Worst, Average and Best Case Analysis of Algorithms

1. Worst Case Analysis (Mostly used)

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one.
- This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

2. Best Case Analysis (Very Rarely used)

- In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
- For linear search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

3. Average Case Analysis (Rarely used)

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
- Assuming all cases (including x not being in the array) are equally likely in linear search, the average case is computed by summing all cases and dividing by n+1n+1n+1, to account for the possibility that the element is absent.

Time Complexity Analysis:

- Best Case: The order of growth will be constant because in the best case we are assuming that (n) is even.
- Average Case: In this case, we will assume that even and odd are equally likely, therefore Order of growth will be linear.

- Worst Case: The order of growth will be linear because in this case, we are assuming that (n) is always odd.

Analysis of Algorithms

Analysis of Algorithms is a fundamental aspect of computer science that involves evaluating performance of algorithms and programs. Efficiency is measured in terms of time and space.

Order of Growth

Let f(n) and g(n) be the time taken by two algorithms where n >= 0 and f(n) and g(n) are also greater than equal to 0. A function f(n) is said to be growing faster than g(n) if g(n)/f(n) for n tends to infinity is 0 (or f(n)/g(n) for n tends to infinity is infinity).

Example 1: f(n) = 1000, g(n) = n + 1

For n > 999, g(n) would always be greater than f(n) because order of growth of g(n) is more than f(n).

Example 2: f(n) = $4n^2$, g(n) = 2n + 2000

f(n) has higher order of growth as it grows quadratically in terms of input size.

How do we Quickly find order of Growth?

When n >= 0, f(n) >= 0 and g(n) >= 0, we can use the below steps.

- Ignore the order terms.
- Ignore the constants
  - **How do we compare two order of growths?**

    The following are some standard terms that we must remember for comparison.
  - c < Log Log n < Log n < n1/3 < n1/2 < n < n Log n < n2 < n2 Log n < n3 < n4 < 2n < nn
  - Here c is a constant

Asymptotic Analysis

Given two algorithms for a task, how do we find out which one is better?

A naive approach is to implement both algorithms and compare their running times on different inputs, but this method has many drawbacks for analyzing algorithms.

- It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.

- It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.

Asymptotic analysis evaluates an algorithm's performance based on input size, ignoring actual running time. It measures the order of growth of time or space; for example, linear search grows linearly, while binary search grows logarithmically.