UNIT - I

Introduction to Algorithms

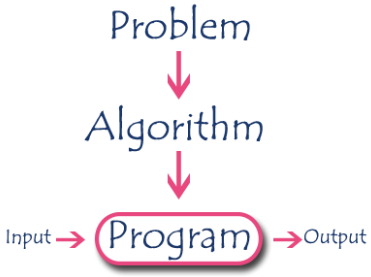
Introduction to Algorithms: Algorithms, Pseudo code for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst Case Complexities, Analyzing Recursive Programs.

1.1 What is an algorithm?

- An algorithm is a step by step procedure to solve a problem. In normal language, the algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

- Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution.
- Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.



- Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:
  - Step 1: First, we will cut the lemon into half.
  - Step 2: Squeeze the lemon as much you can and take out its juice in a container.
  - Step 3: Add two tablespoon sugar in it.
  - Step 4: Stir the container until the sugar gets dissolved.
  - Step 5: When sugar gets dissolved, add some water and ice in it.
  - Step 6: Store the juice in a fridge for 5 to minutes.
  - Step 7: Now, it's ready to drink.
- The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Example:

write an algorithm to add two numbers entered by the user.

The following are the steps required to add two numbers entered by the user:

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 and store the result in the sum variable, i.e., sum=a+b.

Step 5: Print sum

Step 6: Stop

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Issues in study of Algorithms:

- There are several issues in the study of algorithms and those are:
    1. **How to devise algorithms:**
        - The creation of algorithms is a logical activity and one can't automate it.
        - But there are certain algorithm design strategies, and using these strategies one can create many useful algorithms.
        - Hence mastering of such design strategies is an important activity in study of Design and Analysis of Algorithms.
    2. **How to validate algorithms:**
        - The next step after creation of algorithms is validating them.

- The process of checking whether an algorithm computes the correct answer for all possible inputs is called algorithm validation.
- The purpose of validation of algorithm is to find whether algorithms works properly without being dependent upon programming languages.
- Once validation of algorithm is done, a program can be written using corresponding algorithm.

3. **How to analyze algorithms:**
   - Analysis of algorithm is a task of determining how much computing time and storage is required by an algorithm.
   - Analysis of algorithm is also called performance analysis.
   - This analysis is based on mathematics and a judgment is often needed about better algorithm when two algorithms get compared.
   - The behavior of algorithm in best, worst and average case needs to obtained.
4. **How to test a program:**
   - After finding an efficient algorithm, it is necessary to test that the program written using the efficient algorithm behaves properly or not.
   - Testing a program is an activity that can be carried out in 2 phases:
     1. Debugging
     2. Profiling (Performance Analysis)
   - While debugging a program, it is checked whether program produces faulty results for a valid set of input, and if it is found then the program has to be correct.
   - Profiling is a process of measuring time and space required by a program for a valid set of inputs.

Algorithm Specification:

Algorithm can be described in three ways.

1. **Natural language like English:**
   When this way is chooses care should be taken, we should ensure that each & every statement is definite.
2. **Graphic representation called flowchart:**
   This method will work well when the algorithm is small& simple.
3. **Pseudo-code Method:**
   In this method, we should typically describe algorithms as program, which resembles language like Pascal & C.

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

**********END**********

1.2 Pseudo code for expressing algorithms

What is Pseudo code?

Programming language description of algorithm is called as pseudo code. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

Pseudo-Code Conventions:

We present most of our algorithms using a pseudo-code that resembles C and Pascal.

1. Algorithm consists of heading and body. The heading consists of name of the procedure and parameter list. The syntax is:

Algorithm: Name_of_Procedure(Parameter_1, …, Parameter_n)

2. Comments begin with // and continue until the end of line.
3. Blocks are indicated with matching braces {and}.
4. The delimiter **;** is used at the end of the each statement.
5. An identifier begins with a letter. It is not necessary to write data types for variables.
6. Assignment of values to variables is done using the assignment statement.
   <Variable> := <expression>;
7. There are two Boolean values TRUE and FALSE.
   → Logical Operators AND, OR, NOT
   →Relational Operators <, <=,>,>=, =, !=
8. Elements of multidimensional arrays are accessed using [and].
   For Example: A[i, j]
9. The following looping statements are employed.

   While Loop:

```
While < condition > do
{
        <statement-1>
                …
                …
                …
        <statement-n>
}
```

   For Loop:

```
for variable := value-1 to value-2 step step_value do
{
        <statement-1>
                …
                …
                …
        <statement-n>
```

}

repeat-until:

repeat
<statement-1>
…
…
…
<statement-n>
until <condition>

10. A conditional statement has the following forms.

if:

if <condition> then <statement>

if else:

if <condition> then <statement-1>
else <statement-2>

Case statement:

Case
{
: <condition-1> : <statement-1>
…
…
…
: <condition-n> : <statement-n>
: else : <statement-n+1>
}

11. Input and output are done using the instructions read & write.

Example:

1. Write a pseudo code to perform addition of two numbers
Pseudo Code:
Step-1: Start
Step-2: Input A
Step-3: Input B
Step-4: total := A + B
Step-5: Print Total
Step-6: Stop

Assignment:

1. **Design algorithm to find sum of n numbers.**
2. **Design algorithm to find given number is even or odd.**
3. **Design algorithm to sort the given list of integers.**
4. **Design algorithm to find factorial of a given number.**
5. **Design algorithm to find multiplication of two matrices.**

********END*******

1.3 Performance Analysis-Space complexity, Time complexity

- Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.
- We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like **memory required by that algorithm**, the **execution speed** of that algorithm, easy to understand, easy to implement, etc.,
- The efficiency of an algorithm can be decided by measuring the performance of algorithm.
- Performance analysis of an algorithm can be defined as follows...

  Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.
- We can measure the performance of an algorithm by computing amount of time(Time Complexity) and space requirement(Space Complexity).

Space Complexity:

- Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...
  1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
  2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
  3. **Data Space:** It is the amount of memory used to store all the variables and constants.
- **Note -** When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.
  That means we calculate only the memory required to store Variables, Constants, Structures, etc.,
  - The space complexity of an algorithm is the amount of memory required by an algorithm to run.
  - To compute the space complexity the following elements are used:
    1. Constants
    2. Instance Characteristics

  1. Constants:
  - A fixed part that is independent of the characteristics of the inputs and outputs.
  - This part includes the instruction space, space for simple variables and fixed size component variables, space for constants and so on.

  2. Instant Characteristics(S_P):
  - It is also called a dependant part or variable part
  - A variable part that consists of the space needed by component variables whose size id dependent in the particular problem instance being solved, the space need by referenced variables and the recursion stack space.

Formula:

- The space requirement S(P) of any algorithm 'P' is written as:

$$S(P) = c + S_P$$

Where, c is a constant part

S_P is instance characteristics or dependent or variable part

- Consider various examples to compute space complexity:
- **Example:**

1. Compute space complexity of a+b+b*c+(a+b-c)/(a+b)+4.0

Solution:

```
Algorithm abc(a, b, c)
{
      return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- Here, the above problem is characterized by the values a, b, & c.
- Let assume that one word is need to store the values of each a, b, & c.
- Therefore c = 3, S_P = 0
  - ➔ S(P) = 3
  - ➔ 3 units of memory is fixed for any input value of 'a',b,c. This space complexity is said to be *Constant Space Complexity*.

2. Find the space complexity for the algorithm to find sum of list of values.

Solution:

```
Algorithm sum(a, n)
{
     sum := 0.0;
    for i:=0 to n do
         sum := sum + a[i];
   return sum;
}
```

- The above algorithm is characterized by 'n': the number of elements to be summed.
- The space needed by 'n' is one word. Since it is integer.
- 'a' is an array which depends on n value. So size of 'a' is n i.e., S_p = n.
- Space for 'i' is one.
- Space for 'sum' is one.
- Therefore
  - ➔ $S(P) = c + S_P$
  - ➔ $S(P) = (3) + n$
  - ➔ $S(P) \geq n + 3$
  - ➔ Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be *Linear Space Complexity*.

3. Find the space complexity for the algorithm to find sum of list of values using recursion.

Solution:

```
Algorithm Recursive_sum(a, n)
```

```
{
     if (n <= 0 ) then
          return 0.0;
     else
          return Recursive_sum(a, n-1) + a[n]
}
```

- For algorithm Recurive_sum, recursion stack space includes space for the formal parameters, the local variables and the return address.
- Let assume that return address requires only one word of memory.
- Each call to Recurive_sum requires at least 3 words (space for 'n', return address, pointer to array 'a')
- Since the depth of recursion is n+1.
- The recursion stack space needed is:
  ➔ $S(P) \geq 3(n+1)$

Time Complexity:

- The time complexity of an algorithm is the amount of computer time it needs to run to compilation.
- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)
- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation. Therefore, we concern that runtime of the program. This rum time is denoted by t_p(instance characteristics).
- Let us take an example, that determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code form. So, we would obtain an expression for $t_p(n)$ of the form:

$$t_p(n) = C_a Add(n) + C_s Sub(n) + C_m Mul(n) + C_d Div(n) + \ldots$$

Where 'n' denotes the instance characteristics and C_a, C_s, C_m, C_d *and so on* denotes the time needed for addition, subtraction, multiplication, division and so on. And Add, Sub, Mul, Div and o on are the functions to perform addition, subtraction, multiplication, division and so on.

Using Frequency Count Method:

- In this method first identify which steps are executed and which are not executed, this is called steps for execution(i.e. **s/e**)

- For this steps for execution calculate the frequency i.e. how many times the steps are executed
  - To find the total steps Multiply steps for execution(s/e) and frequency and finally add all total steps

Example:

1. Find the space complexity for the algorithm to find sum of list of values.

Solution:

```
Algorithm sum(a, n)
{
     sum := 0.0;
    for i:=0 to n do
        sum := sum + a[ i ];
   return sum;
}
```

Step	Statement	s/e	Frequency	Total steps
1	**Algorithm sum(a, n)**	0	---	0
2	**{**	0	---	0
3	**sum := 0.0;**	1	1	1
4	**for i:=0 to n do**	1	n + 1	n + 1
5	**sum := sum + a[i];**	1	n	n
6	**return sum;**	1	1	1
7	**}**	0	---	0
			Total	2n + 3

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 **Algorithm** RSum(a, n)	0	—	—	0	0
2 **{**					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1 + x$	0	1	0	$1 + x$
7 **}**	0	—	—	0	0
Total				2	$2 + x$

$x = t_{\text{RSum}}(n-1)$

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	—	0
2 **{**	0	—	0
3 **for** $i := 1$ **to** m **do**	1	$m + 1$	$m + 1$
4 **for** $j := 1$ **to** n **do**	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 **}**	0	—	0
Total			$2mn + 2m + 1$

Table 1.3 Step table for Algorithm 1.11

*******END*******

1.4 Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation

- 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.
- **Note -** In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm
- For example, consider the following time complexities of two algorithms...

  Algorithm 1 : $5n^2 + 2n + 1$

  Algorithm 2 : $10n^2 + 8n + 3$

  - Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. '**n**' value).
  - Here, for larger value of '**n**' the value of most significant terms ($5n^2$ and $10n^2$) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**).
  - So for larger value of '**n**' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.
- Usually, the time required by an algorithm comes under three types:
  - **Worst case:** It defines the input for which the algorithm takes a huge time.
  - **Average case:** It takes average time for the program execution.
  - **Best case:** It defines the input for which the algorithm takes the lowest time
- There are 4 types of Asymptotic Notations. They are:
  1. Big Oh Notation (O)
  2. Big Omega Notation (Ω)
  3. Theta Notation (Θ)
  4. Little Oh Notation (o)
  5. Little Omega Notation(ω)

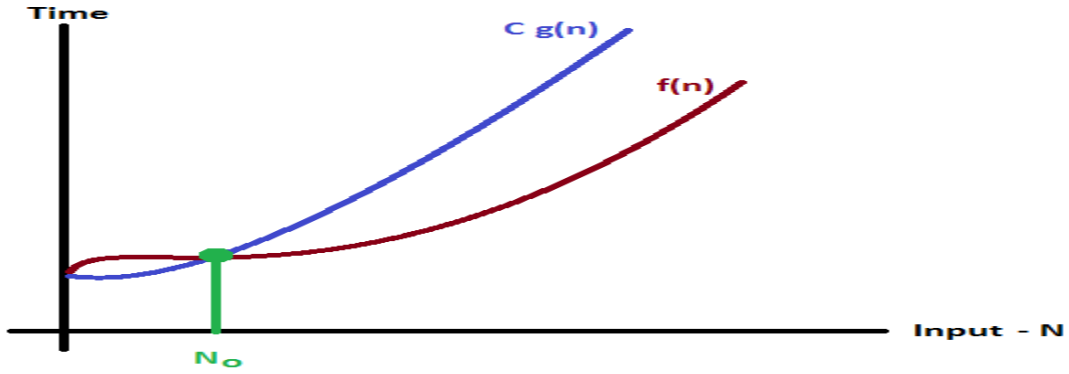
Big - Oh Notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- Big - Oh notation describes the worst case of an algorithm time complexity.
- Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1 where positive constant C and n_0. Then we can represent f(n) as O(g(n)).

$$f(n) = O(g(n))$$

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

- We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n.
- We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n.
- As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n)

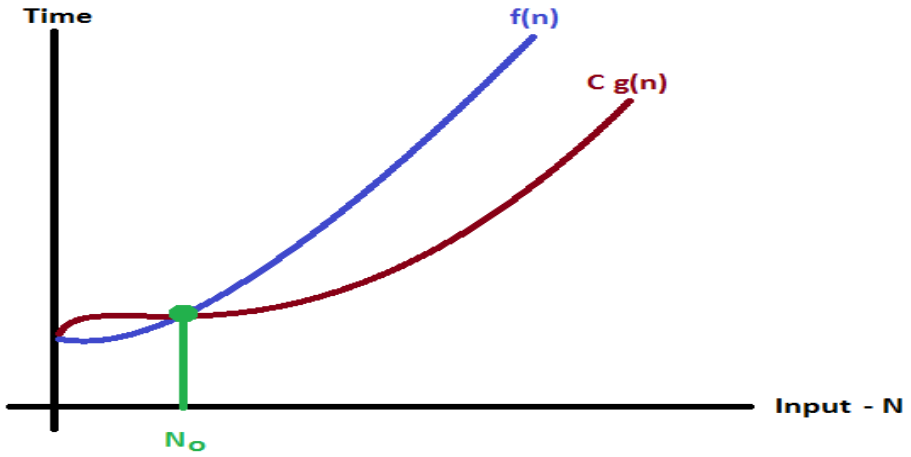
Big-Omege Notation (Ω)

- Big- Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- Big- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm

- Big- Omega Notation can be defined as follows...
  - **Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).**

$$f(n) = \Omega(g(n))$$

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

If f(n) = 2n+3, g(n) = n,
Is f(n)= **Ω** (g(n))?
It must satisfy the condition:
f(n)>=c.g(n)
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.
2n+3>=c*n
Suppose c=1
2n+3>=n (This equation will be true for any value of n starting from 1).
Therefore, it is proved that g(n) is big omega of 2n+3 function.

Big - Theta Notation (Θ)

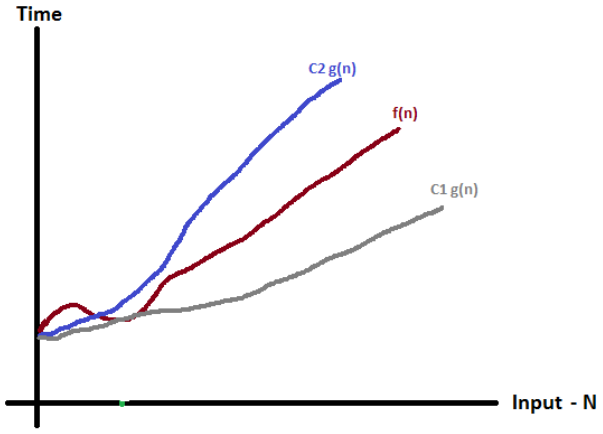
- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average 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.

- Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

$$f(n) = \Theta(g(n))$$

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent f(n) as Θ(g(n)) then it must satisfy C1*g(n) <= f(n) <= C2*g(n)
for all values of C1 > 0, C2 > 0 and n0>= 1
C1 g(n) <= f(n) <= C2 g(n)
⇒C1* n <= 3n + 2 <= C2* n
Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 2.
By using Big - Theta notation we can represent the time compexity as follows...

$$3n + 2 = \Theta(n)$$

Little Oh Notation(o):

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort, it can also be a loose upper-bound. To make its role as a tight upper-bound more clear, "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

Definition: (Little–o, o()): Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is o(g(n)) (or f(n) ∈ o(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) < c ∗ g(n) for every integer n ≥ n0.

Note that in this definition, the set of functions $f(n)$ are strictly smaller than $cg(n)$, meaning that little-o notation is a stronger upper bound than big-O notation. In other words, the little-o notation does not allow the function $f(n)$ to have the same growth rate as $g(n)$.

Intuitively, this means that as the n approaches infinity, $f(n)$ becomes insignificant compared to $g(n)$. In mathematical terms:

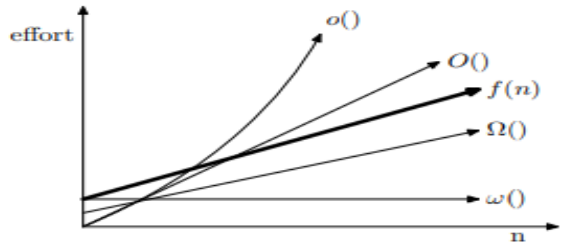
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Little Omega Notation(ω):

- "Big-Omega" (Ω()) is the tight lower bound notation, and "little-omega" (ω()) describes the loose lower bound.
- **Definition**: (Little–Omega, ω()): Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c · g(n) for every integer n ≥ n0.
- Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth. We use ω notation to denote a lower bound that is not asymptotically tight.

Relationship between Big Oh, Little Oh, Big Omega and Little Omega:

This graph should help you visualize the relationships between these notations:



*****END******

Time Complexity of an Algorithm Using O Notation:

Various meanings associated with O are:

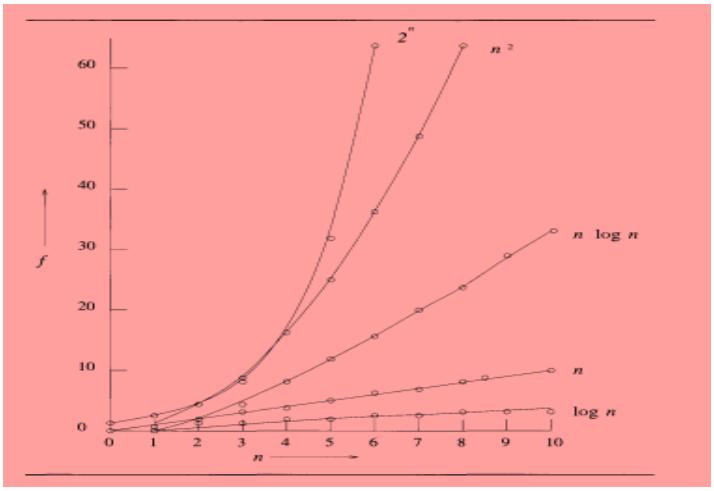
1. O(1) ➔ Constant Computing Time
2. O(n) ➔ Linear
3. O(n^2) ➔ Quadratic
4. O(n^3) ➔ Cubic
5. O(2^n) ➔ Exponential
6. O($\log n$) ➔ Logarithmic

- The relationship among these computing time is:

  O(1) < O($\log n$) < O(n) < O(n $\log n$) < O(n^2) < O(2^n)

- To get a feel for how the various functions grow with n, you are advised to study the blow Table and Figure very closely. We can observe that the function 2^n grows very rapidly with n.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296



*****END*****

Polynomial Vs Exponential Algorithms:

Polynomial Algorithm:

- An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input.
- Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time.
- Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

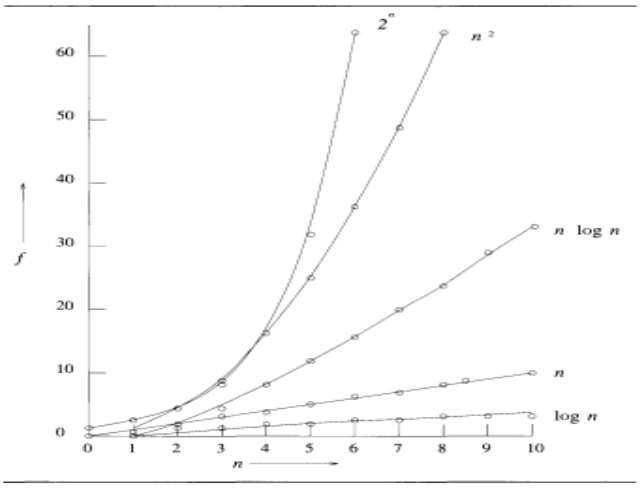
Exponential Algorithm:

- An algorithm is said to be exponential algorithm, if $T(n)$ is upper bounded by $2^{poly(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{nk})$ for some constant k.

Algorithms which have exponential time complexity grow much faster than polynomial algorithms:

n Value	Exponential (2^n)	Polynomial (n^2)
1	2	1
2	4	4
3	8	6

4	16	16
5	32	25
6	64	36



*****END*****

Average, Best and Worst Case Complexities:

Best Case Complexity:

- If an algorithm takes minimum amount of time to run for a specific set of inputs, then it is called best case time complexity
- It can represented using Omega Notation (Ω).
- For example, while searching an element by using linear search we get the desired element at first place then it is called as best case time complexity.

Worst Case Complexity:

- If an algorithm takes maximum amount of time to run for a specific set of inputs, then it is called worst case time complexity
- It can represented using Big Oh Notation (O).
- For example, while searching an element by using linear search we get the desired element at the end of the list or not presented in the list then it is called as best case time complexity.

Average Case Complexity:

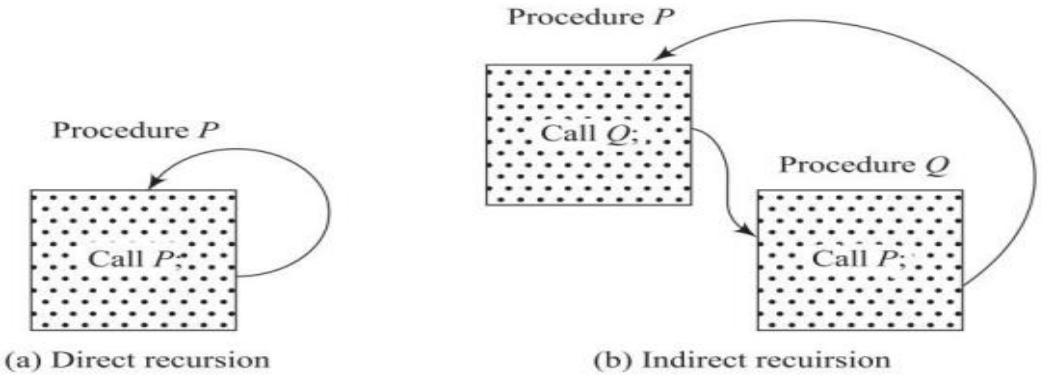
- If an algorithm takes average amount of time to run for a specific set of inputs, then it is called average case time complexity
- It can represented using Theta Notation (Θ).
- For example, while searching an element by using linear search we get the desired element in anywhere from the list except first and last positions, then it is called as best case time complexity.

*****END*****

1. Analyzing Recursive Programs:

1.1. Recursive Algorithms:

- A function or algorithm calls its self is known as recursion and that function is known as recursive function.
- An algorithm call itself is **direct recursive**.
- An algorithm 'A' is said to be **indirect recursive** if it calls another algorithm which in turn calls 'A'.
- The recursive mechanism is more powerful and represents complex process very clearly.
- The following example shows how to develop a recursive algorithm:



(a) Direct recursion (b) Indirect recuirsion

Example:

1. Write a pseudo code to find factorial of a number.

Pseudo Code:

```
Algorithm Factorial(n)
{
    if n == 0 or n == 1 then
        return 1
    else
       return n * factorial(n-1)
}
```

1.2. Space Complexity of Recursive Algorithms:

Find the space complexity for the algorithm to find sum of list of values using recursion.

Solution:

```
Algorithm Recursive_sum(a, n)
{
```

```
    if (n <= 0 ) then
        return 0.0;
    else
        return Recursive_sum(a, n-1) + a[n]
}
```

- For algorithm Recurive_sum, recursion stack space includes space for the formal parameters, the local variables and the return address.
- Let assume that return address requires only one word of memory.
- Each call to Recurive_sum requires at least 3 words (space for 'n', return address, pointer to array 'a')
- Since the depth of recursion is n+1.
- The recursion stack space needed is:

$$S(P) \geq 3(n+1)$$

1.3. Time Complexity of Recursive Algorithms:

Find the space complexity for the algorithm to find sum of list of values using recursion.

Solution:

Step	Statement	s/e	Frequency		Total	
			n<=0	n>0	n<=0	n>0
1	**Algorithm Recursive_sum(a, n)**	0	---	---	0	0
2	**{**	0	---	---	0	0
3	**if (n <= 0) then**	1	1	1	1	1
4	**return 0.0;**	1	1	0	1	0
5	**else**	0	---	---	0	0
6	**return Recursive_sum(a, n-1) + a[n]**	n+1	0	1	0	n+1
7	**}**	0	---	---	0	0
				Total	2	n+1

$$T(P) = \begin{cases} 2 \; when \; n \leq 0 \\ n+2 \; when \; n > 0 \end{cases}$$

1.4 Recurrence relation of recursive algorithms

- **A recurrence relation** is an equation that defines a sequence where any term is defined in terms of its previous terms**.(2 Marks)**
- The recurrence relation for the time complexity of some problems are given below:

Fibonacci Number

$T(N) = T(N-1) + T(N-2)$

Base Conditions: $T(0) = 0$ and $T(1) = 1$

Binary Search

$T(N) = T(N/2) + C$

Base Condition: $T(1) = 1$

Merge Sort

$T(N) = 2\,T(N/2) + CN$

Base Condition: $T(1) = 1$

Recursive Algorithm: Finding min and max in an array

$T(N) = 2\,T(N/2) + 2$

Base Condition: $T(1) = 0$ and $T(2) = 1$

Karastuba algorithm for fast multiplication

$T(N) = 3\,T(N/2) + CN$

Base Condition: $T(0) = 1$ and $T(1) = 1$

Quick Sort

$T(N) = T(i) + T(N-i-1) + CN$

- The time taken by quick sort depends upon the distribution of the input array and partition strategy. T(i) and T(N-i-1) are two smaller subproblems after the partition where i is the number of elements that are smaller than the pivot. **CN** is the time complexity of the partition process where **C** is a constant. .

1.5 Analyzing the Time Efficiency of Recursive Algorithms

- Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity and we need to solve those recurrence relations.
- There are mainly three ways of solving recurrences
  1. **Substitution Method:** The Substitution Method Consists of two main steps:
     - Guess the Solution.
     - Use the mathematical induction to find the boundary condition and shows that the guess is correct.
  2. **Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of the tree. Finally, we sum the work done at all levels.
  3. **Master Method:** Master Method is a direct way to get the solution. The master method works only for recurrences that can be transformed into the following type.

     $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

Substitution Method

Analysis of Recursive Algorithms or programs

Recurrence relation - formula for the time complexity :

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b) + f(n). & n>1 \end{cases}$$

Solve the recurrence relation for quick sort :

$(a=2,\ b=2\quad f(n)=n,\ T(1)=2)$

$$T(n) = \begin{cases} 2 = T(1) \\ 2T(n/2) + n \end{cases}$$

Sol → $T(n) = 2T(n/2) + \frac{n}{2}$ ⊖ a

Sub n with n/2

Step1:- $T(n) = 2T(n/2) + n \rightarrow (a)$

Sub $T(n/2)$ in a

$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$

$T(n) = 4T\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right) + n$

$T(n) = 4T(n/4) + (n+n)$

$T(n) = 4T(n/4) + 2n \rightarrow (b)$

$T(n) = 2T(n/2) + n$

$T(n/2) = 2T(n/4) + n/2$

$T(n/4) = 2T(n/8) + n/4$

Step2:- Sub $T(n/4)$ in b

$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$

$T(n) = 8T\left(\frac{n}{8}\right) + 4\left(\frac{n}{4}\right) + 2n$

$T(n) = 8T\left(\frac{n}{8}\right) + 3n \rightarrow (c)$

sub $T(n/8)$ in c

$T(n) = 8\left[2T\left(\frac{n}{16}\right) + \frac{n}{8}\right] + 3n$

$T(n) = 16T\left(\frac{n}{16}\right) + 8\left(\frac{n}{8}\right) + 3n$

$T(n) = 16T\left(\frac{n}{16}\right) + 4n$

step k:-

$T(n) = 2^4 T(n/2^4) + 4n \Rightarrow [2^k T(n/2^k) + kn]$

$T(n) = 2^k = n$

$k = \log 2^n$

$2^k = n$

$k = \log 2^n$

$T(n) = nT(n/n) + \log_2^n \cdot n$

$= nT(1) + \log_2^n \cdot n$

$T(1) = 2$

$= n(2) + n \cdot \log_2^n$

$T(n) = 2n + n \cdot \log_2^n$

The time complexity for quick sort is $O(n \log n)$.

3, Solve the recurrence relation for binary-search $a = 1$, $b = 2$

$f(n) = 1$, $T(1) = 1$

$$T(n) = \begin{cases} 1 \\ 1 \cdot T(n/2) + 1 \rightarrow a \end{cases}$$

$T(n) = T(n/2) + 1$

Sub n with $n/2$

$T(\frac{n}{2}) = T(\frac{n}{4}) + 1$

$T(n) = (T(\frac{n}{4}) + 1) + 1$

$T(n) = T(\frac{n}{2^2}) + 2 \rightarrow (b)$

$T(n) = T(n/2) + 1$

$T(n/2) = T(n/4) + 1$

$T(n/4) = T(n/8) + 1$

$T(n/8) = T(n/16) + 1$

Sub $T(\frac{n}{4})$ in b

$T(n) = T(\frac{n}{8}) + 3 \rightarrow (c)$

Sub $T(\frac{n}{8})$ in c

$T(n) = T(\frac{n}{16}) + 4 \rightarrow$

$T(n) = T(\frac{n}{2^4}) + 4$

$T(n) = T(\frac{n}{2^k}) + 4$

$k = 4$

$2^k = n$

$k = \log_2^n$

$$T(n) = T(n/n) + \log_2 n$$

$$= T(1) + \log_2 n \qquad T(1) = 1$$

$$= O(1) + \log_2 n = 1 + \log_2 n$$

The time complexity of binary search is $O(\log n)$.

Important Questions

1, What is an algorithm? Explain specifications of algorithm or pseudocode for expressing algorithms?

2, Explain about performance Analysis or space complexity and time complexity?

3, Explain briefly about Asymptotic Notations.

4, Write about polynomical Vs Exponential Algorithms?

5, Explain briefly about Analysis of best case, Average case, worst case time complexity.

6, Apply reccurrance relation to find time complexity of Recursive programs?