Design and Analysis of Algorithms

Text / Reference Books:

1. Fundamental of Computer algorithms, Ellis Horowitz and Sartaj Sahni, Galgotia Publications
2.Introduction to Algorithms, Thomas H Cormen, Charles E Leiserson and Ronald L Rivest, TMH.

Syllabus

MODULE-1: INTRODUCTION

Characteristics of algorithm, Analysis of algorithm: Asymptotic analysis of complexity bounds – best, average and worst-case behavior; Performance measurements of Algorithm, Time and space trade-offs, Analysis of recursive algorithms through recurrence relations: Substitution method, Recursion tree method and Masters' theorem.

MODULE-2: FUNDAMENTAL ALGORITHMIC STRATEGIES

Brute-Force, Greedy, Dynamic Programming, Branch and-Bound and backtracking methodologies for the design of algorithms; Illustrations of these techniques for Problem-Solving, Bin Packing, Knapsack, Job sequencing with deadline, Optimal Binary Search tree, N-Queen problem, Hamiltonian Cycle, TSP, Heuristics – characteristics and their application domains.

MODULE-3:GRAPH AND TREE TRAVERSAL ALGORITHMS

Depth First Search (DFS) and Breadth First Search (BFS); Shortest path algorithms, Transitive closure, Minimum Spanning Tree, Topological sorting, Network Flow Algorithm.

MODULE-4:TRACTABLE AND INTRACTABLE PROBLEMS

Computability of Algorithms, Computability classes – P, NP, NP-complete and NP-hard, Cook's theorem, Standard NP-complete problems and Reduction techniques.

Module-5 Advanced Topics

- Approximation algorithms, Randomized algorithms, Class of problems beyond NP – P SPACE.

Algorithms and its Characteristics

The set of instruction or description in a particular notation of the process is termed as algorithm.

- **Inputs:** An algorithm must have zero or more but must be finite number of inputs.
- **Output:** An algorithm must have at-least one desirable outcome, i.e., output.

Finiteness/ Deterministic/ Unambiguous.

Analysis of Algorithms

Complexity Calculation

Time and Space

- To analyze an algorithm means:
  - developing a formula for predicting *how fast* an algorithm is, based on the size of the input (time complexity), and/or
  - developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (space complexity)
- Usually time is our biggest concern
  - Most algorithms require a fixed amount of space

What does “size of the input” mean?

- If searching an array, the “size” of the input could be the size of the array
- If merging two arrays, the “size” could be the sum of the two array sizes
- If computing the n^{th} Fibonacci number, or the n^{th} factorial, the “size” is n
- Choose the “size” to be the parameter that most influences the actual time/space required.

Time Complexity

Lets take an example of Linear Search:

Input: An array of size n

Output: Boolean result Yes or No

```
For (i=1; i<=n; i++)
        {
                if(array[i]==x)
                {
                        printf("Yes present");
                        flag=1;
                        break;
                }
        }
if(flag==0)
        printf("Not present");
```

Average, best, and worst cases

- Usually we would like to find the *average* time to perform an algorithm
- However,
  - Sometimes the "average" isn't well defined
    - Example: Sorting an "average" array
      - Time typically depends on how out of order the array is
      - How out of order is the "average" unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the *worst* (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest

Best, average and Worst Case Complexity for Linear Search

Best Case

If you find your demanded element at the start of the array.

like a[10]={1,3,6,2,7,4,8,9,5,10}

Time Complexity will be c where c<<<10

if the size of a array is n then c<<<n

Average Case

If you find your demanded element near to the middle of the array.

like a[10]={1,3,6,2,7,4,8,9,5,10}

Time Complexity will be c where c is near to 10/2

the size of a array is n then (n/2-1)<=c<=(n/2+1)

(n/2-s)<=c<=(n/2+s) O(n)

Best, average and Worst Case Complexity for Linear search

Worst Case

If you find your demanded element at the end of the array or not finding in the array

like a[10]={1,3,6,2,7,4,8,9,5,10}

Time Complexity will be c where c is near to 10

If the size of a array is n then (n-1)<=c<=(n+1)

Omega(n)

/theeta(n) //that last element is the desired element

Asymptotic Notations

- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Why to use Asymptotic notations?

Problem1: Algorithm A complexity is n

Algorithm B complexity is n+c

Algorithm C complexity is n-c

Then how to compare which algorithm is best and balancing the complexity with Asymptotic notations.

another example: n^2-6n+50

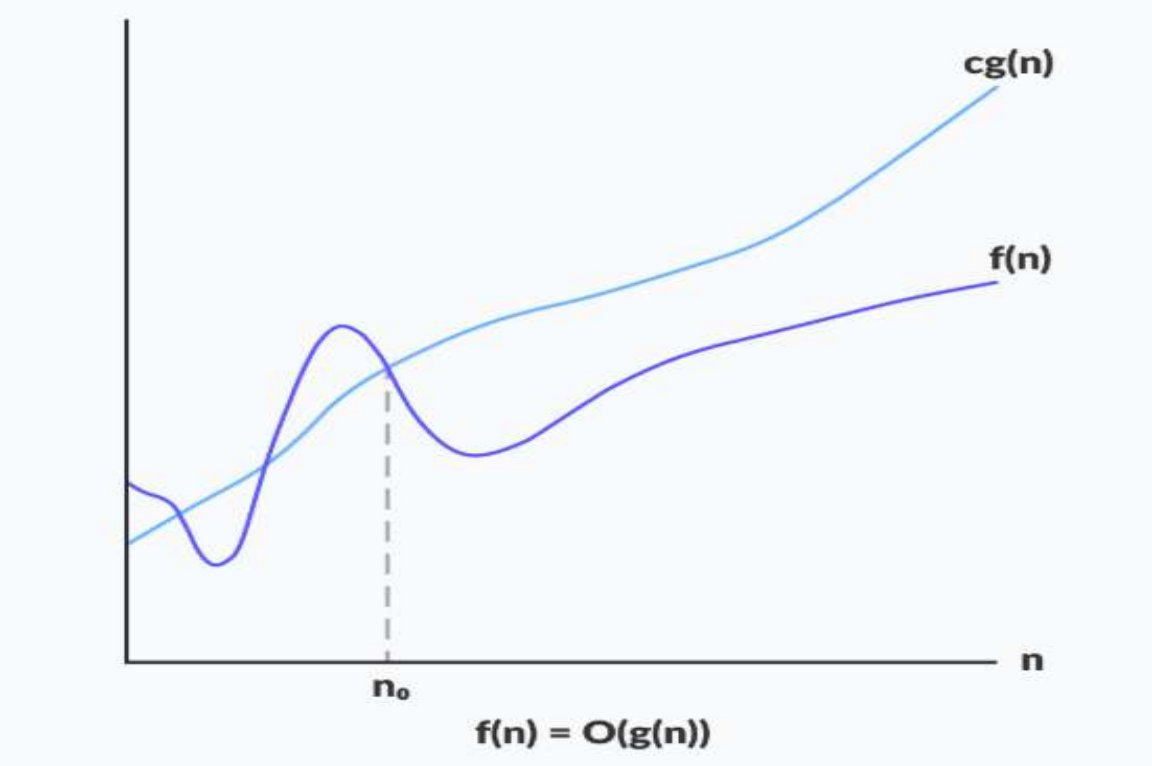
n^4-$3n^3$+$2n^2$+1

Types of Asymptotic Notations

1. Big O (O)
2. Big Omega (Ω)
3. Big Theta (Θ)

Big O Notation

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$



Big O Notation Examples

Example:

n-6 in O(n) here f(n)=n-6, g(n)=n

n-c in O(n)

n^3-2n-6 in O(n^3)

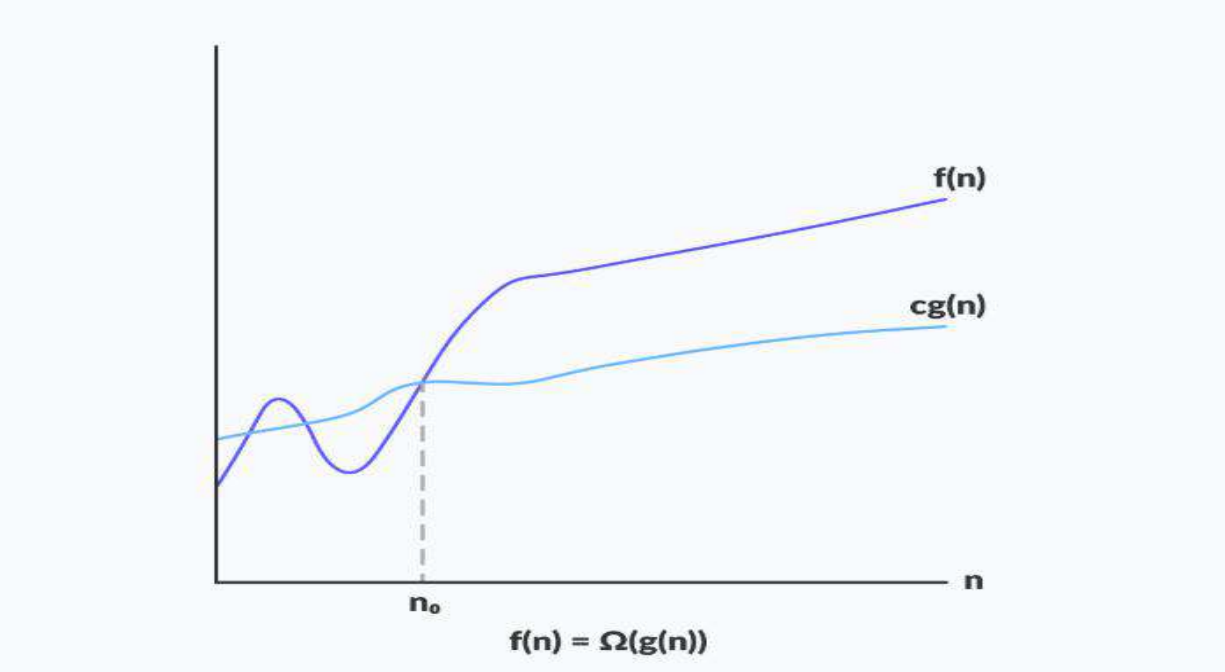
n^3+6n-16 in O(n^3)

n^3-6 in O(n^3)

The biggest advantage of the notation is that complicated expressions can be dramatically simplified.

Big Omega (Ω)

$\Omega(g(n))$ = { f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }



Big Omega (Ω) Notation Examples

Example:

n-6 in Ω(1) here f(n)=n-6, g(n)=1

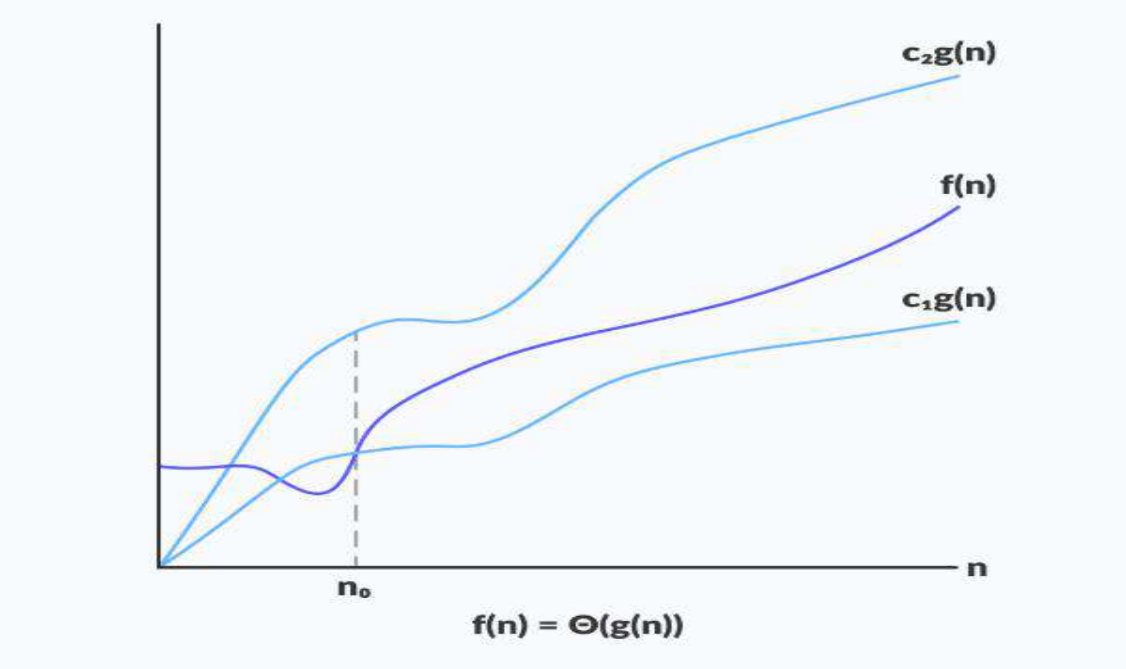
n+c in Ω(n), n-c in Ω(1)

n^3+2n+6 in $\Omega(n^3)$

n^3+6n-1 in $\Omega(n^3)$

Theta (Θ)

$\Theta(g(n))$ = { f(n): there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }



Theta (Θ) Notation Examples

Example:

n-c in Θ(n) here f(n)=n-c, g(n)=n and

c<<<<n (c is negligible to n)

n^5+6 in Θ(n^5)

n^3+or-c in Θ(n^3)