FACE

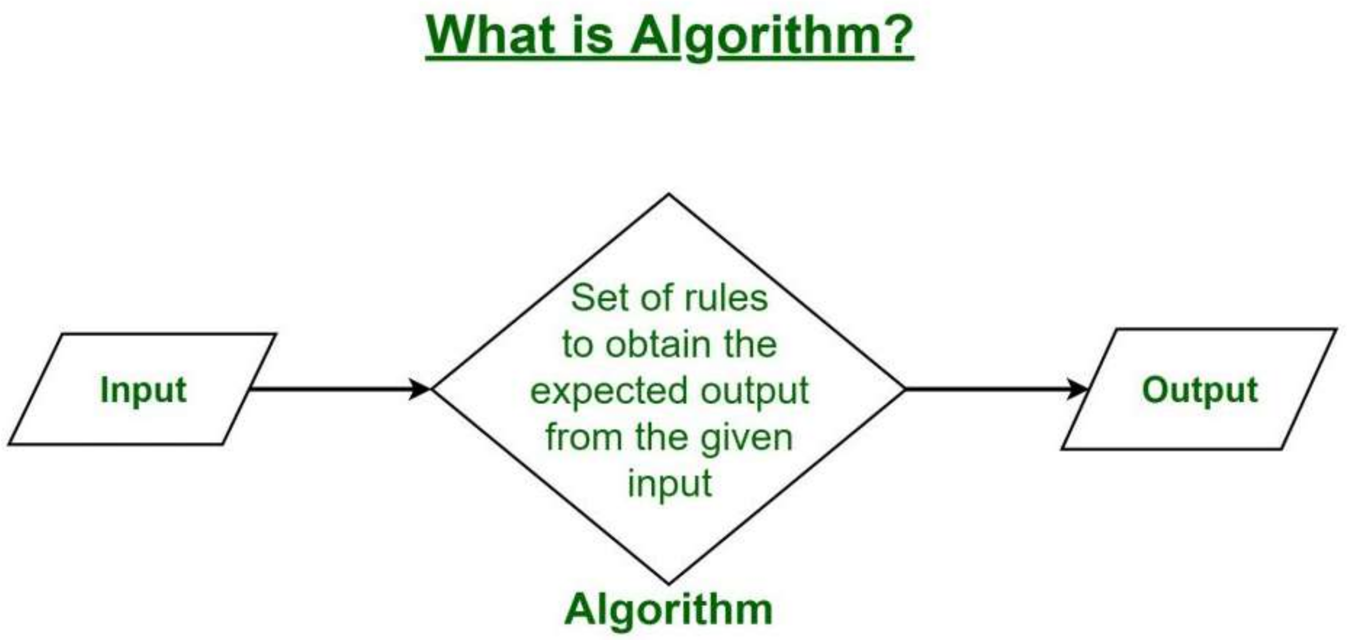
Algorithms

Algorithm

- The word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”.

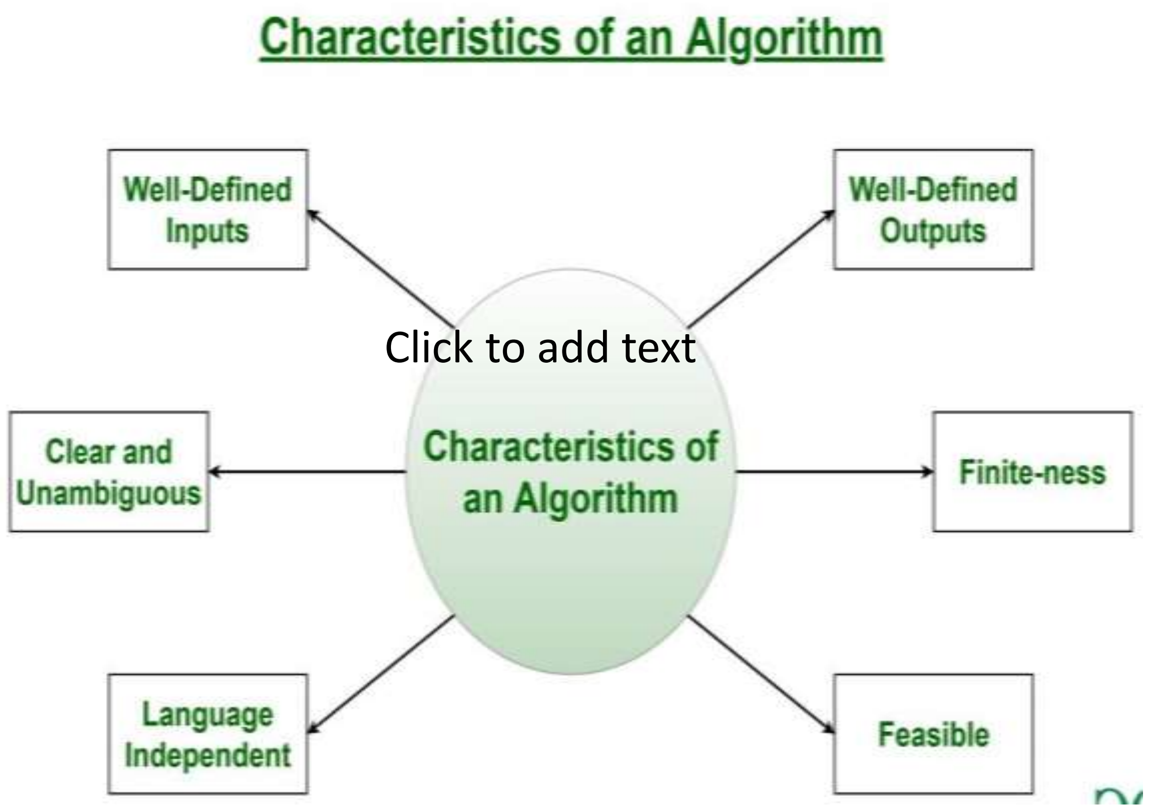
- Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

Algorithm


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Algorithm
Output

FACE

Algorithm



Click to add text

Algorithm

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.

- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.

Algorithm

•**Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

•**Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.

•**Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

FACE

Advantages:

- It is easy to understand.

- Algorithm is a step-wise representation of a solution to a given problem.

- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

FACE

Disadvantages:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Design:

In order to write an algorithm, following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm.
2. The **constraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem the is solved.
5. The **solution** to this problem, in the given constraints.

Design:

Step 1: Fulfilling the pre-requisites

- **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
- **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
- **The input to be taken to solve the problem:** The three numbers to be added.
- **The output to be expected when the problem the is solved:** The sum of the three numbers taken as the input.
- **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Design:

Step 2: Designing the algorithm

Algorithm to add 3 numbers and print their sum:

- START
- Declare 3 integer variables num1, num2 and num3.
- Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
- Declare an integer variable sum to store the resultant sum of the 3 numbers.
- Add the 3 numbers and store the result in the variable sum.
- Print the value of variable sum
- END

Design:

Step 3: Testing the algorithm by implementing it

Inorder to test the algorithm, let’s implement it

```java
import java.util.Scanner;
class Main
{
    public static void main(String args[])
    {
      Scanner obj = new Scanner(System.in);
      int a,b,c;
      System.out.println(“Enter the a”);
      a=s.nextInt();
      System.out.println(“Enter the b”);
      b=s.nextInt();
      System.out.println(“Enter the c”);
      c=s.nextInt();
      int sum=a+b+c;
      System.out.println(sum);
    }
}
```

Priori Analysis:

- “Priori” means “before”. Hence, Priori analysis means checking the algorithm before its implementation.
- In this, the algorithm is checked when it is written in the form of theoretical steps.
- This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- This is done usually by the algorithm designer. It is in this method, that the Algorithm Complexity is determined.

FACE

Posterior Analysis:

- “Posterior” means “after”. Hence Posterior analysis means checking the algorithm after its implementation.
- In this, the algorithm is checked by implementing it in any programming language and executing it.
- This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc

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.

Space Complexity:

Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining following 2 components:

1. **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
2. **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Time Complexity:

Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate Time Complexity?

The time complexity of an algorithm is also calculated by determining following 2 components:

1. **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.
2. **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.