Unit 1: Role of Algorithms in Computing

Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain sequence to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Following are some important categories of algorithms −
Search − Algorithm to search an item in a data structure.
Sort − Algorithm to sort items in a certain order.
Insert − Algorithm to insert item in a data structure.
Update − Algorithm to update an existing item in a data structure.
Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

An algorithm should have the following characteristics−
Unambiguous − Algorithms should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
Input − An algorithm should have 0 or more well-defined inputs.
Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
Finiteness − Algorithms must terminate after a finite number of steps.
Feasibility − Should be feasible with the available resources.
Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.
Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
Space Factor − Space is measured by counting the maximum memory space required by the algorithm.
The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

1. Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −
Algorithm: SUM(A, B)
Step 1 − START
Step 2 − C ← A + B + 10
Step 3 − Stop
Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

2. Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.
For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic Analysis

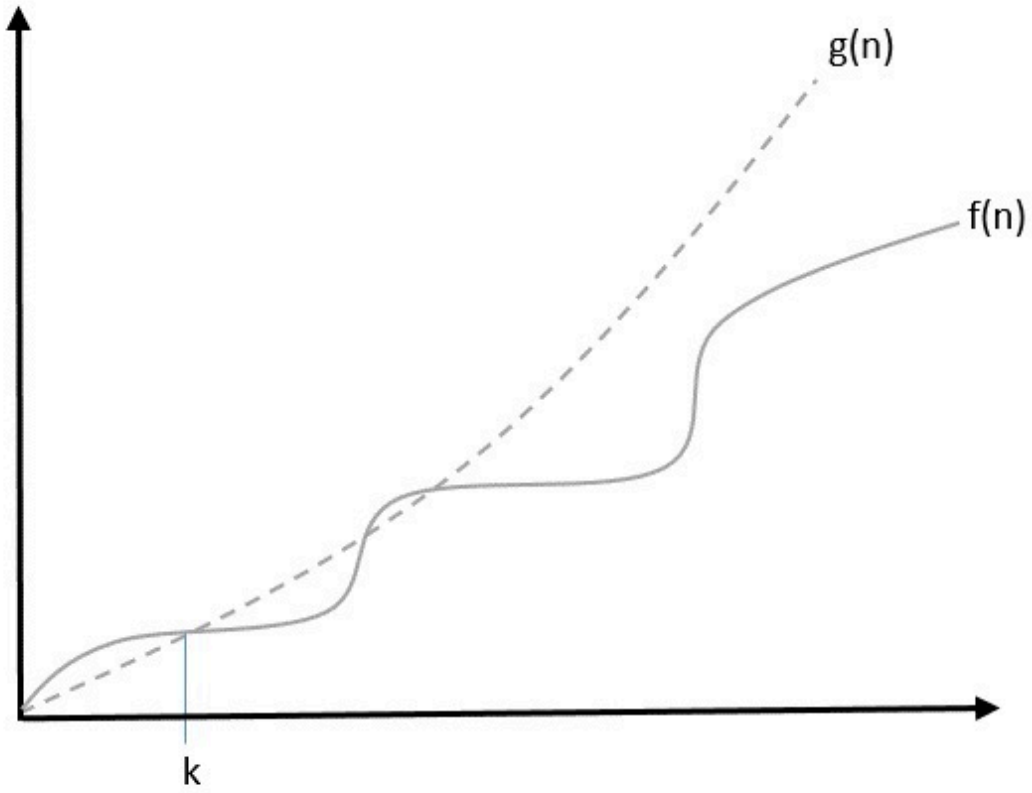
Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.
Usually, the time required by an algorithm falls under three types −
Best Case − Minimum time required for program execution.
Average Case − Average time required for program execution.
Worst Case − Maximum time required for program execution.

Asymptotic Notations

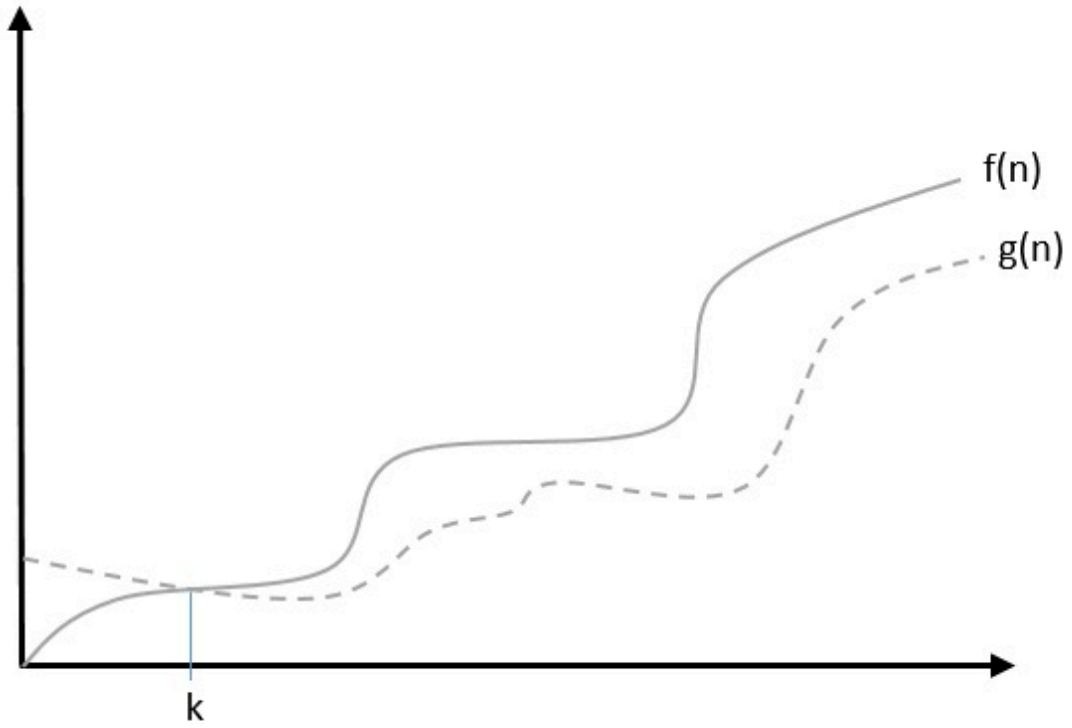
Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.
Time function of an algorithm is represented by T(n), where n is the input size.
Different types of asymptotic notations are used to represent the complexity of an algorithm.
Following asymptotic notations are used to calculate the running time complexity of an algorithm.
O − Big Oh Notation
Ω − Big omega Notation
θ − Big theta Notation

Big Oh, O: Asymptotic Upper Bound
The notation O(n) is the formal way to express the upper bound of an algorithm's running time. is the

most commonly used notation. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
A function f(n) can be represented is the order of g(n) that is O(g(n)), if there exists a value of positive integer n as n0 and a positive constant c such that −
f(n) ⩽ c. g(n) for n > n0 in all case
Hence, function g(n) is an upper bound for function f(n), as g(n) grows faster than f(n).



Big Omega, Ω: Asymptotic Lower Bound
The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
We say that f(n) = Ω(g(n)) when there exists constant c that f(n) ⩾ c. g(n) for all sufficiently large value of n. Here n is a positive integer. It means function g is a lower bound for function f ; after a certain value of n, f will never go below g.

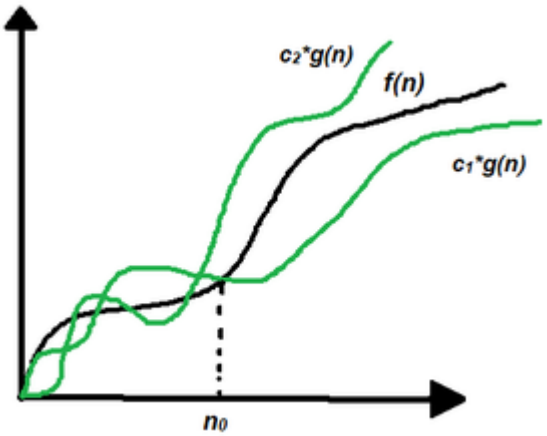


Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

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 c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation
Mathematical Representation of Theta notation:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0}

Note: Θ(g) is a set

The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.
The execution time serves as both a lower and upper bound on the algorithm's time complexity.
It exist as both, most, and least boundaries for a given input value.
A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, Consider the expression 3n3 + 6n2 + 6000 = Θ(n3), the dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.
Examples :

{ 100 , log (2000) , 10^4 } belongs to Θ(1)
{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to Θ(n)
{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to Θ(n2)

Note: Θ provides exact bounds.

Summary of Analysis of Algorithm Efficiency

- **Purpose**: Understanding and measuring how efficiently algorithms perform in terms of time and space (memory).
- **Key Aspects**:
    - **Time Complexity**: Measures the amount of time an algorithm takes to complete as a function of the input size.
    - **Space Complexity**: Measures the amount of memory an algorithm uses as a function of the input size.

Asymptotic Notation

- **Purpose**: Describes the behavior of functions as they go towards infinity. It provides a way to express the growth rates of algorithms.
- **Common Notations**:
    - **Big O Notation (O)**: Describes the upper bound of an algorithm's running time. It gives the worst-case scenario.
    - **Theta Notation (Θ)**: Describes the exact bound of an algorithm's running time. It provides both the upper and lower bounds.
    - **Omega Notation (Ω)**: Describes the lower bound of an algorithm's running time. It gives the best-case scenario.

Basic Efficiency Classes

- **Constant Time (O(1))**: The algorithm's runtime does not change with the input size.
- **Logarithmic Time (O(log n))**: The algorithm's runtime grows logarithmically with the input size.
- **Linear Time (O(n))**: The algorithm's runtime grows linearly with the input size.
- **Linearithmic Time (O(n log n))**: The algorithm's runtime grows in a combination of linear and logarithmic fashion.
- **Quadratic Time (O(n^2))**: The algorithm's runtime grows quadratically with the input size.
- **Cubic Time (O(n^3))**: The algorithm's runtime grows cubically with the input size.
- **Exponential Time (O(2^n))**: The algorithm's runtime grows exponentially with the input size.

Algorithm Design

Algorithm design is a fascinating and crucial field in computer science and engineering. It involves creating efficient step-by-step instructions to solve specific problems or perform tasks. Here are a few key concepts and techniques in algorithm design:

1. Divide and Conquer: Breaking a problem into smaller subproblems, solving each subproblem independently, and then combining the solutions to solve the original problem. Classic examples include merge sort and quicksort.

2. Dynamic Programming: Solving complex problems by breaking them down into simpler overlapping subproblems and storing the results of these subproblems to avoid redundant calculations. Examples include the Fibonacci sequence and the knapsack problem.

3. Greedy Algorithms: Making the best possible choice at each step with the hope of finding the global optimum. Examples include Dijkstra's algorithm for shortest paths and Kruskal's algorithm for minimum spanning trees.

4. Backtracking: Incrementally building candidates for the solution and abandoning a candidate as soon as it is determined that it cannot lead to a valid solution. Examples include the N-queens problem and solving Sudoku puzzles.

5. Graph Algorithms: Solving problems related to graphs, such as finding the shortest path, detecting cycles, and finding connected components. Examples include breadth-first search (BFS), depth-first search (DFS), and the A* algorithm.

6. Heuristics: Using approximate methods to find good enough solutions for complex problems where finding an optimal solution is impractical. Examples include genetic algorithms and simulated annealing.