Notatki z Algorytmów i Struktur Danych

Jakub Kogut

27 marca 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 $T \to$ 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

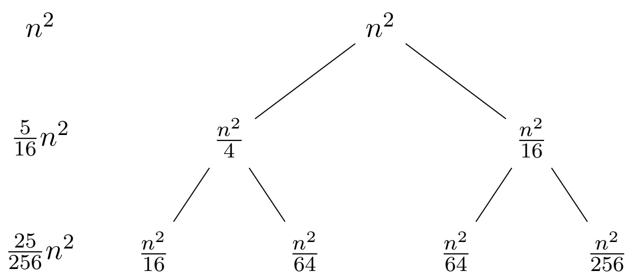
    $$T(n) = \log n \log\log n$$

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:

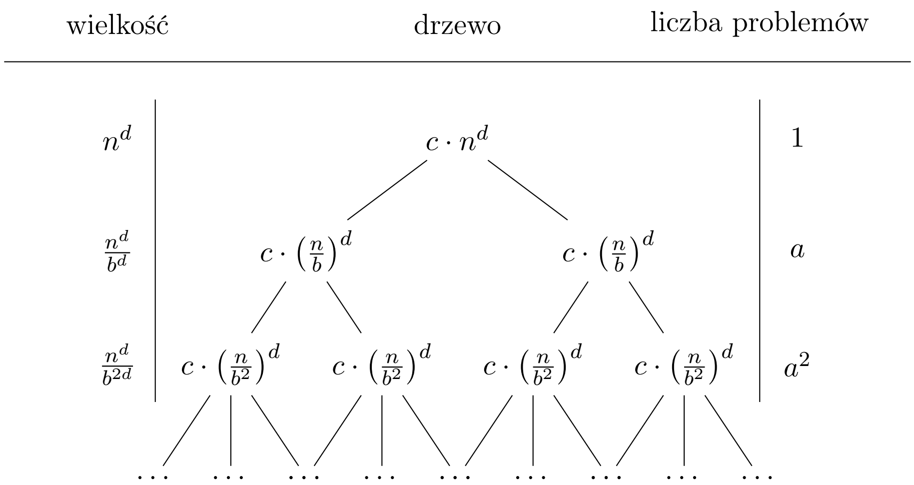
1. $\Theta(n^d)$ jeśli $d > \log_b a$
2. $\Theta(n^d \log n)$ jeśli $d = \log_b a$
3. $\Theta(n^{\log_b a})$ jeśli $d < \log_b a$

[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 – he
   avy

   (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 – he
   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

```
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

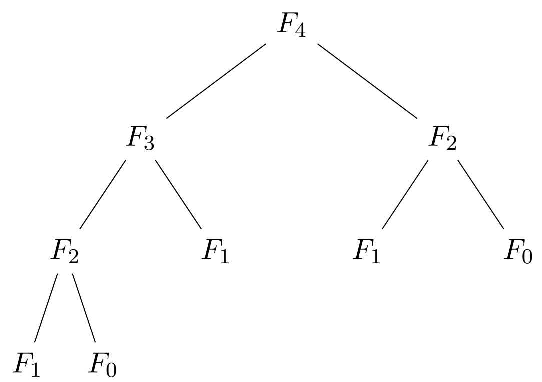
- **Problem**:

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

- **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:
założmy, 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 $x \cdot y$
    end if
    $x_L, x_R =$ LetfMost $\lceil \frac{n}{2} \rceil$, RightMost $\lceil \frac{n}{2} \rceil$ bits of x
    $y_L, y_R =$ LetfMost $\lceil \frac{n}{2} \rceil$, RightMost $\lceil \frac{n}{2} \rceil$ bits of y
    $p_1 = \text{Multiply}(x_L, y_L)$
    $p_2 = \text{Multiply}(x_R, y_R)$
    $p_3 = \text{Multiply}(x_L + x_R, y_L + y_R)$
    return $p_1 \cdot 2^{2n} + (p_3 - p_1 - p_2) \cdot 2^n + p_2$
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.

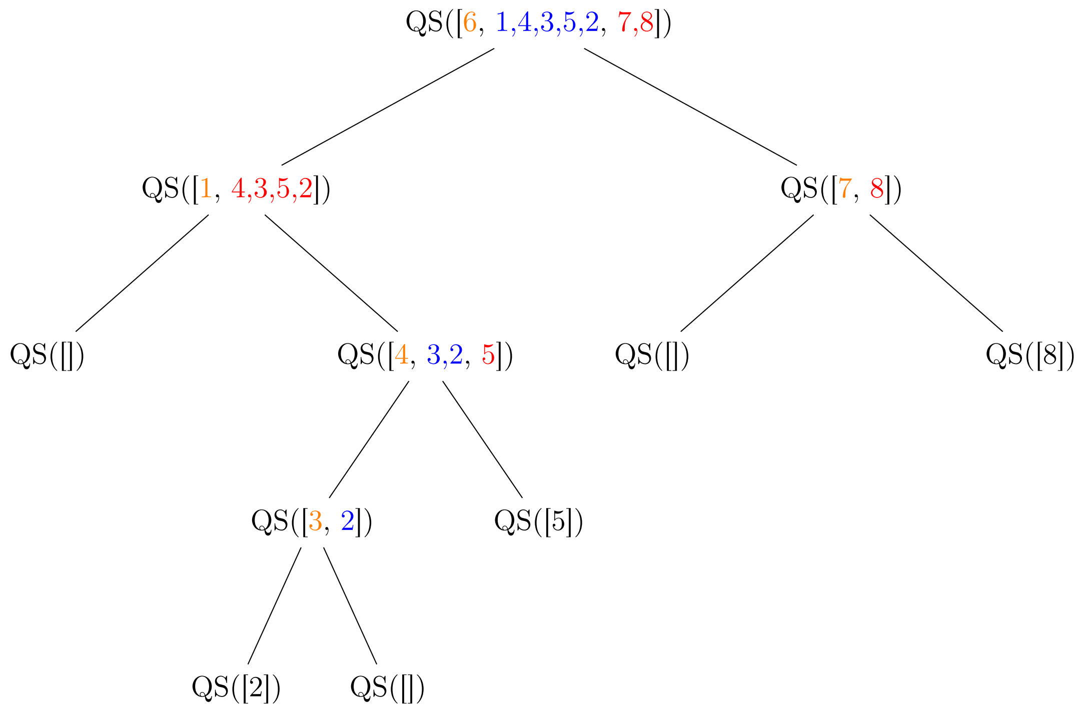
[4] Aby zejść do rekurencji $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$ trzeba wykonać pewne, bardziej wyrafinowane triki, które nie są dokładnie opisane tutaj. Z algorytmu zapisanego wyżej wynika że rekurencja to $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$, a wieć złożoność wynosi $\Theta(n^3)$

[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

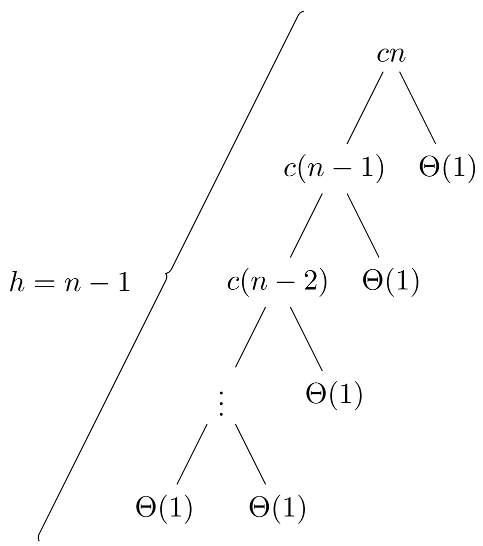
- **Asymptotyka**: *Hoare Partition* wykonue $n \pm c$ porównań – o stałą więcej niżeli *Lemuto Partition*, ale za to wykonuje mniej zamian elementów. W praktyce *Hoare Partition* jest szybszy. Całościowa Asymptotyka wynosi $\Theta(n)$

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:

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

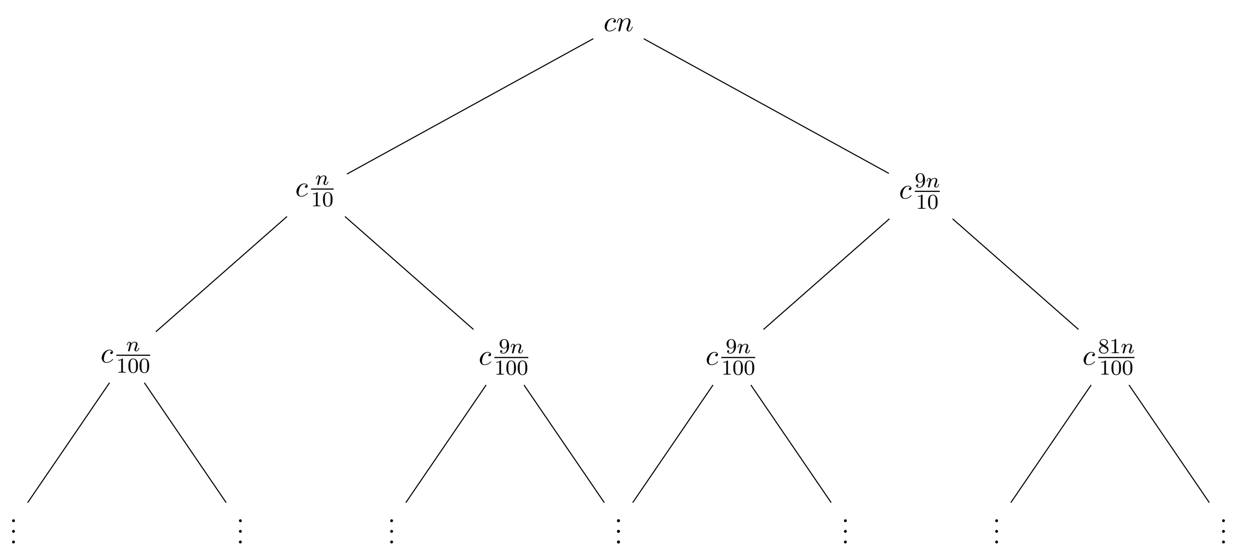
Można zauważyć, z **Master Theorem**, że asymptotyka wynosi:

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

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 bedzięmy 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:

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

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

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.

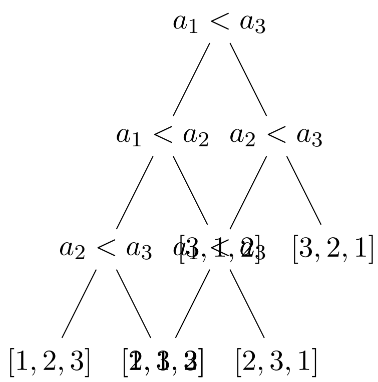
6.1.1 Strategia Count Partition

- Zakładamy $p < q$
- 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.



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

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

$$h \geq \log n! = \log \left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + o(1)) \right) = \log \left(\frac{n}{e}\right)^n + \log ($$

□

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**:

Algorithm 7 Counting Sort

procedure COUNTINGSORT(A, n, k)
  for $i = 1$ to k **do** ▷ $\Theta(k)$
    $C[i] = 0$
  end for
  for $j = 1$ to n **do** ▷ $\Theta(n)$
    $C[A[j]] = C[A[j]] + 1$
  end for
  for $i = 2$ to k **do** ▷ $\Theta(k)$
    $C[i] = C[i] + C[i-1]$
    $C[i] = C[i] + C[i-1]$ ▷ liczba elementów mniejszych lub równych i
  end for
  for $j = n$ downto 1 **do** ▷ $\Theta(n)$
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$
  end for
  return B ▷ $\Theta(1)$
end procedure

- **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 $k = O(n)$

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**:

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

7 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.

7.1 Lista 2

robiona na zajęciach 2025-03-10

7.1.1 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.

7.1.2 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

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

□

7.1.3 zadanie 3

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

Algorithm 8 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
```

Gdzie:

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

Algorithm 9 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
```

- 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

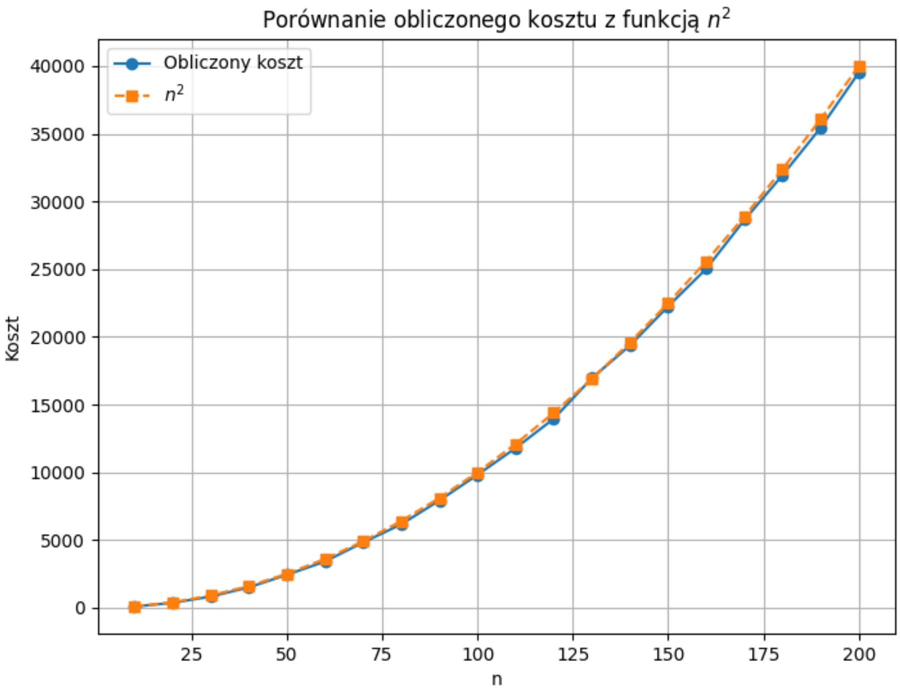
$$\sum_{i=1}^{n} \Theta(n - i) = \Theta(n^2)$$

- 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

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

$\square$

Dla pewnosci sprawdzone empirycznie:



7.1.4 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

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

Górne ograniczenie

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

□

7.1.5 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 10 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.

7.1.6 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 11 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)

7.1.7 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: $11010 + 110100 = 1000000$.

7.2 Lista 3

robiona na zajęciach 2025-03-24

7.2.1 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 12 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 $\lceil \frac{n}{2} \rceil$, RightMost $\lceil \frac{n}{2} \rceil$ bits of x
  $y_L, y_R =$ LetfMost $\lceil \frac{n}{2} \rceil$, RightMost $\lceil \frac{n}{2} \rceil$ bits of y
  $p_1 = \mathrm{Mul}(x_L, y_L)$
  $p_2 = \mathrm{Mul}(x_R, y_R)$
  $p_3 = \mathrm{Mul}(x_L + x_R, y_L + y_R)$
  return $p_1 \cdot 2^{2n} + (p_3 - p_1 - p_2) \cdot 2^n + p_2$

Algorithm 13 Algorytm do zadania 1.

procedure MERGELISTS$(L_1, L_2, \dots, L_k)$
  $n = \sum_{i=1}^{k} |L_i|$
  $B = \text{tablica}[1 \dots n]$
  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

7.2.2 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:

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