# Algorithm

An algorithm is a process, or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Example:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b .

Step 5: store the result in the sum variable. sum= a + b.

Step 6: Print sum

Step 7: Stop

From the data structure point of view, following are some important categories of algorithms.

1. **Search − Algorithm to search an item in a data structure.**
2. **Sort − Algorithm to sort items in a certain order.**
3. **Insert − Algorithm to insert item in a data structure.**
4. **Update − Algorithm to update an existing item in a data structure.**
5. **Delete − Algorithm to delete an existing item from a data structure.**

**Algorithm Analysis:**

You can find more than one algorithms for solving a particular problem after that you have to analyze them and use the most efficient one.

The analysis of an algorithm is done base on its efficiency. The two important terms used for the analysis of an algorithm are “Priori Analysis” and “Posterior Analysis”.

1. **A Priori Analysis** − It is done before the actual implementation of the algorithm when the algorithm is written in the general theoretical language. In this, the efficiency of the algorithm is calculated based on its complexity. It is just an approximate analysis.
2. **A Posterior Analysis** − It is done after the actual implementation and execution of the algorithm using any programming language like C, C++, Java, or Python. It is the actual analysis in which the space and time complexity of the algorithm is calculated more correctly.

|  |  |
| --- | --- |
| **A Posteriori analysis** | **A priori analysis** |
| Posteriori analysis is a relative analysis. | Priori analysis is an absolute analysis. |
| It is dependent on language of compiler and type of hardware. | It is independent of language of compiler and types of hardware. |
| It will give exact answer. | It will give approximate answer. |
| It doesn’t use asymptotic notations to represent the time complexity of an algorithm. | It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution. |
| The time complexity of an algorithm using a posteriori analysis differ from system to system. | The time complexity of an algorithm using a priori analysis is same for every system. |
| If the time taken by the program is less, then the credit will go to compiler and hardware. | If the algorithm running faster, credit goes to the programmer. |
| It is done after execution of an algorithm. | It is done before execution of an algorithm. |
| It is costlier than priori analysis because  of requirement of software and hardware for execution. | It is cheaper than Posteriori Analysis. |
| Maintenance Phase is required to tune the algorithm. | Maintenance Phase is not  required to tune the algorithm. |

**Algorithm Complexity Analysis:**

An algorithm is said to be efficient and fast if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured based on following properties:

1. **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution.

sum=0;

**for** i=1 to n

sum= sum + i ;

**return** sum;

In the above code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

* Constant time – O (1)
* Linear time – O (n)
* Logarithmic time – O (log n)
* Quadratic time – O (n^2)
* Cubic time – O (n^3)

Example-1:

int main(){

cout << "Hello World";

return 0;

}

Time complexity: constant: O (1)

Example-2:

int main(){

    int i, n = 8;

    for (i = 1; i <= n; i++) {

        cout << "Hello World !!!\n";

    }

    return 0;

}

Time complexity: linear: O(n)

Example-3:

int main(){

    int i, n = 8;

    for (i = 1; i <= n; i=i\*2) {

        cout << "Hello World !!!\n";

    }

    return 0;

}

Time complexity: O(log2(n))

1. **Space complexity:**  Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result. So, to find space-complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.

**Space complexity = Auxiliary space + Input size.**

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.

**Asymptotic Analysis:**

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. our focus would be on finding the time complexity rather than space complexity. The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

1. Big oh Notation (O) -- **Worst Case**
2. Omega Notation (Ω) – **Best Case**
3. Theta Notation (θ) -- **AVERAGE CASE**
4. **Big O Notation (O):** The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures **the worst-case** time complexity or the longest amount of time an algorithm can possibly take to complete.



1. **Omega Notation (Ω):** Omega Notation (Ω) describes lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the **best-case** time complexity. It determines what is the fastest time that an algorithm can run.



1. **Theta Notation (θ):** The theta notation mainly describes the **average case** scenarios. It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. Big theta is mainly used when the value of worst-case and the best-case is same. It is the formal way to express both the upper bound and lower bound of an algorithm running time.



**Common Asymptotic Notations:**

1. Constant − Ο(1)
2. logarithmic − Ο(log n)
3. linear − Ο(n)
4. n log n − Ο(n log n)
5. quadratic − Ο(n2)
6. cubic − Ο(n3)
7. polynomial − nΟ(1)
8. exponential − 2Ο(n)

# Algorithms Classification

1. **Classification by Implementation Method:**
2. **Recursion or Iteration:** A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved. Example: The Tower of Hanoi.
3. **Exact or Approximate:** Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. Example: Approximation algorithms
4. **Serial or Parallel or Distributed Algorithms:** In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into sub problems and execute them on different processors
5. **Classification by Design Method:**
6. Greedy Method:
7. Divide and Conquer:
8. Dynamic Programming:
9. Linear Programming:
10. Reduction (Transform and Conquer):
11. Backtracking:
12. Branch and Bound:
13. **Classification by Design Approaches:**
14. **Top-Down Approach:** In the 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.
15. **Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches. In 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.

1. **Other Classifications:**
2. **Randomized Algorithms:** Algorithms that make random choices for faster solutions are known as randomized algorithms.  Example: Randomized Quicksort Algorithm.
3. **Classification by complexity:** Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.   
   Example: Some algorithms take O(n), while some take exponential time.
4. **Classification by Research Area: In** CS each field has its own problems and needs efficient algorithms.  Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.

# Algorithm Design Paradigms

The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:

1. Backtracking
2. Branch and bound
3. Brute-force search
4. Divide and conquer
5. Dynamic programming
6. Greedy algorithm
7. Recursion
8. Prune and search

# Brute force approach:

A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found. Brute force algorithm is a technique that guarantees solutions for problems of any domain helps in solving the simpler problems and also provides a solution that can serve as a benchmark for evaluating other design techniques, but takes a lot of run time and inefficient.

The time complexity of brute force is O(mn), which is sometimes written as O(n\*m).

# Backtracking approach:

1. Backtracking is an improvement of the brute force approach. It tries to search for a solution to a problem among all the available options.
2. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.
3. It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time.
4. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution.

**Types of Backtracking Problems**

Before start solving the problem we must be able to recognize if it can be solved using a backtracking algorithm. There are the following types of problems that can be solved using backtracking:

1. Decision Problem: In this type of problem we always search for a feasible solution.
2. Optimization Problem: In this type of problem we always search for the best possible solution.
3. Enumeration Problem: In this type of problem we try to find all feasible solutions.

Backtracking Algorithm Applications

1. Hamiltonian Paths.
2. N Queen problem.
3. Maze solving problem.
4. The Knight's tour problem.

# Divide and conquer approach:

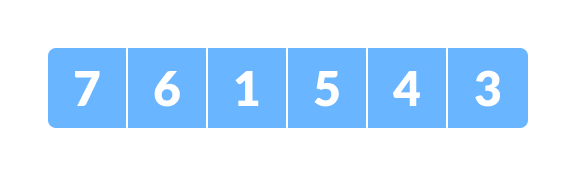
A divide and conquer algorithm is a strategy of solving a large problem by

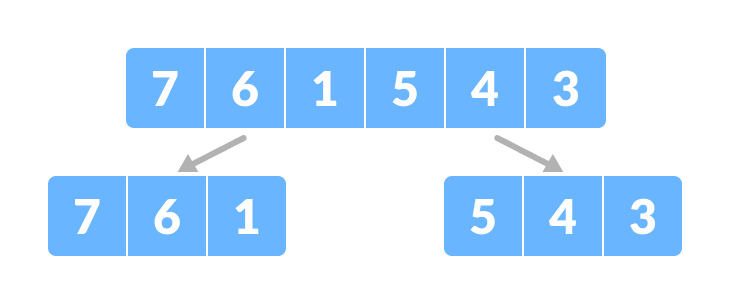
1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.
4. To use the divide and conquer algorithm, recursion is used.

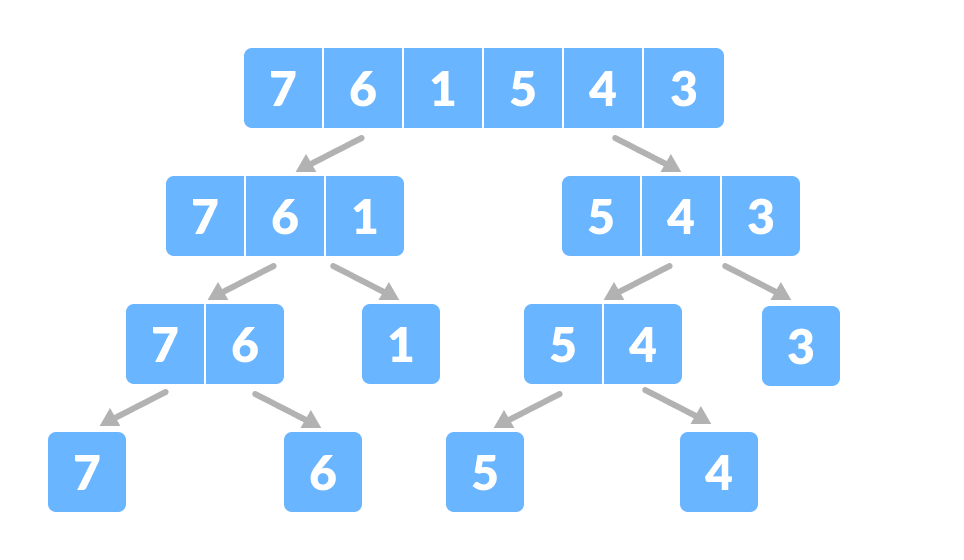
**How Divide and Conquer Algorithms Work?**

1. **Divide:** Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the sub problem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

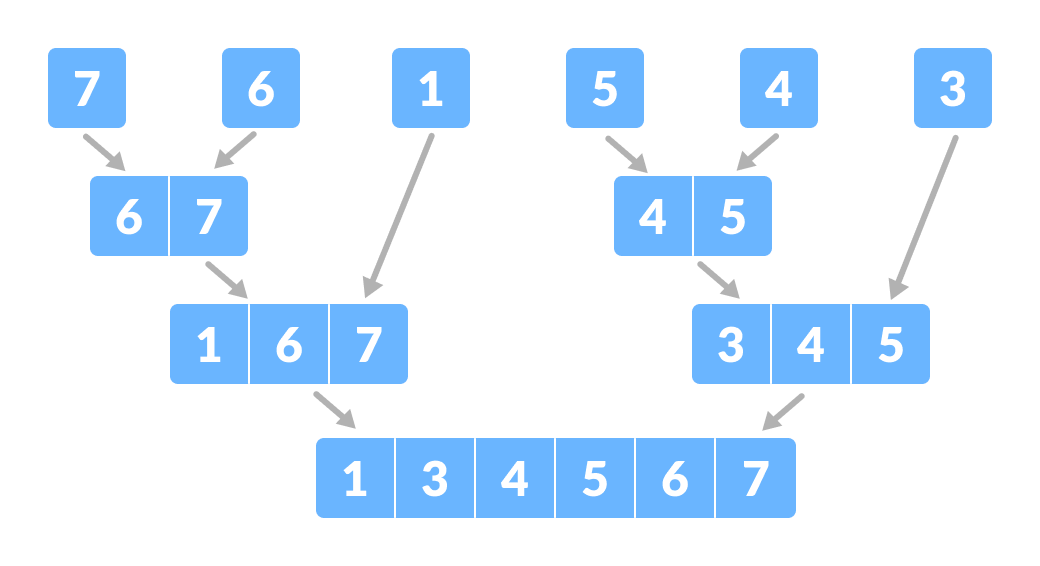
**Example:**



1. Divide the array into two halves. Divide the array into two subparts
2. divide each subpart recursively into two halves until you get individual elements. Divide the array into smaller subparts



1. combine the individual elements in a sorted manner. Here, conquer and combine steps go side by side.



Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

Algorithms are based on divide-and-conquer programming approach --

1. Merge Sort
2. Quick Sort
3. Binary Search
4. Strassen's Matrix Multiplication
5. Closest pair (points)
6. Cooley–Tukey Fast Fourier Transform (FFT) algorithm
7. Karatsuba algorithm.

# Recursion

Recursion tree method is used to solve recurrence relations. Generally, these recurrence relations follow the divide and conquer approach to solve a problem.

**Types of Recursion:** there are two types of recursion namely,

1. Linear Recursion
2. Tree Recursion

**Linear Recursion**

A linear recursive function is a function that only makes a single call to itself each time the function runs. The factorial function is a good example of linear recursion. A linearly recursive function takes linear time to complete its execution that’s why it is called linear recursion.

Consider the pseudo-code written below,

function doSomething(n) {

if nis 0:

return

doSomething(n-1);

}

**Tree Recursion**

* Tree Recursion is just a phrase to describe when you make a recursive call more than once in your recursive case. The fibonacci function is a good example of Tree recursion. The time complexity of tree recursive function is not linear, they run in exponential time.
* Consider the pseudo-code written below,

function doSomething(n) {

if n is less than 2:

return n;

return doSomething(n-1) + doSomething(n-2);

}

# Greedy approach:

1. A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
2. The algorithm never reverses the earlier decision even if the choice is wrong.
3. It works in a top-down approach.
4. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

We can choose greedy approach if the problem has the following properties:

* 1. **Greedy Choice Property:** If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.
  2. **Optimal Substructure:** If the optimal overall solution to the problem corresponds to the optimal solution to its sub problems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

**Standard Greedy Algorithms:**

1. Activity Selection Problem
2. Job Sequencing Problem
3. **Huffman Coding**
4. **Huffman Decoding**
5. Water Connection Problem
6. Minimum Swaps for Bracket Balancing
7. Egyptian Fraction
8. Policemen catch thieves
9. Fitting Shelves Problem
10. Assign Mice to Holes

**Greedy Problems on Array:**

1. Minimum product subset of an array
2. Maximize array sum after K negations using Sorting
3. Minimum sum of product of two arrays
4. Minimum sum of absolute difference of pairs of two arrays
5. Minimum increment/decrement to make array non-Increasing
6. Sorting array with reverse around middle
7. Sum of Areas of Rectangles possible for an array
8. Largest lexicographic array with at-most K consecutive swaps
9. Partition into two subarrays of lengths k and (N – k) such that the difference of sums is maximum

**Greedy Problems on Operating System:**

1. First Fit algorithm in Memory Management
2. Best Fit algorithm in Memory Management
3. Worst Fit algorithm in Memory Management
4. Shortest Job First Scheduling
5. Job Scheduling with two jobs allowed at a time
6. Program for Optimal Page Replacement Algorithm

**Greedy Problems on Graph:**

1. **Kruskal’s Minimum Spanning Tree**
2. **Prim’s Minimum Spanning Tree**
3. Boruvka’s Minimum Spanning Tree
4. **Dijkastra’s Shortest Path Algorithm**
5. Dial’s Algorithm
6. Minimum cost to connect all cities
7. Max Flow Problem Introduction
8. Number of single cycle components in an undirected graph

**Approximate Greedy Algorithm for NP Complete:**

1. Set cover problem
2. Bin Packing Problem
3. Graph Coloring
4. K-centers problem
5. Shortest superstring problem
6. Approximate solution for Travelling Salesman Problem using MST

**Greedy for Special cases of DP:**

1. **Fractional Knapsack Problem**
2. Minimum number of coins required

# Dynamic programming Approach

It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:

* 1. It breaks down the problem into a sub problem to find the optimal solution.
  2. After breaking down the problem, it finds the optimal solution out of these sub problems.
  3. Stores the result of the sub problems is known as memorization.
  4. Reuse the result so that it cannot be recomputed for the same sub problems.
  5. Finally, it computes the result of the complex program.

**Approaches of dynamic programming**

1. **Top-down approach**: The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.
2. B**ottom-Up approach:** The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

The following computer problems can be solved using dynamic programming approach −

1. Fibonacci number series
2. **Knapsack problem**
3. Tower of Hanoi
4. All pair shortest path by Floyd-Warshall
5. Shortest path by Dijkstra
6. Project scheduling
7. Binomial Coefficient
8. Coin change problem
9. **Floyd Warshall Algorithm**
10. **Bellman–Ford Algorithm**

# Branch and Bound approach:

The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.

# Randomized approach:

As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm? In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.