What is an Algorithm?

Definition:

1. An algorithm is a step-by-step finite sequence of instructions to solve a particular problem.
2. It is the sequence of instructions that specify a sequence of steps to be followed to solve a specific problem.

So, to solve any given problem, we first have to define the problem and then design and analyse the algorithm required to solve that problem.

Example of an Algorithm:

Algorithm to add two numbers and store the result in a third variable.

Step1: Start

Step2: Read the first number in variable "num_1".

Step3: Read the second number in variable "num_2".

Step4: Perform the addition of num_1 and num_2 and store the result in variable "sum".

Step5: Print the value of "sum".

Step6: Stop

An algorithm must have the following properties:

1. **Finiteness**: It means that the algorithm must complete its execution after some finite number of steps.
2. **Definiteness**: Each steps of the algorithm must be precisely defined and unambiguous.
3. **Completeness or Generality**: Algorithm should be complete so that it can solve all the problems of the same type for which it is being designed.
4. **Effectiveness**: All the instructions and operations used in the algorithm should be simple and feasible so that the algorithm can be implemented in any programming language on the computer system.
5. **Input:** An algorithm must have 0 or well defined input for execution.
6. **Output:** An algorithm must give at least one output.
7. **Correctness:** The algorithm should produce the output based on the requirement of the algorithm.
8. **Efficient** – Efficiency of an algorithm is measured in terms of time and space complexity for implementing the algorithm. The algorithm should take less running time and memory space for its execution.

Why study Algorithm?

As the speed of processor increases, performance is frequently said to be less central than other software quality characteristics (e.g. security, extensibility, reusability etc.). However, large problem sizes are commonplace in the area of computational science, which makes performance a very important factor. This is because longer computation time, to name a few mean slower results, less through research and higher cost of computation (if buying CPU Hours from an external party). The study of Algorithm, therefore, gives us a language to express performance as a function of problem size.

Need of Algorithm

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
12. With the help of algorithm, we convert art into a science.
13. To understand the principle of designing.
14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. **Divide and Conquer Approach:** It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

Examples: Binary Search, Quick Sort, Merge Sort, Closest Pair of Points, Strassen's matrix multiplication algorithm.

2. **Greedy Technique:** Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

Examples: Knapsack Problem, job Sequencing with deadline, Kruskal's Minimal Spanning Tree Algorithm, Prim's Minimal Spanning Tree Algorithm, Dijkstra's Minimal Spanning Tree Algorithm, Optimal Merge Pattern, Huffman Coding.

3. **Dynamic Programming:** Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

Examples: Matrix Chain Multiplication, Longest Common Subsequence, Travelling Salesman Problem, 0/1 knapsack problem, All pair Shortest path

A branch and bound algo consist of a systematic enumeration of candidate solutions. That is, set of candidate solutions is thought of as forming a rooted tree with full set at root. The algo explores branches of this tree which represents subset of solution set. Before enumerating the candidate solution of a branch, the branch is checked against upper and lower estimated bounds.

problem, Sum of subsets problem, Optimal Cost Binary Search Tree, Multi-stage graph, Fibonacci Series.

for discrete and combinatoric optimization Problems as well as mathematics optimization

4. **Branch and Bound:** In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

   Examples: This approach is used for a number of NP-Hard problems, such as Travelling Salesman Problem, Knapsack Problem, Assignment Problem, Integer Programming, Non-Linear Programming.

5. **Randomized Algorithms:** A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

   Example: Randomized Quick Sort Algorithm, Randomized Minimum-cut Algorithm, randomized Algorithm for the N-Queens Problem.

   A randomized algorithm uses a random number at least once during the computation make a decision.

   Example 1: In Quick Sort, using a random number to choose a pivot.

   Example 2: Trying to factor a large number by choosing a random number as possible divisors.

6. **Backtracking Algorithm:** Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

   Example: Recursive Maze Algorithm, Hamiltonian Circuit Problem, Subset-sum Problem, N-Queens Problems, Graph Coloring Problem.

Analysis of Algorithm

The analysis of algorithm is performed to calculate the efficiency of an algorithm. The efficiency of an algorithm is measured on the basis of the performance of the algorithm.

We can measure the performance of an algorithm by computing two factors:

1. Amount of time required by an algorithm to execute (Time Complexity).
2. Amount of space required by an algorithm (Space Complexity).

Types:

1. **Worst Case Efficiency:** It is the maximum number of steps that an algorithm can take for any input size n.
2. **Best Case Efficiency:** It is the minimum number of steps that an algorithm can take for any input size n.
3. **Average Case Efficiency:** It is the average number of steps that an algorithm can take for any input size n.

Complexity of Algorithm

It is very convenient to classify algorithm based on the relative amount of time or relative amount of space they required and specify the growth of time/space requirement as a function of input size.

1. **Time Complexity:** Running time of a program as a function of the size of the input.
2. **Space Complexity:** Some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on the computer was limited. When considering this algorithm are divided into those that need extra space to do their work and those that work in place.

But now a day's problem of space rarely occurs because space on the computer (internal or external) is enough.

Analysis and Complexity of an Algorithm

As discussed an Algorithm is defined as "An algorithm is a step-by-step finite sequence of instructions to solve a particular problem."

The analysis of algorithm is performed to calculate the efficiency of an algorithm.

The efficiency of an algorithm is measured on the basis of the performance of the algorithm.

We can measure the performance of an algorithm by computing two factors:

1. Amount of time required by an algorithm to execute (**Time Complexity**).
2. Amount of space required by an algorithm (**Space Complexity**).

Time Complexity means CPU Time

Space Complexity means space allocated in main memory or RAM

So which is more important?

Time Complexity is more important than Space Complexity