**DESING AND ANALYSIS OF ALGORITHM**

**UNIT 1**

**WHAT IS ALGORITHM? ALGORITHM BASICS**

The word [Algorithm](https://www.geeksforgeeks.org/fundamentals-of-algorithms/) 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.



It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output.
 The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

### What are the Characteristics of an Algorithm?

 ****

 As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instructions to be an algorithm, it must have the following characteristics:

* **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.
* **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.

**Advantages of Algorithms:**

* 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.

**Disadvantages of Algorithms:**

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

**How to Design an Algorithm?**

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.

Then the algorithm is written with the help of above parameters such that it solves the problem.
**Example:** Consider the example to add three numbers and print the sum.

* **Step 1: Fulfilling the pre-requisites**
As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
	1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
	2. **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
	3. **The input to be taken to solve the problem:** The three numbers to be added.
	4. **The output to be expected when the problem the is solved:** The sum of the three numbers taken as the input.
	5. **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.
* **Step 2: Designing the algorithm**
Now let’s design the algorithm with the help of above pre-requisites:
**Algorithm to add 3 numbers and print their sum:**
	1. START
	2. Declare 3 integer variables num1, num2 and num3.
	3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
	4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
	5. Add the 3 numbers and store the result in the variable sum.
	6. Print the value of variable sum
	7. END

**ALGORITHM SPECIFICATION**

An algorithm is defined as a finite set of instructions that, if followed, performs a particular task. All algorithms must satisfy the following criteria

Input. An algorithm has zero or more inputs, taken or collected from a specified set of objects.

Output. An algorithm has one or more outputs having a specific relation to the inputs.

Definiteness. Each step must be clearly defined; Each instruction must be clear and unambiguous.

Finiteness. The algorithm must always finish or terminate after a finite number of steps.

Effectiveness. All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length.

We can depict an algorithm in many ways.

* Natural language: implement a natural language like English
* Flow charts: Graphic representations denoted flowcharts, only if the algorithm is small and simple.
* Pseudo code: This pseudo code skips most issues of ambiguity; no particularity regarding syntax programming language.

Example 1: Algorithm for calculating factorial value of a number

Step 1: a number n is inputted

Step 2: variable final is set as 1

Step 3: final<= final \* n

Step 4: decrease n

Step 5: verify if n is equal to 0

Step 6: if n is equal to zero, goto step 8 (break out of loop)

Step 7: else goto step 3

Step 8: the result final is printed

## Recursive Algorithms

A recursive algorithm calls itself which generally passes the return value as a parameter to the algorithm again. This parameter indicates the input while the return value indicates the output.

Recursive algorithm is defined as a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is treated as the input for the next recursion. The repletion is in the self-similar fashion manner. The algorithm calls itself with smaller input values and obtains the results by simply accomplishing the operations on these smaller values. Generation of factorial, Fibonacci number series are denoted as the examples of recursive algorithms.

Example: Writing factorial function using recursion

intfactorialA(int n)

{

   return n \* factorialA(n-1);

}

**PERFORMANCE ANALYSIS**

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of:

* **Space Complexity.**
* **Time Complexity.**

**Space Complexity** of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components:

1. **Fixed Component**: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.
2. **Variable Component**: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Therefore the total space requirement of any algorithm 'A' can be provided as

**Space(A) = Fixed Components(A) + Variable Components(A)**

Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'. To identify the space complexity of any algorithm following steps can be followed:

1. Determine the variables which are instantiated by some default values.
2. Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
3. Generally the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.
4. Sometimes more complex measures of the interrelationships among the data items can used.

**Example:** Space Complexity

Algorithm Sum(number,size)\\ procedure will produce sum of all numbers provided in 'number' list

{

 result=0.0;

 for count = 1 to size do \\will repeat from 1,2,3,4,....size times

 result= result + number[count];

 return result;

}

In above example, when calculating the space complexity we will be looking for both fixed and variable components. here we have

Fixed components as 'result','count' and 'size' variable there for total space required is three(3) words.

Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). because this will decide the size of 'number' list and will also drive the for loop. therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').

therefore the space complexity can be written as **Space(Sum) = 3 + n;**

**Time Complexity** of an algorithm(basically when converted to program) is the amount of computer time it needs to run to completion. The time taken by a program is the sum of the compile time and the run/execution time . The compile time is independent of the instance(problem specific) characteristics. following factors effect the time complexity:

1. Characteristics of compiler used to compile the program.
2. Computer Machine on which the program is executed and physically clocked.
3. Multiuser execution system.
4. Number of program steps.

Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:

**Time(A) = Fixed Time(A) + Instance Time(A)**

Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like

* comments count as zero steps,
* an assignment statement which does not involve any calls to other algorithm is counted as one step,
* for iterative statements we consider the steps count only for the control part of the statement etc.

Therefore to calculate total number program of program steps we use following procedure. For this we build a table in which we list the total number of steps contributed by each statement. This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed. This procedure is explained using an example.

**Example**: Time Complexity

In above example if you analyze carefully frequency of "for count = 1 to size do" it is 'size +1' this is because the statement will be executed one time more die to condition check for false situation of condition provided in for statement. Now once the total steps are calculated they will resemble the instance characteristics in time complexity of algorithm. Also the repeated compile time of an algorithm will also be constant every time we compile the same set of instructions so we can consider this time as constant 'C'. Therefore the time complexity can be expressed as: **Time(Sum) = C + (2size +3)**

So in this way both the Space complexity and Time complexity can be calculated. Combination of both complexity comprises the Performance analysis of any algorithm and can not be used independently. Both these complexities also helps in defining parameters on basis of which we optimize algorithms.

## ASYMPTOTIC NOTATIONS

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.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

## Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.



 Big-O gives the upper bound of a function

O(g(n)) = { f(n): there exist positive constants c and n0

 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst case scenario.

## Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.



 Omega gives the lower bound of a function

Ω(g(n)) = { f(n): there exist positive constants c and n0

 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

## 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 bounds the function within constants factors

**Asymptotic Notations:** For every algorithm corresponding to efficiency analysis, we have three basic cases :

* Best Case
* Average Case
* Worst Case

And there are 5 notations to resemble them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | Asymptotic Notation Name | Symbol Used |  Efficiency Case Resemblance |   Examples |
|  1 |      Theta notation |  Θ |  Average Case (Tightly Bound) |  Θ(1), Θ(n), Θ(n 2 ), Θ(n 3 ), Θ(n n ), Θ(log n), Θ(n log n) |
|  2 |      Big oh notation |  O |  Worst Case (Tightly Bound) |  O(1), O(n), O(n 2 ), O(n 3 ), O(n n ), O(log n), O(n log n) |
|  3 |      Omega notation |  Ω |  Best Case (Tightly Bound) |  Ω(1), Ω(n), Ω(n 2 ), Ω(n 3 ), Ω(n n ), Ω(log n), Ω(n log n) |
|  4 |      Little oh notation |  o |  Worst Case (Loosely Bound) |  o(1), o(n), o(n 2 ), o(n 3 ), o(n n ), o(log n), o(n log n) |
|  5 |     Little omega notation |  ω |  Best Case (Loosely Bound) |  ω(1), ω(n), ω(n 2 ), ω(n 3 ), ω(n n ), ω(log n), ω(n log n) |

## ****WHAT IS A RANDOMIZED ALGORITHM?****

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). And in [Karger’s algorithm](https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/), we randomly pick an edge.

Randomizsed Algorithm

Randomized algorithm are explain in following below topics:

 1.Basic probability theory

 2.randomized algorithm :an informal description

 3.identifying the repeated element

 4.primarily testing

**Basic of probability theory:**

Probability theory is a main conceptual experiments

**Examples:**

 Tossing a coin ten times

 Rolling a die three times

 Playing a lottery

 Gambling

 Piking

 A ball from white and red balls,etc,.

**Sample point:**

 Each possible outcome of an experimental is called sample point

**Sample space(S)**

 The set of all possible outcome is known as the sample space S.S is finite

**Event(E)**

An event E is a subset of the S. if the S consists of n sample points, then 2n possible events

**1.probablity**

 The probability of an event $\frac{(E)}{(S)}$ , where S is the sample space.

**2.Mutual Exclusion:**

 Two events E1 and E2 are said to be mutually exclusive. Its not have a common Si.e1.e2

**3.Conditional probability**

The conditional probability of e1 given e2 denoted by prob(E1|E2), is defined as

**4.Independence**

Two events e and e said to be independence

 If prob.[e1 e2}=prob.[e1]\*prob[e2].

**5. Random variable**

 Let S be the sample of an experiment. A random variable os s is as function that maps the elements of S to the set of real numbers

**Primarily testing**

Any integer greater than one in sidfe to be a prime if its only division are 1 the integer itself

 Given an integer n, the problem of deciding whether n is a prime is known ass primarily testing

**Theorem1**

 If n is prime then an-1=1(mod n)

 For any integer

**Theorem2**

The equation x2=1 (mod n) has exactly two solutions nam,ely 1 and n-1, if n is prime

**Algorithm:**

primeO(n,$α$)

{

Q:n-1;

{

m;=q;y;=1

a:=random()mod q+1;

z;=a;

while(m>0) do

{

While (m mod 2=0)do

(

 Z;=z2 od n; m:=[m/2];

}

M:=m-1;y:=(y\*z) mod n;

}

If(y≠1) then return false;

}

Return true

}