# Lekcja: Obliczanie złożoności obliczeniowej i pamięciowej

Złożoność obliczeniowa Big O jest sposobem określania, jak długo będzie działał algorytm w zależności od wielkości danych wejściowych. Kiedy mówimy o "Big O", zwykle skupiamy się na najgorszym scenariuszu, czyli jak długo algorytm będzie działał w najbardziej wymagających warunkach.

## Co to jest "Big O"?

Big O (czytane jako "duże O") to matematyczna notacja używana do opisania górnego limitu czasu wykonania algorytmu. Mówi nam, jak "zły" może być algorytm w najgorszym przypadku.

Jak to działa?

Wyobraźmy sobie, że masz listę liczb i chcesz znaleźć największą liczbę na tej liście. Jeśli masz 10 liczb, możesz potrzebować 10 sprawdzeń. Jeśli masz 100 liczb, możesz potrzebować 100 sprawdzeń. Widzisz, że czas potrzebny na wykonanie zadania rośnie wprost proporcjonalnie do liczby elementów. Mówimy, że taki algorytm ma złożoność O(n), gdzie n to liczba elementów.

## Różne typy złożoności Big O:

O(1) – stała: Czas wykonania nie zmienia się, bez względu na wielkość danych. Na przykład, dostęp do elementu w tablicy po indeksie.

O(n) – liniowa: Czas wykonania rośnie liniowo wraz ze wzrostem liczby danych. Na przykład, przeszukanie listy od początku do końca.

O(n²) – kwadratowa: Czas wykonania rośnie kwadratowo wraz ze wzrostem liczby danych. Na przykład, podwójne pętle przeszukujące każdy element z każdym elementem w liście.

O(log n) – logarytmiczna: Czas wykonania rośnie logarytmicznie, co oznacza, że z każdym krokiem przeszukujemy połowę pozostałych danych. Na przykład, wyszukiwanie binarne.

Dlaczego to ważne?

Zrozumienie złożoności Big O pomaga w projektowaniu efektywnych algorytmów. Jeśli algorytm ma zbyt wysoką złożoność, może być zbyt wolny lub wymagać zbyt dużo pamięci, co jest problemem przy dużych danych.

Analiza przykładów:

Kiedy analizujesz algorytm, zastanów się, jak liczba kroków, które musi wykonać, zmienia się w zależności od rozmiaru danych wejściowych. Pomijaj mniejsze detale – skup się na głównym trendzie wzrostu.

Podsumowując, Big O to sposób na myślenie o efektywności algorytmów, pozwalający przewidzieć, jak będą się zachowywać przy różnych wielkościach danych wejściowych. To ważna wiedza dla każdego, kto chce tworzyć szybkie i skuteczne programy komputerowe.

Przyjrzymy się prostemu przykładowi kodu w języku Java, który oblicza sumę pierwszych n liczb naturalnych. Następnie przeanalizujemy złożoność obliczeniową i pamięciową tego algorytmu.

public class SumCalculator {

public static int sumOfNNumbers(int n) {

int total = 0;

for (int i = 1; i <= n; i++) {

total += i;

}

return total;

}

public static void main(String[] args) {

int n = 10; // Możesz zmienić tę wartość, aby przetestować z innymi liczbami

System.out.println("Suma pierwszych " + n + " liczb naturalnych wynosi: " + sumOfNNumbers(n));

}

}

Teraz przeanalizujmy ten kod pod kątem złożoności obliczeniowej i pamięciowej:

**Złożoność obliczeniowa:**

Funkcja sumOfNNumbers używa pętli for, która wykonuje się od 1 do n (włącznie). To oznacza, że liczba operacji (w tym przypadku dodawanie) jest bezpośrednio proporcjonalna do wartości n.

Dla każdego i od 1 do n, wykonujemy jedną operację dodawania. Jeśli n wynosi 10, mamy 10 operacji; jeśli n wynosi 100, mamy 100 operacji, i tak dalej.

Zatem złożoność obliczeniowa tej funkcji wynosi O(n), co oznacza, że jest to złożoność liniowa. Im większa wartość n, tym więcej kroków potrzebujemy do obliczenia sumy.

**Złożoność pamięciowa:**

W tej funkcji używamy tylko kilku zmiennych (total, i, n). Liczba tych zmiennych nie zmienia się wraz ze wzrostem n. Oznacza to, że niezależnie od wielkości n, zużycie pamięci pozostaje stałe.

Nie tworzymy żadnych dużych struktur danych, takich jak tablice czy listy, które by rosły wraz z n.

Dlatego złożoność pamięciowa tej funkcji jest O(1), co oznacza złożoność stałą. Niezależnie od wielkości n, zużycie pamięci przez naszą funkcję nie wzrasta.

Podsumowując, funkcja sumOfNNumbers w Java ma liniową złożoność obliczeniową (O(n)) i stałą złożoność pamięciową (O(1)). To ważne w programowaniu, ponieważ pozwala nam zrozumieć, jak nasz kod będzie się skalował, gdy będziemy pracować na większych danych.

Określ złożoność obliczeniową następujących funkcji napisanych w pseudokodzie.

## Zadanie 1

A white background with black text

Description automatically generated

## Zadanie 2

A close-up of a math equation

Description automatically generated

## Zadanie 3

A white background with black text

Description automatically generated

## Zadanie 4

A close-up of a computer code

Description automatically generated

## Zadanie 5

A close-up of a mathematical equation

Description automatically generated

## Zadanie 6

A close-up of a computer code

Description automatically generated

## Zadanie 7

A close-up of a mathematical equation

Description automatically generated

## Zadanie 8

A white paper with black text

Description automatically generated

## Zadanie 9

A close-up of a white background

Description automatically generated