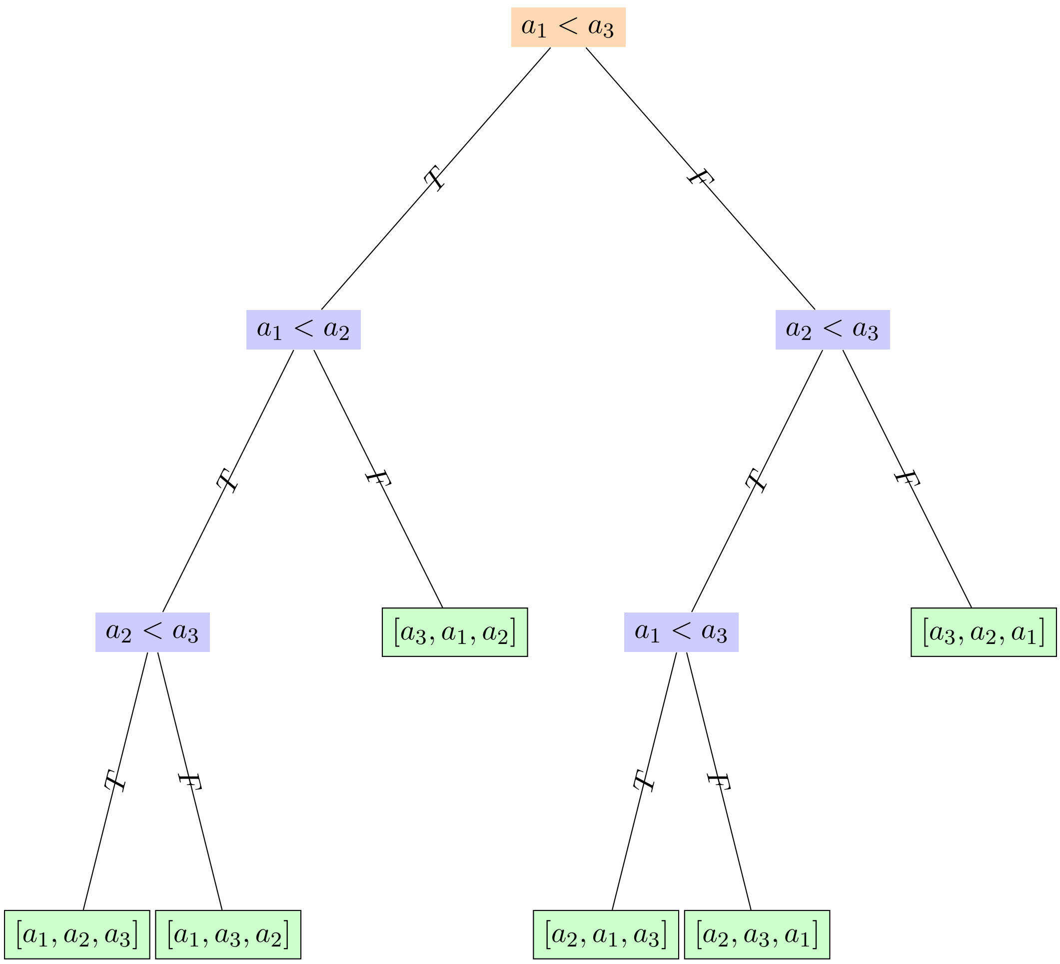
$$\left[\, a \,\middle|\, \underbrace{\cdots}_{S_{i-1}} \; p \,\middle|\, \cdots \; q \,\middle|\, \underbrace{\cdots}_{L_{i-1}} \; \boxed{i} \,\middle|\, b \,\right]$$

- 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

```
procedure COUNTINGSORT(A, n, k)
    for i = 1 to k do                                   ▷ Θ(k)
        C[i] = 0
    end for
    for j = 1 to n do                                   ▷ Θ(n)
        C[A[j]] = C[A[j]] + 1
    end for
    for i = 2 to k do                                   ▷ Θ(k)
        C[i] = C[i] + C[i − 1]
        C[i] = C[i] + C[i − 1]      ▷ liczba elementów mniejszych lub równych i
    end for
    for j = n downto 1 do                               ▷ Θ(n)
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
    end for
    return B                                            ▷ Θ(1)
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):

$$\underbrace{|\_\_\_|\_\_\_|\ \dots\ |\overbrace{\_\_\_}^{r\text{-bitów}}|}_{b\text{-bitów}}$$

- cyfry są z $\mid \{0, \dots, 2^r - 1\} \mid = 2^r$
- *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 $\{0, \dots, n^d - 1\}$ - zakres sortowanych liczb, wtedy $b = \log n^d = d \log n$, a więc

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

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

$$A = \boxed{6}\boxed{10}\boxed{13}\boxed{5}\boxed{8}\boxed{3}\boxed{2}\boxed{11}$$

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

Algorithm 8 Random Select

```
procedure RandomSelect(A, p, q, i)
    if p = q then
        return A[p]
    end if
    r = RandomPartition(A, p, q)                ▷ losowa partycja
    k = r − p + 1
    if i = k then
        return A[r]                             ▷ A[r] jest i-tą statystyką pozycyjną
    else if i < k then
        return RandomSelect(A, p, r − 1, i)
    else
        return RandomSelect(A, r + 1, q, i − k)
    end if
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

$$\begin{array}{|c|c|c|c|} \hline 6 & 5 & 3 & 2 \\ \hline \end{array}.$$

Krok 2.
Dla podtablicy

$$B = \begin{array}{|c|c|c|c|} \hline 6 & 5 & 3 & 2 \\ \hline \end{array},$$

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

$$B = \begin{array}{|c|c|c|c|} \hline \underline{2} & 5 & 3 & 6 \\ \hline \end{array}$$

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

$$B' = \begin{array}{|c|c|c|} \hline 5 & 3 & 6 \\ \hline \end{array}.$$

Krok 3.
Dla podtablicy

$$C = \begin{array}{|c|c|c|} \hline 5 & 3 & 6 \\ \hline \end{array},$$

zakładamy, że pivotem jest 6.
Po partycjonowaniu otrzymujemy:

$$C = \begin{array}{|c|c|c|} \hline 5 & 3 & \underline{6} \\ \hline \end{array}$$

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:

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

    Wówczas możemy zapisać:

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

    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:

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

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 $1 \le k \le n-2$) 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:

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

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 $\left\lfloor \frac{n}{5} \right\rfloor$ pięcio elementowe częsci oraz ostatnią część z resztą
  2. Sortujemy każdą pięcio elementową część i wybieramy z nich mediany:

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

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

- **Asymptotyka**: Algorytm spełnia rekurencję:

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

  Gdzie $T(?)$ oznacza czas rekuryncеyjnego 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:

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

  Zatem górne ograniczenie na *?* jest rzędu:

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

  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 $a \in A \to \exists 1 \geq a.key \geq \text{length}()$, 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	$\Theta(n)$	–	–	–	–	–

8.1.1 Binary Search Tree – BST

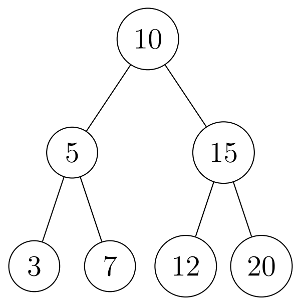
Po polsku drzewo przeszukiwań binarnych. Struktura ta zawiera następujące pola:

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

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

  Rozwiążemy to rekurencyjnie:

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

Algorithm 9 InorderTreeWalk

```
procedure InorderTreeWalk(x)
    if x ≠ NIL then
        InorderTreeWalk(x.left)
        print(x.key)
        InorderTreeWalk(x.right)
    end if
end procedure
```

- 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

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

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.

Algorithm 11 TreeMin

```
procedure TreeMin(x)
    while x.left ≠ null do
        x = x.left
    end while
    return x
end procedure
```

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 $\Theta(h)$.

Algorithm 12 TreeSuccessor

```
procedure TreeSuccessor(x)
    if x.right ≠ null then
        return TreeMin(x.right)
    else
        y = x.parent
        while y ≠ null and x = y.right do
            x = y
            y = y.parent
        end while
        return y
    end if
end procedure
```

- **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 13 BST-Insert

```
procedure BST-Insert(T, z)
    y = null
    x = T.root
    while x ≠ null do
        y = x
        if z.key < x.key then
            x = x.left
        else
            x = x.right
        end if
    end whilez.parent = y
    if y = null then T.root = z
    else if z.key < y.key then y.left = z
    elsey.right = z
    end ifz.left = null z.right = null
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.

  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.

  Złożoność algorytmu to $\Theta(n)$.

Algorithm 14 BST-Delete

procedure BST-DELETE(T, z)
  if $z.left = null$ **then** $y = z.right$
  else if $z.right = null$ **then** $y = z.left$
  else$y = TreeSuccessor(z)$ $z.key = y.key$ $y = y.right$
  end if$y.parent = z.parent$
  if $z.parent = null$ **then** $T.root = y$
  else if $z = z.parent.left$ **then** $z.parent.left = y$
  else$z.parent.right = y$
  end if
end procedure

Algorithm 15 BST-Sort

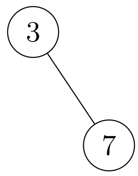
procedure BST-SORT(T)
  InorderTreeWalk($T.root$)
end procedure

Na przykładzie wygląda to następująco:

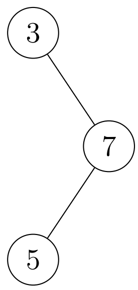
Teraz przedstawiamy drzewo BST krok po kroku przy wstawianiu elementów z tablicy:

Krok 1: Wstawienie 3

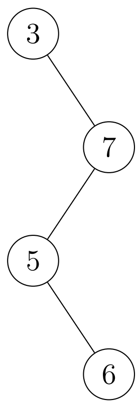
Krok 2: Wstawienie 7



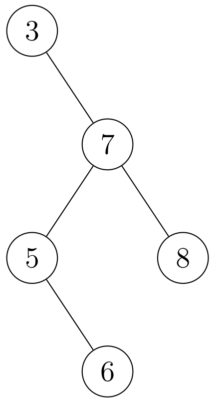
Krok 3: Wstawienie 5



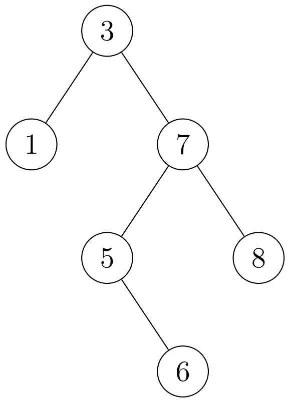
Krok 4: Wstawienie 6



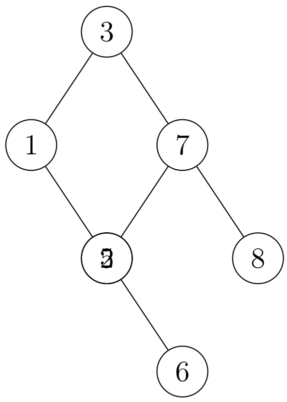
Krok 5: Wstawienie 8



Krok 6: Wstawienie 1



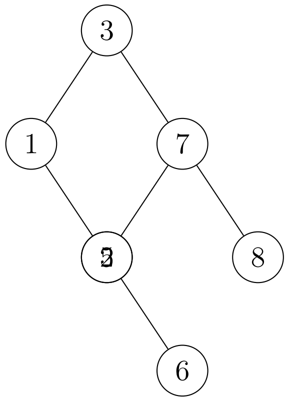
Krok 7: Wstawienie 2



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:

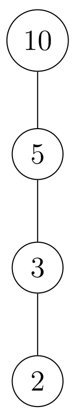
$$\mathbb{E}\left[\sum_{x \in T} \text{depth}(x)\right] = \sum_{x \in T} \mathbb{E}\left[\text{depth}(x)\right] = \Theta(n \log n)$$

Dzieląc obustronnie przez n otrzymujemy:

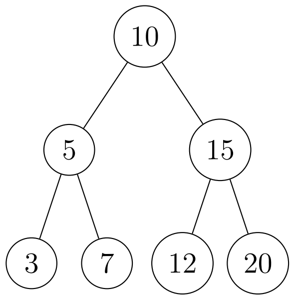
$$\mathbb{E}\underbrace{\left[\frac{1}{n}\sum_{x \in T} \text{depth}(x)\right]}_{\substack{\text{średnia głębokość} \\ \text{węzła w losowym} \\ \text{drzewie BST}}} = \Theta(\log n)$$

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:

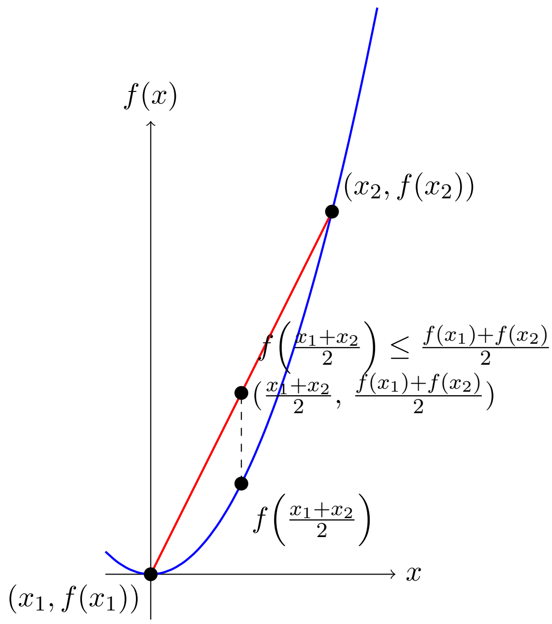
$$\mathbb{E}\left[h(T)\right] \leq 3\log_2 n + o(\log n)$$

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):

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

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

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

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

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

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

Udowodnimy powyższe przy użyciu indukcji:

- Base Case: $y_0 = y_1 = 0$
- założenie indukcyjne:

$$\forall k < n : y_k \le cn^3$$

- krok indukcyjny:

$$y_n \le \frac{4}{n}\sum_{k=0}^{n-1} y_k \le \frac{4}{n}\sum_{k=0}^{n-1} ck^3 = \frac{4c}{n}\sum_{k=0}^{n-1} k^3$$

  Ograniczmy to z góry przez całkę:

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

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

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 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.

9.1 Lista 2

robiona na zajęciach 2025-03-10

9.1.1 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

```
function f(n)
if n > 1 then
    print_line('still going')
    f(n/3)
    f(n/3)
end if
```

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.

9.1.2 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

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

□

9.1.3 zadanie 3

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

Algorithm 16 Pierwszy fragment kodu

for $i = 1$ to n **do**
    $j = i$
    while $j < n$ **do**
        $sum = P(i, j)$
        $j = j + 1$
    end while
end for

Algorithm 17 Drugi fragment kodu

for $i = 1$ to n **do**
    $j = i$
    while $j < n$ **do**
        $sum = R(i, j)$
        $j = j + j$
    end while
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

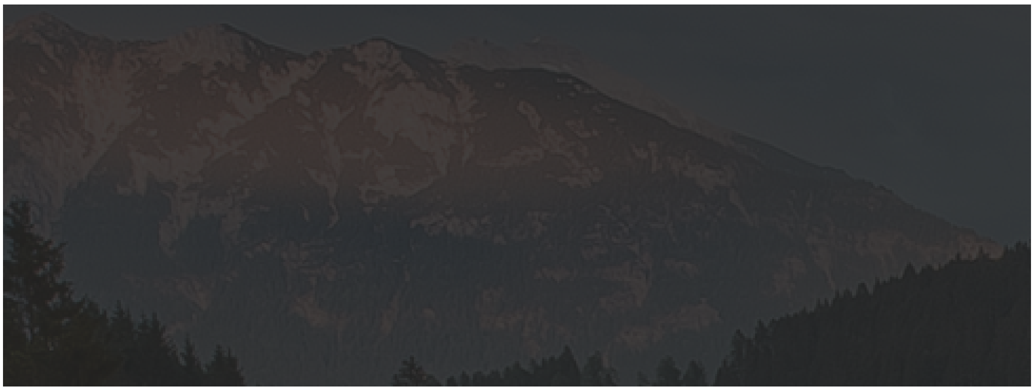
$$\sum_{i=1}^{n} \Theta(n - i) = \Theta(n^2)$$

- 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:



9.1.4 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.

$$a = 2, b = 2, d = 0$$

$$\log_b a = \log_2 2 = 1 > 0 = d$$

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

$\square$

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

  Dowód.

  $$a = 2, b = 2, d = 1$$

  $$\log_b a = \log_2 2 = 1 = d$$

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

  $\square$

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

  Dowód. Dolne ograniczenie

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

  Górne ograniczenie

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

  $\square$

9.1.5 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 18 Algorytm do zadania 5.

```
B[1] = A[1]
for i = 2 to N do
    B[i] = B[i − 1] + A[i]
end for
procedure Sum(i, j)
    if i = 1 then
        return B[j]
    else
        return B[j] − B[i − 1]
    end if
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.

9.1.6 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.

Algorithm 19 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)

9.1.7 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:

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: $11010 + 110100 = 1000000$.

9.2 Lista 3

robiona na zajęciach 2025-03-24

Algorithm 20 Algorytm do zadania 7 (pokazany na wykładzie)

procedure MUL(x, y)
  $n = \max\{|x|, |y|\}$
  if $n = 1$ **then**
    return $x \cdot y$
  end if
  $x_L, x_R =$ LetfMost $\left\lceil \frac{n}{2} \right\rceil$, RightMost $\left\lceil \frac{n}{2} \right\rceil$ bits of x
  $y_L, y_R =$ LetfMost $\left\lceil \frac{n}{2} \right\rceil$, RightMost $\left\lceil \frac{n}{2} \right\rceil$ bits of y
  $p_1 = \text{Mul}(x_L, y_L)$
  $p_2 = \text{Mul}(x_R, y_R)$
  $p_3 = \text{Mul}(x_L + x_R, y_L + y_R)$
  return $p_1 \cdot 2^{2n} + (p_3 - p_1 - p_2) \cdot 2^n + p_2$

9.2.1 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:

Algorithm 21 Algorytm do zadania 1.

procedure MERGELISTS($L_1, L_2, \ldots, L_k$)
  $n = \sum_{i=1}^{k} |L_i|$
  $B = \text{tablica}[1 \ldots n]$
  heap = MinHeap
  for $i = 1$ to k **do**
    heap.insert(L_i.pop())
  end for
  for $i = 1$ to n **do**
    $B[i] =$ heap.pop()
    heap.insert(L_i.pop())
  end for
end procedure=0

9.2.2 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 $O(\cdot)$).

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:

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

9.2.3 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 22 Algorytm do zadania 3.

procedure FINDMIN(A, p, q)
    if $p = q$ **then**
        return $A[p]$
    end if
    $m = \left\lfloor \frac{p+q}{2} \right\rfloor$
    if $A[m] > A[m+1]$ **then**
        return FindMin$(A, m+1, q)$
    else
        return FindMin(A, p, m)
    end if
end procedure

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

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

□

9.2.4 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

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

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

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

□

9.2.5 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 $\Theta(n^2)$.

9.2.6 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.

9.2.7 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 23 Algorytm do zadania 7.

```
procedure DOMINAT(A, p, q)
    if p = q then
        return A[p]
    end if
    dom_L = Dominat(A, p, ⌊(p+q)/2⌋)
    dom_R = Dominat(A, ⌊(p+q)/2⌋ + 1, q)
    if dom_L = dom_R then
        return dom_L
    end if
    count_L = count(A, p, q, dom_L)
    count_R = count(A, p, q, dom_R)
    if count_L > count_R then
        return dom_L
    else
        return dom_R
    end if
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 24 Algorytm do zadania 7. działajcy w czasie liniowym.

```
procedure MAJORITY(A)
    count = 0
    candidate = None
    for i = 0 to n − 1 do
        if count = 0 then
            candidate = A[i]
        end if
        if A[i] = candidate then
            count = count + 1
        else
            count = count − 1
        end if
    end for
    return candidate
end procedure
```

9.2.8 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 $O(n \log n)$

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

9.2.10 zadanie 10

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