DATA STRUCTURES

LECTURE-2

ALGORITHM ANALYSIS

Dr. Sumitra Kisan

Design and development of Algorithms

Design approaches:

The primary goal in computer problem solving is to design an algorithm in such a way that it can be implemented as a correct and efficient computer program.
Two common design approaches:-

- Top-down approach
- Bottom-up approach

Top-Down Approach:

- A large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.
- Breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually.
- Designing a system starting from the highest level of abstraction and moving towards the lower levels.

Bottom-up approach:

- The bottom-up approach is also known as the reverse of top-down approaches.
- In this approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.
- Building a system by starting with the individual components and gradually integrating them to form a larger system.
- Solving sub-problems first and then using the solutions to build up to a solution of a larger problem.

Note: Both approaches have their own advantages and disadvantages and the choice between them often depends on the specific problem being solved.

Design Techniques

Brute force technique: It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate .

Divide and Conquer: This strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.
Example: Merge sort, Quicksort.

Dynamic Programming: The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table.
Example: 0-1 Knapsack, subset-sum problem.

Greedy Method: In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.

Example: Fractional Knapsack, Activity Selection.

Backtracking: This technique is very useful in solving combinatorial problems that have *a single unique solution*, where we have to find the correct combination of steps that lead to fulfillment of the task.

Such problems have multiple stages and there are multiple options at each stage.

This approach is based on exploring each available option at every stage one-by-one.

While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option.

In this way, the program explores all possible course of actions and finds the route that leads to the solution.

Example: N-queen problem, maize problem.

Performance: How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

Complexity: How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Complexity affects performance but not vice-versa.

Types of Algorithm Analysis:

1. **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case, calculate the lower bound of an algorithm.
   Example: In the linear search when search data is present at the first location of large data then the best case occurs.

2. **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm.
   Example: In the linear search when search data is not present at all then the worst case occurs.
3. **Average case**: In the average case, take all random inputs and calculate the computation time for all inputs. And then we divide it by the total number of inputs.
   Average case = all random case time / total no of case

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

•**Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity is mainly calculated by counting the number of steps to finish the execution.

• **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

Space complexity = Auxiliary space + Input size.

Asymptotic Analysis

- Data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space.
- The ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space.
- The time complexity can be compared based on operations performed on them

Let's consider an example:
Suppose we have 100 elements, and we want to insert a new element at the beginning

1. If Array is used, we first need to shift the elements towards the right, and we will add new element at the starting of the array. i.e 100 elements are shifted towards right.
2. If we consider the linked list as a data structure to add the element at the beginning. We simply add the address of the first node in the new node, and head pointer will now point to the newly added node.

Therefore, we conclude that adding the data at the beginning of the linked list is faster than the arrays.

- The running time to perform any operation depends on the size of the input.
- If the input size is n, then f(n) is a function of n that denotes the time complexity.
- There might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes.

Example: **$f(n) = 5n^2 + 6n + 12$**

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to $5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$

% of running time due to $6n = \frac{6}{5+6+12} * 100 = 26.09\%$

% of running time due to $12 = \frac{12}{5+6+12} * 100 = 52.17\%$

From the previous calculation, it is observed that most of the time is taken by 12.

But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12.

Let's assume the different values of n to find the growth rate of f(n).

n	$5n^2$	6n	12
1			
10			
100			
1000			

- $f(n) = 5n^2$

*Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an **Asymptotic complexity**. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.*

Asymptotic Notations

The commonly used asymptotic notations for calculating the running time complexity of an algorithm are:

- ➢Big oh Notation (O)
- ➢Omega Notation (Ω)
- ➢ Theta Notation (θ)

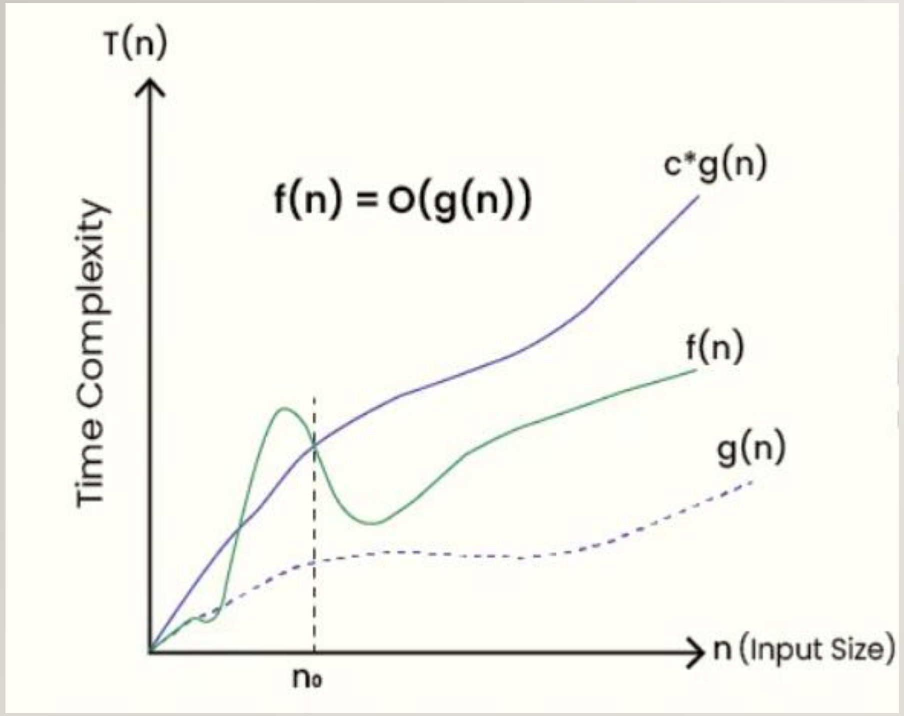
Big oh Notation (O)

- **Big-O**, commonly referred to as "**Order of**", is a way to express the **upper bound** of an algorithm's time complexity, since it analyses the **worst-case** situation of algorithm.
- It provides an **upper limit** on the time taken by an algorithm in terms of the size of the input.
- It's denoted as **O(f(n))**, where **f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.

Definition of Big-O Notation:

Given two functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there exist constants **c > 0** and $\mathbf{n_0}$ **>=** 0 such that

f(n) <= c*g(n) for all **n >=** $\mathbf{n_0}$.



Example 1: f(n)=2n+3 , g(n)=n

Is f(n)=O(g(n))?

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

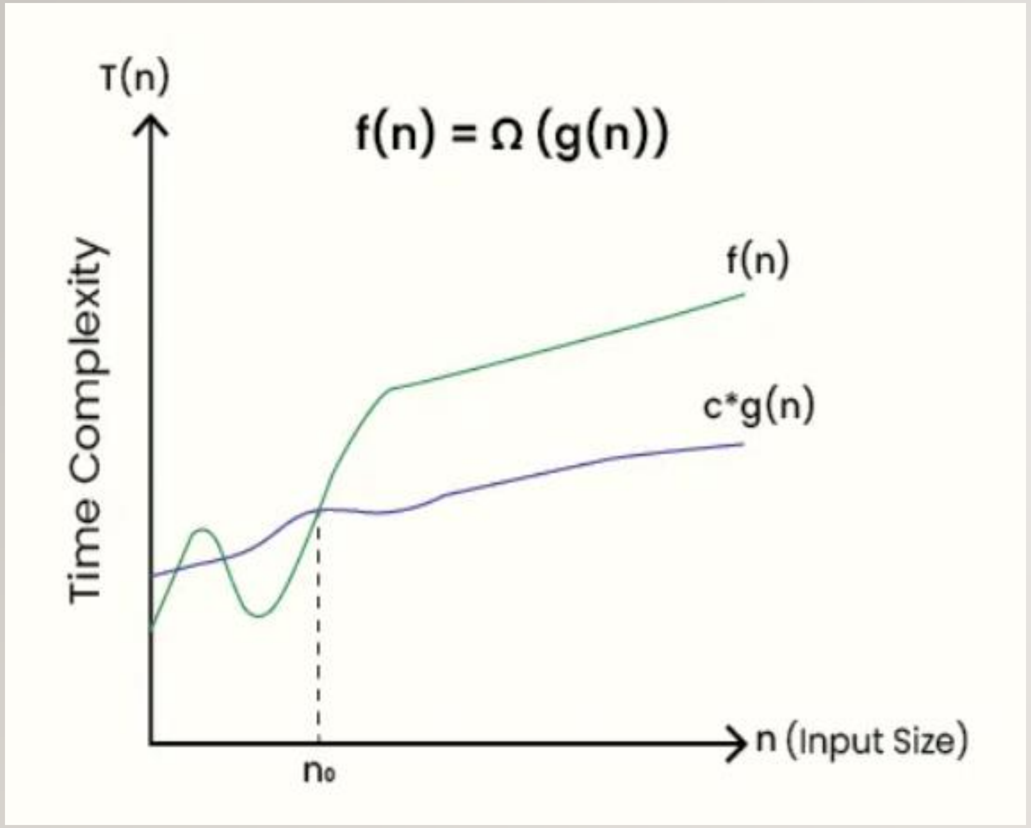
Big-Omega Ω Notation

- **Big-Omega Ω Notation**, is a way to express the **asymptotic lower bound** of an algorithm's time complexity, since it analyses the **best-case** situation of algorithm.
- It provides a **lower limit** on the time taken by an algorithm in terms of the size of the input.
- It's denoted as **Ω(f(n))**, where **f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.

Definition of Big-Omega Ω Notation:

Given two functions **g(n)** and **f(n)**, we say that **f(n) = Ω(g(n))**, if there exists constants $c > 0$ and $n_0 >= 0$ such that

$$f(n) >= c*g(n) \text{ for all } n >= n_0.$$



If f(n) = 2n+3, g(n) = n,
Is f(n)= Ω (g(n))?

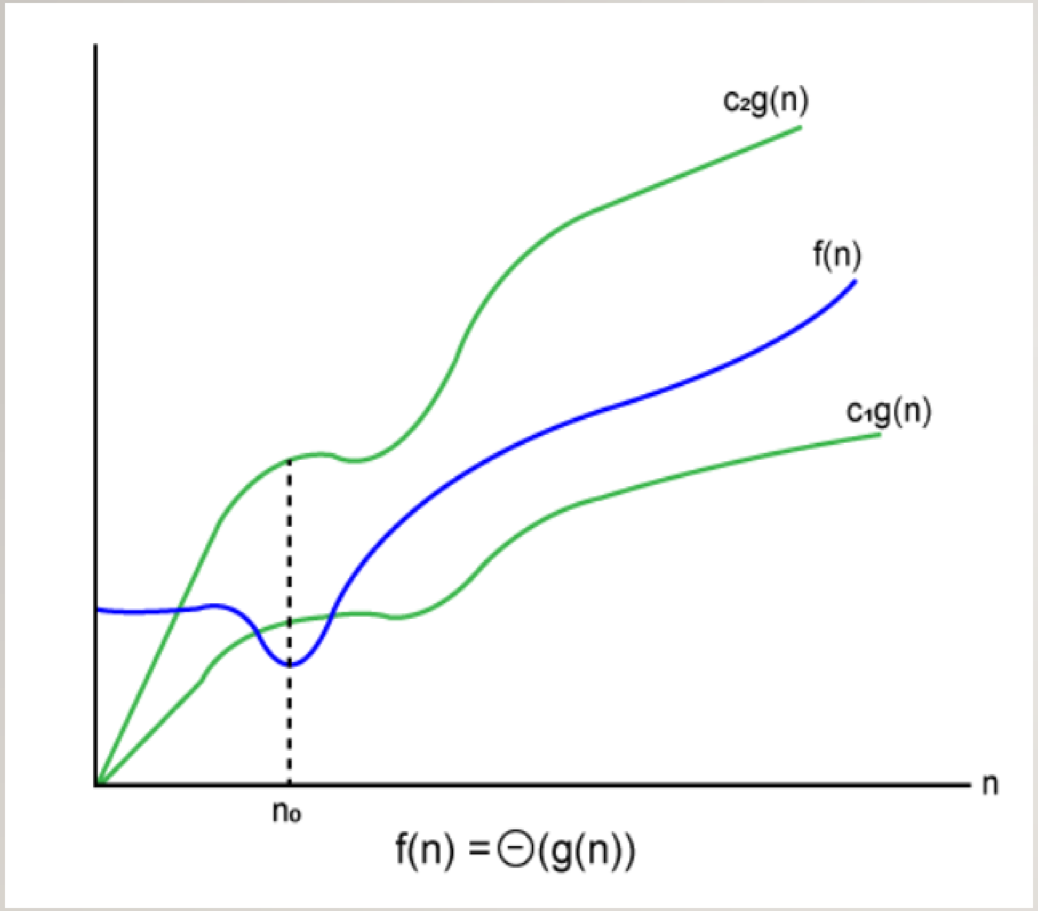
Big – Θ (Big Theta) Notation

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c_1, $c_2 > 0$ and a natural number n_0 such that

$$\mathbf{c_1 * g(n) \le f(n) \le c_2 * g(n)} \text{ for all } n \ge n_0$$

- Big – Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.

- Big theta is mainly used when the value of worst-case and the best-case is same.



Consider the same example where

f(n)=2n+3

g(n)=n

c1.g(n)<=f(n)<=c2.g(n)

S.No.	Big O	Big Omega (Ω)	Big Theta (Θ)
1.	It is like (<=) rate of growth of an algorithm is less than or equal to a specific value.	It is like (>=) rate of growth is greater than or equal to a specified value.	It is like (==) meaning the rate of growth is equal to a specified value
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. Asymptotic upper bound is given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bound is given by Omega notation.	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big O – Upper Bound	Big Omega (Ω) – Lower Bound	Big Theta (Θ) – Tight Bound
4.	It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).	It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).	It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take
5.	Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0	Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0	Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0