UNIT 1 BASICS OF AN ALGORITHM

Structure **Page Nos.**

1.0 INTRODUCTION

The word "***Algorithm***" comes from the Persian author ***Abdullah Jafar Muhammad ibn Musa Al-khowarizmi*** in ninth century, who has given the definition of algorithm as follows:

- An Algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- **An Algorithm** is a well defined computational procedure that takes input and produces output.
- An Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output.

Any Algorithm must satisfy the following criteria (or Properties)

1. **Input**: It generally requires finite no. of inputs.
2. **Output:** It must produce at least one output.
3. **Uniqueness**: Each instruction should be clear and unambiguous
4. **Finiteness:** It must terminate offer a finite no. of steps.

Analysis Issues of algorithm is

1. **WHAT DATA STRUCTURES TO USE!** (Lists, queues, stacks, heaps, trees, etc.)
2. **IS IT CORRECT!** (All or only most of the time?)
3. **HOW EFFICIENT IS IT!** (Asymptotically fixed or does it depend on the inputs?)
4. **IS THERE AN EFFICIENT ALGORITHM!!** (i.e. P = NP or not)

1.1 OBJECTIVES

After studying this unit, you should be able to:

- Understand the definition and properties of an Algorithm
- Pseudo-code conventions for algorithm
- Differentiate the fundamental techniques to design an Algorithm
- Understand the Time and space complexity of an Algorithm
- Use on Asymptotic notations O (Big 'Oh') Ω (Big 'Omega') and Θ (Theta) Notations
- Define a Recurrence and various methods to solve a Recurrence such as Recursion tree or Master Method.

1.2 ANALYIS AND COMPLEXITY OF ALGORITHMS

In this unit we will examine several issues related to basics of algorithm: starting from how to write a Pseudo-code for algorithm, mathematical induction, time and space complexity and Recurrence relations. Time and space complexity will be further discussed in detail in unit 2.

"**Analysis of algorithm**" is a field in computer science whose overall goal is an understanding of the complexity of algorithms (in terms of time Complexity), also known as ***execution time*** & storage (or space) requirement taken by that algorithm.

Suppose **M** is an algorithm, and suppose n is the size of the input data. The time and space used by the algorithm M are the two main measures for the efficiency of **M.** The time is measured by counting the number of key operations, for example, in case of sorting and searching algorithms, the number of comparisons is the number of key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The ***complexity*** of an algorithm M is the *function f(n),* which give the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. In general the term "***complexity***" given anywhere simply refers to the running time of the algorithm. There are 3 cases, in general, to find the complexity function f(n):

1. **Best case:** The minimum value of $f(n)$ for any possible input.
2. **Worst case:** The maximum value of $f(n)$ for any possible input.
3. **Average case:** The value of $f(n)$ which is in between maximum and minimum for any possible input. Generally the Average case implies the ***expected value*** of $f(n)$.

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The Average case also uses the concept of probability theory. Suppose the numbers $N_1, N_2, \ldots\ldots N_k$ occur with respective probabilities $p_1, p_2, \ldots\ldots p_k$. Then the expectation or average value of E is given by $E = N_1p_1 + N_2p_2 + \cdots\ldots\ldots + N_kp_k$

To understand the Best, Worst and Average cases of an algorithm, consider a linear array $A[1 \ldots . n]$, where the array A contains n-elements. Students may you are having some problem in understanding. Suppose you want *either* to find the location LOC of a given element (say x) in the given array A or to send some message, such as LOC=0, to indicate that x does not appear in A. Here the linear search algorithm solves this problem by comparing given x, one-by-one, with each element in A. That is, we compare x with A[1], then A[2], and so on, until we find LOC such that $x = A[LOC]$.

Algorithm: (**Linear search)**
/* **Input:** A linear list A with n elements and a searching element x.
Output: Finds the location LOC of x in the array A (by returning an index)
or return LOC=0 to indicate x is not present in A. */

1. [Initialize]: Set K=1 and LOC=0.
2. Repeat step 3 and 4 $while(LOC == 0 \;\&\&\; K \leq n)$
3. $if(x == A[K])$
4. {
5. LOC=K
6. $K = K + 1;$
7. }
8. $if(LOC == 0)$
9. $printf(" x \; is \; not \; present \; in \; the \; given \; array \; A);$
10. $else$
11. $printf(" x \; is \; present \; in \; the \; given \; array \; A \; at \; location \; A[LOC]);$
12. Exit [end of algorithm]

Analysis of linear search algorithm

The complexity of the search algorithm is given by the number C of comparisons between x and array elements A[K].

Best case: Clearly the best case occurs when x is the first element in the array A. That is $x = A[LOC]$. In this case $C(n) = 1$

Worst case: Clearly the worst case occurs when x is the last element in the array A or x is not present in given array A (to ensure this we have to search entire array A till last element). In this case, we have
$C(n) = n$.

Average case: Here we assume that searched element x appear array A, and it is equally likely to occur at any position in the array. Here the number of comparisons can be any of numbers $1,2,3, \dots \dots, n$, and each number occurs with the probability $p = \frac{1}{n}$. then

$$C(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \cdots \dots \dots + n.\frac{1}{n}$$
$$= (1 + 2 + \cdots \dots \dots \dots + n).\frac{1}{n}$$
$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

It means the average number of comparisons needed to find the location of x is approximately equal to half the number of elements in array A. From above discussion, it may be noted that the complexity of an algorithm in the average case is much more complicated to analyze than that of worst case. Unless otherwise stated or implied, we always find and write the complexity of an algorithm in the worst case.

There are three basic asymptotic $(i.e.\ n \to \infty)$ notations which are used to express the running time of an algorithm in terms of function, whose domain is the set of natural numbers N={1,2,3,…..}. These are:

- $O\ (Big-'Oh')$ [This notation is used to express Upper bound (maximum steps) required to solve a problem]
- Ω $(Big-'Omega')$ [This notation is used to express Lower bound i.e. minimum (at least) steps required to solve a problem]
- Θ ('Theta') Notations. [Used to express both Upper & Lower bound, also called tight bound]

Asymptotic notation gives the *rate of growth*, i.e. performance, of the run time for "*sufficiently large input sizes"* $(i.e.\ n \to \infty)$ and is **not** a measure of the *particular* run time for a specific input size (which should be done empirically). O-notation is used to express the Upper bound (worst case); Ω-notation is used to express the Lower bound (Best case) and Θ- Notations is used to express both upper and lower bound (i.e. Average case) on a function.

We generally want to find either or both an asymptotic *lower bound* and *upper bound* for the growth of our function. The *lower bound* represents the *best case* growth of the algorithm while the *upper bound* represents the *worst case* growth of the algorithm.

1.3 BASIC TECHNIQUES FOR DESIGN OF EFFICIENT ALGORITHMS

There are basically 5 ***fundamental techniques*** which are used to design an algorithm efficiently:

1. Divide-and-Conquer
2. Greedy method
3. Dynamic Programming
4. Backtracking
5. Branch-and-Bound

In this section we will briefly describe these techniques with appropriate examples.

1. ***Divide & conquer technique*** is a top-down approach to solve a problem. The algorithm which follows divide and conquer technique involves 3 steps:
   - **Divide** the original problem into a set of sub problems.
   - **Conquer** (or Solve) every sub-problem individually, recursive.
   - **Combine** the solutions of these sub problems to get the solution of original problem.

2. ***Greedy technique*** is used to solve an optimization problem. An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or minimized (known as **objective function**) w. r. t. some constraints or conditions. Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function. That is, it makes a ***locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.*** The greedy algorithm does not always guaranteed the optimal solution but it generally produces solutions that are very close in value to the optimal.

3. ***Dynamic programming*** technique is similar to divide and conquer approach. Both solve a problem by breaking it down into a several sub problems that can be solved recursively. The difference between the two is that in dynamic programming approach, the results obtained from solving smaller sub problems are ***reused*** (by maintaining a table of results) in the calculation of larger sub problems. Thus dynamic programming is a ***Bottom-up*** approach that begins by solving the smaller sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained. *Reusing* the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the ***re-computations*** **(**computing results twice or more**)** of the same problem. Thus Dynamic programming approach takes much less time than naïve or straightforward methods, such as divide-and-conquer approach which solves problems in *top-down* method and having lots of re-computations. The dynamic programming approach always gives a guarantee to get a optimal solution.

4. The term "***backtrack"*** was coined by American mathematician D.H. Lehmer in the 1950s. Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and relatively quick test of whether it can possibly be completed to a valid solution. Backtrack algorithms try each possibility until they find the right one. It is a depth-first-search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. If there are no more choice points, the search fails.

5. **Branch-and-Bound** (B&B) is a rather general optimization technique that applies where the greedy method and dynamic programming fail.

B&B design strategy is very similar to backtracking in that a state-space-tree is used to solve a problem. Branch and bound is a systematic method for solving optimization problems. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found. Branch and Bound (B&B) is the most widely used tool for solving large scale NP-hard combinatorial optimization problems.

The following table-1 summarizes these techniques with some common problems that follows these techniques with their running time. Each technique has different running time (…time complexity).

Design strategy	**Problems that follows**
Divide & Conquer	▪ Binary search ▪ Multiplication of two n-bits numbers ▪ Quick Sort ▪ Heap Sort ▪ Merge Sort
Greedy Method	▪ Knapsack (fractional) Problem ▪ Minimum cost Spanning tree ✓ Kruskal's algorithm ✓ Prim's algorithm ▪ Single source shortest path problem ✓ Dijkstra's algorithm
Dynamic Programming	▪ All pair shortest path-Floyed algorithm ▪ Chain matrix multiplication ▪ Longest common subsequence (LCS) ▪ 0/1 Knapsack Problem ▪ Traveling salesmen problem (TSP)
Backtracking	▪ N-queen's problem ▪ Sum-of subset
Branch & Bound	▪ Assignment problem ▪ Traveling salesmen problem (TSP)

Table 1: Important Techniques to solve problems

1.4 PSEUDO-CODE FOR ALGORITHM

Pseudo-code (derived from pseudo-code) is a compact and informal high level description of a computer programming algorithm that uses the structural conventions of some programming language. Unlike actual computer language such as C,C++ or JAVA, Pseudo-code typically omits details that are not essential for understanding the algorithm, such as functions (or subroutines), variable declaration, semicolons, special words and so on. Any version of pseudo-code is acceptable as long as its instructions are unambiguous and is resembles in form. Pseudo-code is independent of

any programming language. Pseudo-code cannot be compiled nor executed, and not following any syntax rules.

Flow charts can be thought of as a graphical alternative to pseudo-code. A *flowchart* is a schematic representation of an algorithm, or the step-by-step solution of a problem, using some geometric figures (called flowchart symbols) connected by flow-lines for the purpose of designing or documenting a program.

The purpose of using pseudo-code is that it may be easier to read than conventional programming languages that enables (or helps) the programmers to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, one can write a pseudo-code for any given problem without even knowing what programming language one will use for the final implementation.

Example: The following pseudo-code "finds the maximum of three numbers".

Input parameters: a, b, c
Output parameter: x

```
FindMax(a, b, c, x)
{
  x = a
  if(b > x)    /  if b is larger than x then update x
   x = b
  if(c > x)    / if b is larger than x then update x
   x = c
}
```

The first line of a function consists of the name of the function followed parentheses, in parentheses we pass the parameters of the function. The parameters may be data, variables, arrays, and so on, that are available to the function. In the above algorithm, The parameters are the three input values, $a, b, and\ c$ and the output parameter, x, that is assigned the maximum of the three input values $a, b, and\ c$.

1.4.1 PSEUDO-CODE CONVENTIONS

The following conventions must be used for pseudo-code.

1. Give a valid name for the pseudo-code procedure. (See sample code for insertion sort at the end).
2. Use the line numbers for each line of code.
3. Use proper **Indentation** for every statement in a block structure.
4. For a flow control statements use **if-else.** Always end an **if** statement with an **end-if.** Both if, else and end-if should be aligned vertically in same line.

```
Ex: If(conditional expression)
            statements          (see the indentation)
    else statements
```

$end-if$

5. Use $:= \ or \ " \leftarrow "$ operator for assignments.

   Ex: $i = j \quad or \quad i \leftarrow j$

   $n = 2 \ to \ length[A] \quad or \quad n \leftarrow 2 \ to \ length[A]$

6. Array elements can be represented by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the **ith** element of the array A.

7. For looping or iteration use **for** or **while** statements. Always end a **for** loop with **end-for** and a **while** with **end-while**.

8. The conditional expression of **for** or **while** can be written as shown in rule (4). You can separate two or more conditions with "**and".**

9. If required, we can also put comments in between the symbol /* and */.

A simple pseudo-code for insertion sort using the above conventions:

```
INSERTION − SORT(A)
1. for j ← 2 to length[A]
2.     key ← A[j]
3.     i ← j − 1     /* insert A[j] into sorted sequence A[1…j-1] */
4.     while i > 0 and A[i] > key
5.       A[i + 1] ← A[i]
6.         i ← i − 1
7.      end − while
8.      A[i + 1] ← key
9. end − for
```

1.5 MATHEMATICAL INDUCTION AND MATEMATICAL FORMULE FOR ALGORITHMS

Mathematical Induction

In this section we will describe what mathematical induction is and also introduce some formulae which are extensively used in mathematical induction.

Mathematical induction is a method of **mathematical proof** typically used to establish that a given statement is true for all natural number (**positive integers**). It is done by proving that the first statement in the infinite sequence of statements is true, and then proving that if any one statement in the infinite sequence of statements is true, then so is the next one.

For Example, let us prove that

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots . + n = \frac{n(n+1)}{2} \quad for\ all\ n \geq 1$$

Here the sequence of statements is:

Statement1: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$

Statement2: $\sum_{i=1}^{2} i = 1 + 2 = \frac{2(2+1)}{2}$

Statement3: $\sum_{i=1}^{3} i = 1 + 2 + 3 = \frac{3(3+1)}{2}$

…….

…….

Statement n: $\sum_{i=1}^{n} i = 1 + 2 + \cdots . + n = \frac{n(n+1)}{2}$

…….

……

Statements 1-3 are obtained by everywhere replacing n by 1 in the original equation, then n by 2, and then n by 3.

Thus a Mathematical induction is a method of **mathematical proof** of a given formula (or statement), by proving a sequence of statements $S(1), S(2), \ldots \ldots$

Proof by using Mathematical Induction of a given statement (or formula), defined on the positive integer N, consists of two steps:

1. (**Base Step**): Prove that S(1) is true
2. **(Inductive Step)**: Assume that S(n) is true, and prove that $S(n+1)$ is true for all $n \geq 1$.

Example1: Prove the proposition that "The sum of the first n positive integers is $\frac{n(n+1)}{2}$; that is $S(n) = 1 + 2 + \cdots \ldots + n = \frac{n(n+1)}{2}$ by induction.

Proof:

(**Base Step**): We must show that the given equation is true for $n = 1$.

i.e. $S(1) = 1 = \frac{1(1+1)}{2} = 1 \Rightarrow$ this is true.

Hence we proved "S(1) is true".

(Induction Step):

Let us assume that the given equation S(n) is true for n;

that is $S(n) = 1 + 2 + \cdots \ldots + n = \frac{n(n+1)}{2}$;

Now we have to prove that it is true for (n+1).

Consider

$$S(n+1) = 1+2+3+.....+n+(n+1)$$
$$= P(n)+(n+1)$$
$$= \frac{n(n+1)}{2}+(n+1)$$
$$= \frac{(n+1)(n+2)}{2} = \frac{(n+1)[(n+1)+1]}{2}$$

Hence S(n+1) is also true whenever S(n) is true. This implies that equation S(n) is true for all $n \geq 1$

Example2: Prove the following proposition using induction:

$$P(n): 1^2+2^2+3^2+.....+n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Proof: (Base Step):

Consider n=1, then

$$P(1) = 1^2 = \frac{1(1+1)(2+1)}{6} = \frac{1.2.3}{6} = 1$$

Hence for n=1 it is true.

(Induction Step):

Assume P(n) is true for 'n' i.e.

$$P(n) = 1^2+2^2+3^2+.....+n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Now

$$P(n+1) = 1^2+2^2+3^2+.....+n^2+(n+1)^2$$
$$= \frac{n(n+1)(2n+1)}{6}+(n+1)^2$$
$$= \frac{n(n+1)(2n+1)+6(n+1)^2}{6}$$
$$= \frac{(n+1)[n(2n+1)+6(n+1)]}{6}$$
$$= \frac{(n+1)[2n^2+n+6n+6]}{6}$$
$$= \frac{(n+1)(n+2)(2n+3)}{6}$$
$$= \frac{(n+1)(n+2)[(n+1)+1]}{6}$$

Which is P(n+1).

Hence P(n+1) is also true whenever P(n) is true $\Rightarrow$ P(n) is true for all n.

In the subsequent section we will look at some formulae which are usually used in mathematical proof.

Sum formula for Arithmetic Series

Given a arithmetic Series: $a, a+d, a+2d, \ldots\ldots\ldots\ldots, a+(n-1).d$

$$\sum_{i=0}^{n-1}(a+id) = \frac{n}{2}[2a+(n-1).d] = \frac{n}{2}[a+l]$$

where a is a first term, d is a common difference *and*

$l = a+(n-1).d$ is a last or n^{th} term

Sum formula for Geometric Series

For $real\ r \neq 1, the\ summation$

$$\sum_{i=0}^{n-1} ax^k = a + ax + ax^2 + ax^3 + \cdots\ldots + ax^{n-1}$$

is a ***geometric*** or ***exponential*** series and has a value

I) $\sum_{k=0}^{n-1} ax^k = \frac{a(x^n-1)}{x-1}$ (when $x > 1$)

II) $\sum_{k=0}^{n-1} ax^k = \frac{a(1-x^n)}{1-x}$ (when $x < 1$)

III) When the summation is $infinite$ and $|x| < 1$, we have

$$\sum_{k=0}^{\infty} ax^k = \frac{a}{1-x}$$

Logarithmic Formulae

The following logarithmic formulae are quite useful for solving recurrence relation.

For all real $a > 0, b > 0, c > 0,$ *and* n;

1. $b^{log_b a} = a$
2. $log_c(ab) = log_c a + log_c b$
3. $log_b a^n = n log_b a$
4. $log_b\left(\frac{m}{n}\right) = log_b m - log_b n$ $where\ a > 0\ and\ a \neq 1$
5. $log_b a = \frac{log_c a}{log_c b}$
6. $log_b a = 1/log_a b$

7. $a^{log_b n} = n^{log_b a}$

(Note: Proof is left as an exercise)

Remark: Since changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor (see property 4), we don't care about the base of a "log" when we are writing a time complexity using O, Ω, or θ- notations. We always write 2 as the base of a log. **For Example**: $log_{3/2} n = \frac{log_2 n}{log_2 \frac{3}{2}} = O(log_2 n)$

1.6 SOME MORE EXAMPLES TO UNDERSTAND TIME & SPACE COMPLEXITY

Given a program and we have to find out its time complexity (i.e. Best case, Worst case and Average case). In order to do that, we will measure the complexity for every step in the program and then calculate overall time complexity.

Space Complexity

The Space Complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion. The time complexity of an algorithm is given by the no. of steps taken be the algorithm to compute the function it was written for.

Time Complexity

The time t_p, taken by a program P, is the sum of the Compile time & the Run (execution) time. The Compile time does not depends on the instance characteristics (i.e. no. of inputs, no. of outputs, magnitude of inputs, magnitude of outputs etc.).

Thus we are concerned with the running time of a program only.

1. Algorithm X (a,b,c)

   { $return\ a + b + b * c + \frac{a+b+c}{a} + b$

   }

   Here the problem instance is characterized by the specified values of a, b, and c.

2. Algorithm SUM (a, n)

```
{ S:= 0
  For i = 1 to n do
      S = S + a [i];
}     Return S;
```

Here the problem instance is characterized by value of n, i.e., number of elements to be summed.

This run time is denoted by t_p, we have:

$t_p(n) = C_a ADD(n) + C_s SUB(n) + C_m MUL(n) + C_d DIV(n) + \ldots.$

where n → instance characteristics.
C_a, C_s, C_m, C_d denotes time needed for an Addition, Subtraction, Multiplication, Division and so on.

ADD, SUB, MUL, DIV is a functions whose values are the numbers of performed when the code for P is used on an instance with characteristics n.
Generally, the time complexity of an algorithm is given by the no. steps taken by the algorithm to complete the function it was written for.
The number of steps is itself a function of the instance characteristics.

How to calculate time complexity of any program

The number of machine instructions which a program executes during its running time is called its *time complexity*. This number depends primarily on the size of the program's input. Time taken by a program is the sum of the compile time and the run time. In time complexity, we consider run time only. The time required by an algorithm is determined by the number of elementary operations.

The following primitive operations that are independent from the programming language are used to calculate the running time:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation
- Comparing two variables
- Indexing into a array of following a pointer reference
- Returning from a function

The following fragment shows how to count the number of primitive operations executed by an algorithm.

```
    int sum(int n)
    {
    int i,sum;
1.  sum = 0;                          //add 1 to the time count
2.  for(i = 0; i < n;i++)             //add n + 1 to the time count
     {
3.     sum = sum + i * i;             //add n to the time count
     }
4.   return sum;                      //add 1 to the time count
```

This function returns the sum from
$i = 1$ *to* n *of* i *squared, i.e.* $sum = 1^2 + 2^2 + \cdots \ldots + n^2$.

To determine the running time of this program, we have to count the number of statements that are executed in this procedure. The code at line 1 executes 1 time, at line 2 the **for loop** executes $(n+1) time,$ Line 3 executes $n\ times,$ and line 4 executes 1 time. Hence $the\ sum\ is = 1 + (n+1) + n + 1 = 2n + 3.$

In terms of O-notation this function is $O(n).$

Example1

	Frequency	Total steps
Add(a, b, c, m, n)	–	
{	–	
For i = 1 to m do	m	m
For j = 1 to n do	m(n)	m.n
c[i, j] = a[i, j] + b[i, j]	m n	m n
}	–	–
		2mn+m

Time Complexity= (2mn + m)=$O(mn)$

Best, Worst and Average case (Step Count)

- **Best Case:** It is the minimum number of steps that can be executed for the given parameter.
- **Worst Case: It** is the maximum no. of steps that can be executed for the given parameter.
- **Average case**: It is the Average no. of steps that can be executed for the given parameter.

To better understand all of the above 3 cases, consider an example of English dictionary, used to search a meaning of a particular word.

Best Case: Suppose we open a dictionary and luckily we get the meaning of a word which we are looking for. This requires only one step (minimum possible) to get the meaning of a word.
Worst case: Suppose you are searching a meaning of a word and that word is either not in a dictionary or that word takes maximum possible steps (i.e. now no left hand side or right hand side page is possible to see).

Average Case: If you are searching a word for which neither a Minimum (best case) nor a maximum (worst case) steps are required is called average case. In this case, we definitely get the meaning of that word.

To understand the concept of Best, Average and worst case and the where to use basic notations O, Ω, and Θ, consider again another example known as **Binary search** algorithm. A Binary search is a well known searching algorithm which follows the same concept of English dictionary.
To understand a Binary search algorithm consider a sorted linear array$A[1 \ldots . n]$, Suppose you want *either* to find(or search) the location LOC of a given element (say x) in the given array A (successful search) or to send some message, such as LOC=0, to indicate that x does not appear in A(Unsuccessful search). A Binary search algorithm is an efficient technique for finding a position of specified value (say x) within a **sorted array** A. The best example of binary search is "dictionary", which we are using in our daily

life to find the meaning of any word (as explained earlier). The Binary search algorithm proceeds as follow:

1) Begin with the interval covering the whole array; binary search repeatedly divides the search interval (i.e. Sorted array A) in half.
2) At each step, the algorithm compares the given input key (or search) value x with the key value of the middle element of the array A.
3) If it match, then a searching element x has been found so its index, or position, is returned. Otherwise, if the value of the search element x is less than the item in the middle of the interval; then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search element x is greater than the middle element's key, then on the sub-array to the right.
4) We repeatedly check until the searched element is found (i.e.$x = A[LOC]$) or the interval is empty, which indicates x is "not found".

Best, Worst and Average case for Binary search algorithm

Best Case: Clearly the best case occurs when you divide the array 1st time and you get the searched element x. This requires only one step (minimum possible step). In this case, the number of comparison, $C(n) = 1 = O(1)$.

Worst case: Suppose you are searching a given key $(i.e.\ x)$ and that x is either not in a given array $A[1, \dots, n]$ or to search that element x takes maximum possible steps (i.e. now no left hand side or right hand side elements in array A is possible to see). Clearly the worst case occurs, when x is searched at last.
Let us assume for the moment that the size of the array is a power of 2, say 2^k. Each time when we examine the middle element, we cut the size of the sub-array is half. So before the 1st iteration size of the array is 2^k.
After the 1st iteration size of the sub-array of our interest is : 2^{k-1}.
After the 2nd iteration size of the sub-array of our interest is : 2^{k-2}
……
…….
After the k^{th}.iteration size of the sub-array of our interest is : $2^{k-k} = 1$
So we stop now for the next iteration. Thus we have at most $(k+1) = (logn + 1)$ iterations.

Since with each iteration, we perform a constant amount of work: Computing a mid point and few comparisons. So overall, for an array of size n, we perform $C.(logn + 1) = O(logn)$ comparisions. Thus $T(n) = O(logn)$

Average Case: If you are searching given key x for which neither a Minimum (best case) nor a maximum (worst case) steps are required is called average case. In this case, we definitely get the given key x in the array A. In this case also $C(n) = O(logn)$

The following table summarizes the time complexity of Binary search in various cases:

Cases	Suppose Element (say x) present in Array A	Element x, not present in A
Best:	1 step required $\Rightarrow O(1)$	$O(logn)$
Worst:	$logn\ steps\ required\ \Rightarrow O(logn)$	$O(logn)$
Average	$logn\ steps\ required\ \Rightarrow O(logn)$	$O(logn)$

1.7 ASYMPTOTIC NOTATIONS (O, Ω, and θ)

Asymptotic notations have been used in earlier sections in brief. In this section we will elaborate these notations in detail. They will be further taken up in the next unit of this block.

These notations are used to describe the Running time of an algorithm, in terms of functions, whose domains are the set of natural numbers, N = {1, 2, ……}. Such notations are convenient for describing the worst case running time function. T(n) (problem size input size).

The complexity function can be also be used to compare two algorithms P and Q that perform the same task.

The basic Asymptotic Notations are:

1. O(Big-"Oh") Notation. [Maximum number of steps to solve a problem, (upper bound)]
2. Ω (Big-"Omega") Notation [Minimum number of steps to solve a problem, (lower bound)]
3. Θ(Theta) Notation [Average number of steps to solve a problem, (used to express both upper and lower bound of a given $f(n)$)

1.7.1 THE NOTATION O (BIG 'Oh')

Big 'Oh' Notation is used to express an asymptotic upper bound (maximum steps) for a given function $f(n)$. Let $f(n)$ and $g(n)$ are two positive functions , each from the set of natural numbers (domain) to the positive real numbers.

We say that the function $f(n) = O(g(n))$ [read as "f of n is big "Oh" of g of n"], if there exist two positive constants C and n_0 such that

$f(n) \leq C.g(n)$: $\forall$ $n \geq n_0$

The intuition behind O- notation is shown in Figure 1.

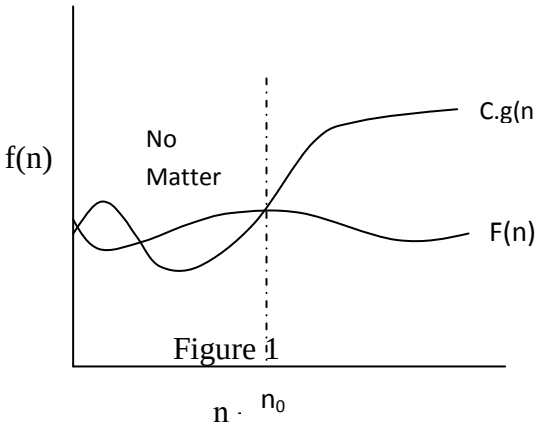


Figure 1

Figure 1

For all values of n to the right of n_0, the value of f(n) is always lies on or below $Cg(n)$.

To understand O-Notation let us consider the following examples:

Example1.1: For the function defined by $(n) = 2n^2 + 3n + 1$: show that

$(i) f(n) = O(n^2)$

$(ii)\ f(n) = O(n^3)$

$(iii) n^2 = O(f(n))$

$(iv)\ f(n) \neq O(n)$

$(v)\ n^3 \neq O(f(n))$

Solution:

(i) To show that $(n) = O(n^2)$; we have to show that

$f(n) \leq C.g(n)\ \forall n \geq n_0$

$\Rightarrow\ 2n^2 + 3n + 1 \leq C.n^2\ \ldots..(1)$

$clearly\ this\ equation\ (1)\ is\ satisfied\ for\ C = 6\ and\ for\ all\ n \geq 1$

$2n^2 + 3n + 1 \leq 6.n^2\ \ for\ all\ n \geq 1$; since we have found the required constant C and $n_0 = 1$. Hence $f(n) = O(n^2)$.

Remark: The value of C and n_0 is not unique. For example, to satisfy the above equation (1), we can also take $= 3\ and\ n \geq 3$. So depending on the value of C, the value of n_0 is also changes. Thus any value of C and n_0 which satisfy the given inequality is a valid solution.

(ii) To show that $(n) = O(n^3)$; we have to show that

$f(n) \leq C.g(n)\ \forall n \geq n_0$

$\Rightarrow\ 2n^2 + 3n + 1 \leq C.n^3\ \ldots..(1)$; **Let C=3**

Value of n	$2n^2+3n+1$	$3n^3$
$n=1$	6	3
$n=2$	15	24
$n=3$	27	81
…..	…..	…..

$clearly\ this\ equation\ (1)\ is\ satisfied\ for\ C=3\ and\ for\ all\ n \geq 2$

$2n^2+3n+1 \leq 3.n^2\ \ for\ all\ n \geq 2$; Since we have found the required constant C and $n_0 = 2$. Hence $f(n) = O(n^3)$.

(iii) To show $n^2 = O(f(n))$ we have to show that $n^2 \leq C(2n^2+3n+1)$

For $C = 1\ and\ n \geq 1$; we get $n^2 \leq (2n^2+3n+1)$.

(iv) To show $(n) \neq O(n)$, you have to show that

$$f(n) \leq C.n \Longrightarrow 2n^2+3n+1 \leq C.n$$

$$or\ \ (2n+3+\tfrac{1}{n}) \leq C \ \ldots\ldots. (1)$$

There is no value of C and n_0, which satisfy this equation (1). For example, if you take $C = 10^6$, then to contradict this inequality you can take any value greater than C, that is $n_0 = 10^7$. Since we do not found the required constant C and n_0 to satisfy (1). Hence $f(n) \neq O(n)$

(v) Do yourself.

Theorem: $If\ f(n) = a_m n^m + a_{m-1}n^{m-1} + \cdots \ldots \ldots + a_1 n^1 + a_0$;

$$then\ f(n) = O(n^m)$$

1.7.2 THE NOTATION Ω (BIG 'Omega')

O- Notation provides an asymptotic upper bound for a function; Ω-Notation, provides an asymptotic *lower-bound* for a given function.

We say that the function $f(n) = \Omega(g(n))$ [read as "f of n is big "Omega" of g of n"], if and only if there exist two positive constants C and n_0 such that

$f(n) \geq C.g(n)$: $\forall$ n ≥ n_0

The intuition behind Ω- notation is shown in Figure 2.

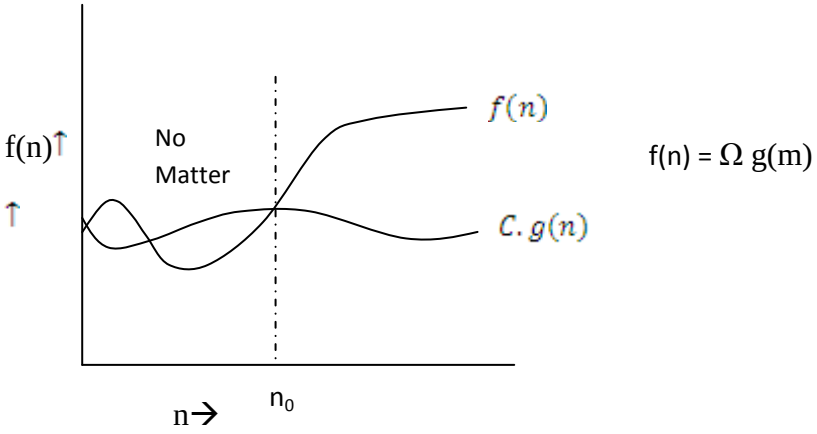


Figure 2

Note that for all values of f(n) always lies on or above g(n).

To understand Ω-Notation let us consider the following examples:

Example1.1: For the function defined by

$f(n) = 2n^3 + 3n^2 + 1$ *and*

$g(n) = 2n^2 + 3$: show that

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

$(ii)\ g(n) \neq \Omega(f(n))$

$(iii)\ n^3 = \Omega\big(g(n)\big)$

$(iv)\ f(n) \neq \Omega(n^4)$

$(v)\ n^2 \neq \Omega(f(n))$

Solution:

(i) To show that $(n) = \Omega(g(n))$; we have to show that

$f(n) \geq C.g(n)\ \forall n \geq n_0$

$\Longrightarrow\ 2n^3 + 3n^2 + 1\ \geq C(2n^2 + 3)$ ..... (1)

clearly this equation (1) *is satisfied for* $C = 1$ *and for all* $n \geq 1$

Since we have found the required constant C and $n_0 = 1$.

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

(ii) To show that $g(n) \neq \Omega(f(n))$; we have to show that no value of C and n_0 is there which satisfy the following equation (1). We can prove this result by contradiction.

Let $g(n) = \Omega(f(n)) \Rightarrow 2n^2 + 3 = \Omega(2n^3 + 3n^2 + 1)$ then there exist a positive constant C and n_0 such that

$2n^2 + 3 \geq C(2n^3 + 3n^2 + 1) \quad \forall n \geq n_0 \ldots\ldots\ldots (1)$

$(2 - 3C)n^2 \geq 2Cn^3 - 2 \geq Cn^3$

$\Rightarrow \frac{(2-3C)}{C} \geq n \quad for\ all\ n \geq n_0$

But for any $> \frac{(2-3C)}{C}$; the inequality (1) is not satisfied ⇒ Contradiction.

Thus $g(n) \neq \Omega(f(n))$.

(iii) To show that $n^3 = \Omega(g(n))$; we have to show that

$n^3 \geq C.g(n) \ \forall n \geq n_0$

$\Longrightarrow n^3 \geq C(2n^2 + 3) \ldots..(1)$

$clearly\ this\ equation\ (1)\ is\ satisfied\ for\ C = \frac{1}{2}\ and\ for\ all\ n \geq 2\ (i.e.\ n_0 = 2)$

Thus $n^3 = \Omega(g(n))$

(iv) We can prove the result by contradiction.

Let $f(n) = \Omega(n^4) \Rightarrow 2n^3 + 3n^2 + 1 \geq C.n^4 \quad \ldots..(1)$

$\Rightarrow 6n^3 \geq C.n^4 \ for\ all\ n \geq n_0 \geq 1$

$\Rightarrow 6 \geq C.n \Rightarrow \frac{6}{C} \geq n \ \ for\ all\ n \geq n_0 \quad \ldots\ldots.(2)$

$But\ for\ x = (\frac{6}{C} + 1)$ the inequality (1) or (2) is not satisfied

⇒ Contradiction, hence proved.

(v) Do yourself.

1.7.3 THE NOTATION θ (Theta)

Θ-Notation provides simultaneous both asymptotic lower bound and asymptotic upper bound for a given function.

Let $f(n)$ and $g(n)$ are two positive functions , each from the set of natural numbers (domain) to the positive real numbers. In some cases, we have

$f(n) = O(g(n))\ and\ f = \Omega(g(n))\ then\ f(n) = \Theta(g(n))$.

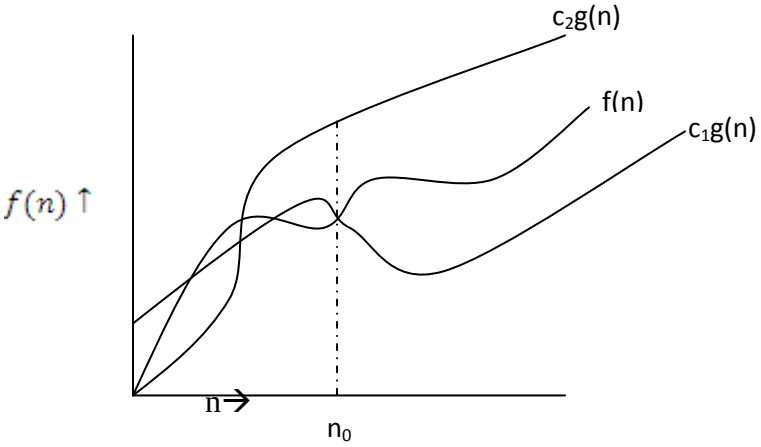
We say that the function $f(n) = \Theta(g(n))$ [read as "f of n is Theta" of g of n"], if and only if there exist three positive constants $C_1,\ C_2\ and\ n_0$ such that

$C_1.g(n) \leq f(n) \leq C_2.g(n) \quad for\ all\ n \geq n_0 \qquad \ldots\ldots\ldots..(1)$

(Note that this inequality (1) represents two conditions to be satisfied simultaneously viz $C_1.g(n) \leq f(n)$ and $f(n) \leq C_2.g(n)$;

clearly this implies
if $f(n) = O(g(n))$ *and* $f = \Omega(g(n))$ *then* $f(n) = \Theta(g(n))$.

The following figure -1 shows the intuition behind the Θ-Notation.



f(n) = θ(g(n))

Figure 3

Note that for all values of n to the right of the n_0 the value of f(n) lies at or above $C_1g(n)$ and at or below $C_2.g(n)$.

To understand Ω-Notation let us consider the following examples:

Example1.1: For the function defined by

$f(n) = 2n^3 + 3n^2 + 1$ *and*

$g(n) = 2n^3 + 1$: show that

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

$(ii)\ f(n) \neq \Theta(n^2)$

$(iii)\ n^4 \neq \Theta(g(n))$

Solution: To show that $(n) = \Theta(g(n))$; we have to show that

$C_1.g(n) \leq f(n) \leq C_2.g(n)$ *for all* $n \geq n_0$

$\Rightarrow C_1(2n^3 + 1) \leq 2n^3 + 3n^2 + 1 \leq C_2(2n^3 + 1)$ *for all* $n \geq n_0$ ….(1)

To satisfy this inequality (1) simultaneously, we have to find the value of C_1, C_2 and n_0 , using the following inequality

$C_1(2n^3 + 1) \leq 2n^3 + 3n^2 + 1$ ……..(2) *and*

$2n^3 + 3n^2 + 1 \leq C_2(2n^3 + 1)$ ……..(3)

Inequality (2) is satisfied for $C_1 = 1$ *and* $n \geq 1$ $(i.e.\ n_0 = 1)$

Inequality (3) is satisfied for $C_2 = 2$ *and* $n \geq 1$ $(i.e.\ n_0 = 1)$

Hence inequality (1) simultaneously satisfied for $C_1 = 1, C_2 = 2 \ and \ n \geq 1$.

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

(ii) We can prove this by contradiction that no value of $C_1,\ C_2 \ or \ n_0 \ exist.$

Let
$f(n) = \Theta(n^2) \Rightarrow C_1.n^2 \leq 2n^3 + 3n^2 + 1 \leq C_2.n^2 \ for \ all \ n \geq n_0 \ \ldots(1)$

Left side inequality: $C_1.n^2 \leq 2n^3 + 3n^2 + 1$; is satisfied for
$C_1 = 1 \ and \ n \geq 1.$

Right side inequality: $2n^3 + 3n^2 + 1 \leq C_2.n^2$

$\Rightarrow n^3 \leq C_2.n^2 \Rightarrow n \leq C_2 \ for \ all \ n \geq n_0;$

But for $n = (C_2 + 1)$, this inequality is not satisfied.

Thus $f(n) \neq \Theta(n^2)$.

(iii) Do yourself.

1.8 SOME USEFUL THEOREMS FOR $O, \Omega \ and \ \Theta$.

The following theorems are quite useful when you are dealing

(or solving problems) with $O, \Omega \ and \ \Theta$.

Theorem1: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots\ldots\ldots + a_1 n + a_0$

$where \ a_m \neq 0 \ and \ \ a_i \in R, \ \ then \ f(n) = O(n^m)$

Proof: $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots\ldots\ldots + a_1 n + a_0$

$= \sum_{i=0}^{m} a_k n^k$

$f(n) \leq \sum_{i=0}^{m} |a_k| n^k$

$\leq n^m \sum_{i=0}^{m} |a_k| n^{k-m} \leq n^m \sum_{i=0}^{m} |a_k|$ for $n \geq 1$

Let us assume $|a_m| + |a_{m-1}| + \cdots \ldots + |a_1| + |a_0| = c$

Then $f(n) \leq cn^m \Longrightarrow f(n) = O(n^m)$.

Theorem2: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots\ldots\ldots + a_1 n + a_0$

$where \ a_m \neq 0 \ and \ \ a_i \in R, \ \ then \ f(n) = \Omega(n^m)$.

Proof: $f(n) = a_m n^m + \ldots\ldots\ldots + a_1 n + a_0$

Since $f(n) \geq cn^m \ for \ all \ n \geq 1 \Longrightarrow f(n) = \Omega(n^m)$

Theorem3: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots\ldots\ldots + a_1 n + a_0$

$where\ a_m \neq 0\ and\ \ a_i \in R,\ \ then\ f(n) = O(n^m)$

Proof: From Theorem1 and Theorem2,

$$f(n) = O(n^m) \quad \ldots\ldots(1)$$

$$f(n) = \Omega(n^m) \quad \ldots\ldots\ldots(2)$$

From (1) and (2) we can say that $f(n) = \Theta(n^m)$.

Example1: By applying theorem, find out the O-notation, Ω- notation and Θ-notation for the following functions.

(i) $f(n) = 5n^3 + 6n^2 + 1$

(ii) $f(n) = 7n^2 + 2n + 3$

Solution:

(i) Here $The\ degree\ of\ a\ polynomial\ f(n)\ is,\ m = 3$, So by Theorem1,2 and 3:

$f(n) = O(n^3),\ f(n) = \Omega(n^3)$ and $f(n) = \Theta(n^3)$.

(ii) $The\ degree\ of\ a\ polynomial\ f(n)\ is,\ m = 2$, So by Theorem1,2 and 3:

$f(n) = O(n^2),\ f(n) = \Omega(n^2)$ and $f(n) = \Theta(n^2)$.

☞ Check Your Progress 1

Q.1 $\sum_{i=1}^{n} O(n)$, Where O(n) stands for order n is:

$a)\ O(n)$ $b)\ O(n^2)$ c) $O(n^3)$ d) none of these

Q.2: What is the running time to retrieve an element from an array of size *n* (in worst case):

$a)\ O(n)$ $b)\ O(n^2)$ c) $O(n^3)$ d) none of these

Q.3: The time complexity for the following function is:

```
for (i = 1; i <= n; i *= 2)
  {  sum = 0;
  }
```

$a)\ O(n)$ $b)\ O(n^2)$ c) $O(nlogn)$ d) $O(logn)$

Q.4: The time complexity for the following function is:

```
for (i = 1; i <= n; i++)
  for(j = 1; j <= n; j = j * 2)
```

{ ……

……

}

a) $O(n)$ b) $O(n^2)$ c) $O(nlogn)$ d) $O((logn)^2)$

Q.5: Define an algorithm? What are the various properties of an algorithm?

Q.6: What are the various fundamental techniques used to design an algorithm efficiently? Also write two problems for each that follows these techniques?

Q.7: Differentiate between profiling and debugging?

Q.8: Differentiate between asymptotic notations O, Ω and Θ?

Q.9: Define time complexity. Explain how time complexity of an algorithm is computed?

Q.10: Let f(n) and g(n) be two asymptotically positive functions. Prove or disprove the following (using the basic definition of O, Ω and Θ):

a) $4n^2 + 7n + 12 = O(n^2)$

b) $logn + \log(logn) = O(logn)$

c) $3n^2 + 7n - 5 = \Theta(n^2)$

d) $2^{n+1} = O(2^n)$

e) $2^{2n} = O(2^n)$

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

g) $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$

h) $f(n) = O(g(n))$ *implies* $2^{f(n)} = O(2^{g(n)})$

i) $f(n) + g(n) = \Theta(\min(f(n), g(n))$

j) $33n^3 + 4n^2 = \Omega(n^4)$

k) $f(n) + g(n) = O(n^2)$ where

$f(n) = 2n^2 - 3n + 5$ *and* $g(n) = nlogn + 10$

1.9 RECURRENCE

There are two important ways to categorize (or major) the effectiveness and efficiency of algorithm: ***Time complexity*** and ***space complexity***. The *time complexity* measures the amount of time used by the algorithm. The *space complexity* measures the amount of space used. We are generally interested to find the best case, average case and worst case complexities of a given algorithm. When a problem becomes "large", we are interested to find *asymptotic complexity* and O (Big-Oh) notation is used to quantify this.

Recurrence relations often arise in calculating the time and space complexity of algorithms. Any problem can be solved either by writing ***recursive*** algorithm or by writing ***non-recursive*** algorithm. A *recursive algorithm* is one which makes a recursive call to itself with smaller inputs. We often use a *recurrence relation* to describe the running time of a recursive algorithm.

A **recurrence relation** is an equation or inequality that describes a function in terms of its value on smaller inputs or as a function of preceding (or lower) terms.

Like all recursive functions, a recurrence also consists of two steps:

1. **Basic step**: Here we have one or more constant values which are used to terminate recurrence. It is also known as **initial conditions** or **base conditions.**
2. **Recursive steps:** This step is used to find new terms from the existing (preceding) terms. Thus in this step the recurrence compute next sequence from the k preceding values $f_{n-1}, f_{n-2}, \cdots, f_{n-k}$. This formula is called a ***recurrence relation*** (or ***recursive formula***). This formula refers to itself, and the argument of the formula must be on smaller values (close to the base value).

Hence a recurrence has one or more initial conditions and a recursive formula, known as ***recurrence relation.***
For example: A Fibonacci sequence $f_0, f_1, f_2, \cdots$ can be defined by the recurrence relation

$$f_n = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & if\ n \geq 2 \end{cases}$$

1. (**Basic Step**) The given recurrence says that if n=0 then $f_0 = 0$ and if n=1 then $f_1 = 1$. These two conditions (or values) where recursion does not call itself is called a **initial conditions** (or **Base conditions**).
2. (**Recursive step**): This step is used to find new terms $f_2, f_3, \cdots,$ from the existing (preceding) terms, by using the formula

$f_n = f_{n-1} + f_{n-2}$; for $n \geq 2$,

This formula says that "by adding two previous sequence (or term) we can get the next term".
For example $f_2 = f_1 + f_0 = 1 + 0 = 1$;
$f_3 = f_2 + f_1 = 1 + 1 = 2$; $f_4 = f_3 + f_2 = 2 + 1 = 3$ **and so on**

Let us consider some recursive algorithm and try to write their recurrence relation. Then later we will learn some method to solve these recurrence relations to analyze the running time of these algorithms.

Example 1: Consider a Factorial function, defined as:

Factorial function is defined as:	/* Algorithm for computing n!
$n! = \begin{cases} 1 & if\ n = 1 \\ n.(n-1)! & if\ n > 1 \end{cases}$	**Input:** $n \in N$ **Output**: $n!$ **Algorithm: FACT(n)** 1: if $n = 1$ *then* 2: *return* 3: else 4: *return* $n * FACT(n-1)$ 5: *endif*

Let $M(n)$ denoted the number of multiplication required to execute the n!, that is $M(n)$ denotes the # of times the line 4 is executed in FACT algorithm.

- We have the initial condition $M(1) = 0$; since when n=1, FACT simply return (i.e. Number of multiplication is 0).
- When $n > 1$, the line 4 is perform 1 multiplication plus FACT is recursively called with input (n-1). It means, by the definition of $M(n)$, additional $M(n-1)$ number of multiplications are required.

Thus we can write a recurrence relation for the algorithm FACT as:

$M(1) = 0$ ***(base case)***
$M(n) = 1 + M(n-1)$ ***(Recursive step)***

(We can also write some constant valve instead of writing 0 and 1, that is

$$M(n) = \begin{cases} b & if\ n = 1 \quad (\textbf{\textit{base case}}) \\ c + M(n-1) & (\textbf{\textit{Recursive step}}) \end{cases}$$

Since when n=1 (base case), the FACT at line 1 perform one comparison and one return statement at line 2. Therefore

$M(1) = O(1) = b,$ Where b is a some constant, and for line 4 (when $n > 1)$ it is $O(1) + M(n-1) = c + M(n-1).$ The reason for writing constants b and c , instead of writing exact value (here 0 and 1) is that, we always interested to find "asymptotic complexity" (i.e. problem size n is "large") and O(big "Oh) notation is used (for getting "Worst case" complexity) to quantify this.
In Section 1.81-1.83 of this unit, we will learn some methods to solve these recurrence relations) .

Example2: Let $T(n)$ denotes the number of times the statement $x = x + 1$ is executed in the algorithm2.

Algorithm2: Example(n)

```
1: if n = 1 then
2:    return
3:  for i = 1 to n
4:      x = x + 1
5:    Example(n/2)
5: endif
```

- The base case is reached when $n = 1.$ The algorithm2 perform one comparison and one return statement. Therefore, $T(1) = O(1) = a$
- When $n > 1,$ the statement $x = x + 1$ executed n times at line 4. Then at line 5, $example( )$ is called with the input $\lfloor \frac{n}{2} \rfloor$, which causes $x = x + 1$ *to be* executed $T\left(\lfloor \frac{n}{2} \rfloor\right)$ additional times. Thus we obtain the recurrence relation as:

$$\begin{cases} T(1) = a & \textbf{(base case)} \\ T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + n & \textbf{(Recursive step)} \end{cases}$$

Example3: Let $T(n)$ denotes the time the statement $x = x + 1$ is executed in the algorithm2.

Algorithm3: $f(n)$

```
1: if (n == 1)
2:    return 2
3:  else
4:      return 3 * f(n/2) + f(n/2) + 5;
5: endif
```

- The base case is reached when $n == 1.$ The algorithm2 performs one comparison and one return statement. Therefore, $T(1) = O(1) = a$, where a is some constant.
- When $n > 1,$ the **algorithm3** performs **TWO** recursive calls each with the parameter $\left(\frac{n}{2}\right)$ at line 4, and some constant number of basic operations. Thus we obtain the recurrence relation as:

$$T(n) = \begin{cases} T(1) = a & (base\ case) \\ 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + b & (Recursive\ step) \end{cases}$$

Example4: The following algorithm4 calculate the value of $x^n\ (i.e. exponentiation).$ Let $T(n)$ denotes the time complexity of the algorithm4.

Algorithm4: $Power(x, n)$

```
1: if (n == 0)
2:   return 1
3: if (n == 1)
4:     return x
5:  else
6:     return x * Power(x, n − 1);
7: endif
```

- The base case is reached when $n == 0\ or\ n == 1.$ The algorithm4 performs one comparison and one return statement. Therefore, $T(0)\ and\ T(1) = O(1) = a$, where a is some constant.
- When $n > 1,$ the **algorithm4** performs one recursive call with input parameter $(n - 1)$ at line 6, and some constant number of basic operations. Thus we obtain the recurrence relation as:

$$T(n) = \begin{cases} T(1) = a & (base\ case) \\ T(n-1) + b & (Recursive\ step) \end{cases}$$

Example5: The worst case running time T(n) of Binary Search procedure (explained earlier) is given by the recurrence:

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ T\left(\frac{n}{2}\right) + O(1) & if\ n > 1 \end{cases}$$

METHODS FOR SOLVING RECURRENCE RELTIONS

We will introduce three methods of solving the recurrence equation:

1. The Substitution Method (*Guess the solution & verify by Induction*)

2. Iteration Method (*unrolling and summing*)
3. The Recursion-tree method
4. Master method

In substitution method, we guess a bound and then use mathematical induction to prove our guess correct. The iteration method converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence and the Master method provides bounds for the recurrence of the form

1.9.1 SUBSTITUTION METHOD

A substitution method is one, in which we guess a bound and then use mathematical induction to prove our guess correct. It is basically two step process:

Step1: Guess the form of the Solution.

Step2: Prove your guess is correct by using Mathematical Induction.

Example 1. Solve the following recurrence by using substitution method.

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

Solution*: step1:* The given recurrence is quite similar with that of MERGE-SORT, you guess the solution is $T(n) = O(nlogn)$

Or $T(n) \leq c.nlogn$

Step2: Now we use mathematical Induction.

Here our guess does not hold for n=1 because $T(1) \leq c.1log1$

$i.e.T(n) \leq 0$ *which is contradiction with* $T(1) = 1$

Now for n=2

$T(2) \leq c.2log2$

$2T\left(\frac{2}{2}\right) + 2 \leq c.2$

$2T(1) + 2 \leq c.2$

$0 + 2 \leq c.2$

$2 \leq c.2$ which is true. So $T(2) \leq c.nlogn$ is True for n=2

(ii) Induction step: Now assume it is true for n=n/2

$i.e.T\left(\frac{n}{2}\right) \leq c.\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)$ *is true.*

Now we have to show that it is true for $n = n$

$i.e. T(n) \leq c.nlogn$

We know that $T(n) \leq 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

$\leq 2\left(c\left\lfloor\frac{n}{2}\right\rfloor log\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

$\leq cnlog\left\lfloor\frac{n}{2}\right\rfloor + n \;\; \leq cnlogn - cnlog2 + n$

$\leq cnlogn - cn + n$

$\leq cnlogn \;\; \forall\, c \geq 1$

Thus $T(n) = O(nlogn)$

Remark: Making a good guess, which can be a solution of a given recurrence, requires experiences. So, in general, we are often not using this method to get a solution of the given recurrence.

1.9.2 ITERATION METHOD (*Unrolling and summing*):

In this method we unroll (or substituting) the given recurrence back to itself until not getting a regular pattern (or series).

We generally follow the following steps to solve any recurrence:

- Expend the recurrence
- Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern.
- Evaluate the summation by using the arithmetic or geometric summation formulae as given in section1.4 of this unit.

Example1: Consider a recurrence relation of algorithm1 of section 1.8

$$M(n) = \begin{cases} b & if\ n = 1 \qquad (\textbf{\textit{base case}}) \\ c + M(n-1) & n \geq 2 \qquad (\textbf{\textit{Recursive step}}) \end{cases}$$

Solution: Here

$M(1) = b \qquad \ldots\ldots\ldots(1)$

$M(n) = c + M(n-1) \qquad \ldots\ldots\ldots(2)$

Now substitute the value of M(n-1) in equation(2),

$M(n) = c + M(n-1)$

$= c + \{c + M(n-2)\}$

$= c + c + M(n-2)$

$= c + c + c + M(n-3)$ [By substituting M(n-2) in equation (2)

$= \cdots\cdots\cdots$

$= c + c + c + \cdots..(n-1)times + M(n-(n-1))$

$= (n-1)c + M(1) = nc - c + b = nc + (b-c) = O(n)$

Example2: Consider a recurrence relation of algorithm2 of section 1.8

$$\begin{pmatrix} T(1) = a & (\boldsymbol{base\ case}) \\ T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n & (\boldsymbol{Recursive\ step}) \end{pmatrix}$$

Solution:

Solution: Here

$T(1) = b \qquad \ldots\ldots\ldots(1)$

$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n \qquad \ldots\ldots\ldots(2)$

When we are solving recurrences we always omit the sealing or floor because it won't affect the result. Hence we can write the equation 2 as:

$T(n) = T\left(\frac{n}{2}\right) + n \qquad \ldots\ldots\ldots(3)$

Now substitute the value of $T\left(\frac{n}{2}\right)$ in equation (3),

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

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

$= n + \frac{n}{2} + \frac{n}{4} + T\left(\frac{n}{8}\right)$ (By substituting $T\left(\frac{n}{4}\right)$ *in equation* 3)

$= \underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \ldots \frac{n}{2^{k-1}}}_{k=log_2 n\ terms\ (total)} + T\left(\frac{n}{2^k}\right)$

for getting boundary condition;

$T\left(\frac{n}{2^k}\right) = T(1) \Longrightarrow \frac{n}{2^k} = 1 \Longrightarrow k = log_2 n.$ [Taking log both side]

Thus we have a G.P. series:

$\underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \ldots \frac{n}{2^{k-1}}}_{k=log_2 n\ terms\ (total)} + T(1) = \underbrace{n + \frac{n}{2} + \frac{n}{4} + \cdots \ldots \frac{n}{2^{k-1}}}_{k=log_2 n\ terms} + b$

$= \frac{n\left[1-\left(\frac{1}{2}\right)^{log_2 n}\right]}{\left(1-\frac{1}{2}\right)}$ [By using GP series sum formula $S_n = \frac{a(1-x^n)}{1-x}$]

$= \frac{n[1-n^{log_2 \frac{1}{2}}]}{(1-\frac{1}{2})}$ [Using log property $a^{log_b n} = n^{log_b a}$]

$= 2n[1 - n^{log_2 1 - log_2 2}] = 2n[1 - n^{0-1}] = 2n\left[1 - \frac{1}{n}\right]$

$= 2n - 2 = O(n)$

1.9.3 RECURSION TREE METHOD

A recursion tree is a convenient way to visualize what happens when a recurrence is iterated. It is a pictorial representation of a given recurrence relation, which shows how Recurrence is divided till Boundary conditions.

Recursion tree method is especially used to solve a recurrence of the form:

$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ \ldots\ldots\ldots (1) \quad where\ a > 1, b \geq 1$

This recurrence (1) describe the running time of any divide-and-conquer algorithm.

Method (steps) for solving a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ using recursionTree:

Step1: We make a recursion tree for a given recurrence as follow:

a) To make a recursion tree of a given recurrence (1), First put the value of $f(n)$ at root node of a tree and make a $\boldsymbol{a\ number}$ of child node of this root value $f(n)$. Now tree will be looks like as:

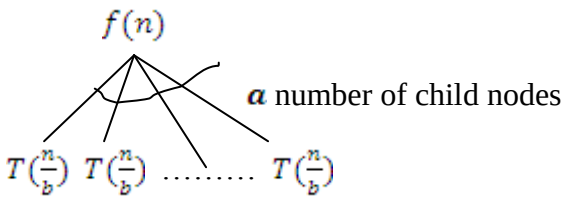


Figure-a

b) Now we have to find the value of $T(\frac{n}{b})$ by putting (n/b) in place of n in equation (1).That is

$T\left(\frac{n}{b}\right) = aT\left(\frac{\frac{n}{b}}{b}\right) + f(n/b) = aT\left(\frac{n}{b^2}\right) + f(n/b) \ldots$ (2)

From equation (2), now $f(n/b)$ will be the value of node having a branch (child nodes) each of size T(n/b). Now each $T\left(\frac{n}{b}\right)$ in **figure-a** will be replaced as follows:

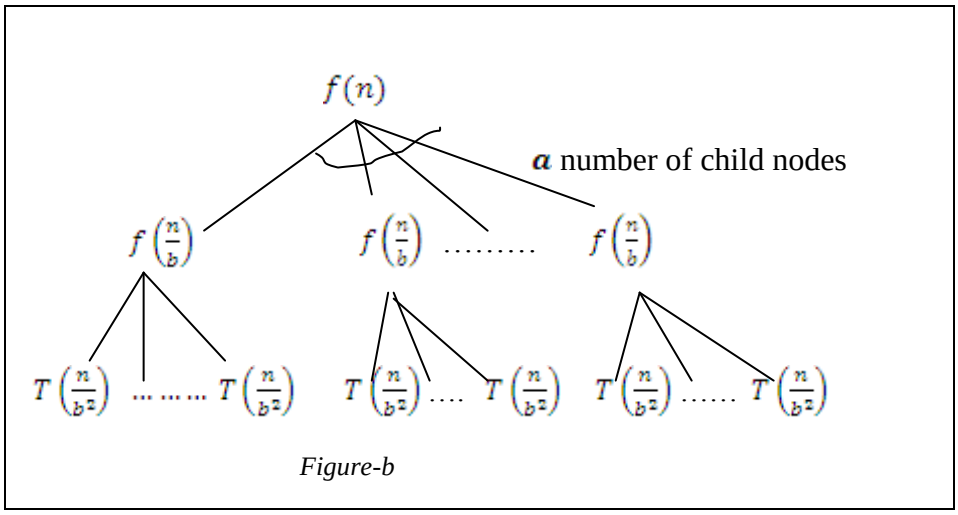


Figure-b

c) In this way you can expend a tree one more level (i.e. up to (at least) 2 levels).

Step2: (a) Now you have to find per level cost of a tree. Per level cost is the sum of the cost of each node at that level. For example per level cost at level1 is $\left(\frac{n}{b}\right) + f\left(\frac{n}{b}\right) + \cdots . f\left(\frac{n}{b}\right)\ (a\ times)$. This is also called **Row-Sum.**

(b) Now the total (final) cost of the tree can be obtained by taking the sum of costs of all these levels.
$i.e.\ Total\ cost = sum\ of\ costs\ of\ (l_0 + l_1 + \cdots \ldots \ldots . + l_k)$. This is also called **Column-Sum.**

Let us take one example to understand the concept to solve a recurrence using recursion tree method:

Example1: Solve the recurrence $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n}$ using recursion tree method.

Solution: **Step1:** First you make a recursion tree of a given recurrence.

1. To make a recursion tree, you have to write the value of $f(n)$ at root node. And

2. The number of child of a Root Node is equal to the value of a. (Here the value of $a = 2$). So recursion tree be looks like as:

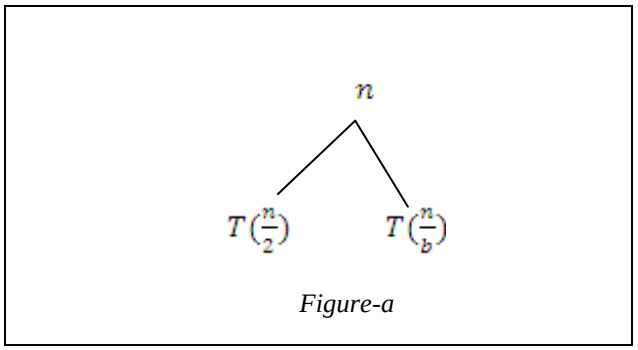


Figure-a

b) Now we have to find the value of $T(\frac{n}{2})$ in figure (a) by putting (n/2) in place of n in equation (1).That is

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{\frac{n}{2}}{2}\right) + n = 2T\left(\frac{n}{2^2}\right) + n/2 \ldots (2)$$

From equation (2), now $(\frac{n}{2})$ will be the value of node having 2 branch (child nodes) each of size T(n/2). Now each $T\left(\frac{n}{2}\right)$ in **figure-a** will be replaced as follows:

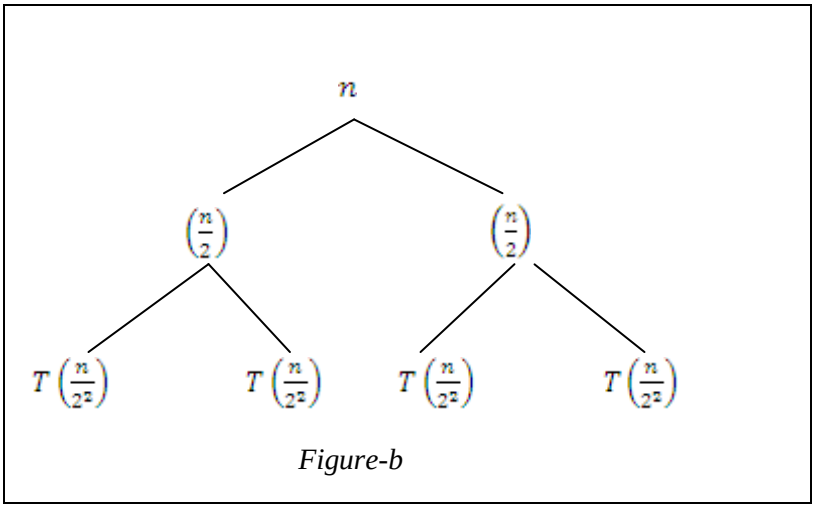


Figure-b

c) In this way, you can extend a tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

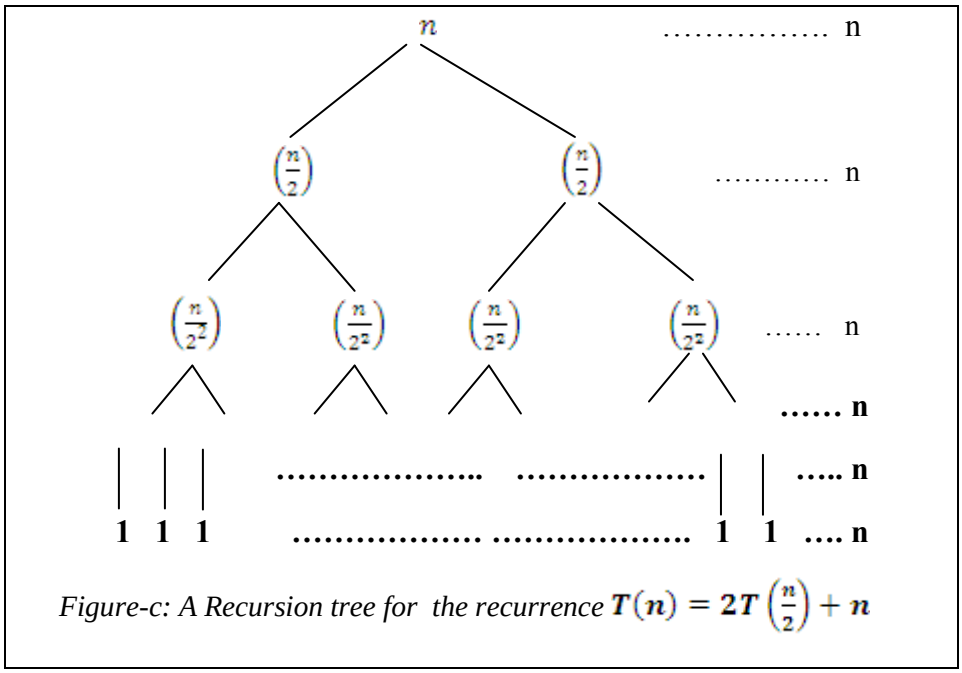


Figure-c: A Recursion tree for the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$

Now we find the per level cost of a tree, Per-level cost is the sum of the costs within each level (called row sum). Here per level cost is n. For example: per level cost at depth 2 in **figure-c** can be obtained as:

$$\left(\frac{n}{2^2}\right) + \left(\frac{n}{2^2}\right) + \left(\frac{n}{2^2}\right) + \left(\frac{n}{2^2}\right) = n.$$

Then total cost is the sum of the costs of all levels (called column sum), which gives the solution of a given Recurrence. The height of the tree is

Total cost= $n + n + n + \cdots \ldots \ldots + n$ ------- (3)

To find the sum of this series you have to find the total number of terms in this series. To find a total number of terms, you have to find a height of a tree.

Height of tree can be obtained as follow (see recursion tree of figure c): you start a problem of size n, then problem size reduces to $\left(\frac{n}{2}\right)$, then $\left(\frac{n}{2^2}\right)$ and so on till boundary condition (problem size 1) is not reached. That is

$$n \to \left(\frac{n}{2}\right) \to \left(\frac{n}{2^2}\right) \to \cdots \ldots \ldots \ldots \to \left(\frac{n}{2^k}\right)$$

At last level problem size will be equal to 1 if
$\left(\frac{n}{2^k}\right) = 1 \Rightarrow n = 2^k \Rightarrow k = log_2 n.$

This k represent the height of the tree, hence height $= k = log_2 n.$

Hence total cost in equation (3) is

$n + n + n + \cdots \ldots \ldots + n\ (log_2 n\ terms) = nlog_2 n \Rightarrow O(nlog_2 n)$.

Example2: Solve the recurrence $\boldsymbol{T(n) = T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + n}$

using recursion tree method.

Solution: We always omit floor & ceiling function while solving recurrence. Thus given recurrence can be written as:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \quad \ldots \ldots \ldots \quad (1)$$

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence (1).

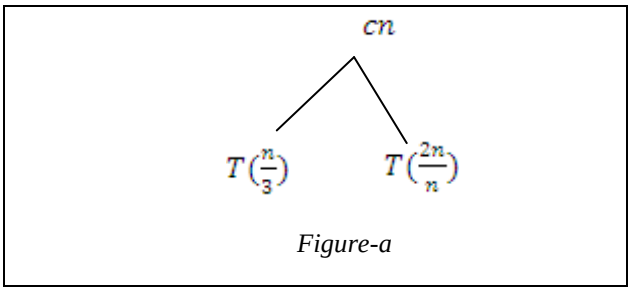


Figure-a

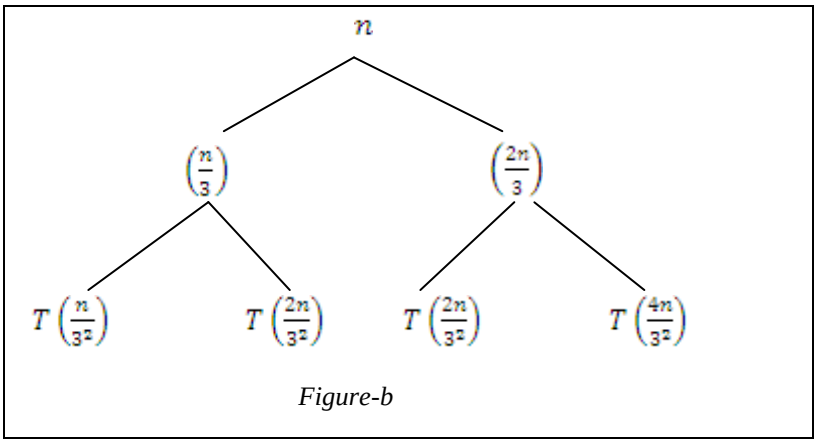


Figure-b

c) In this way, you can extend a tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

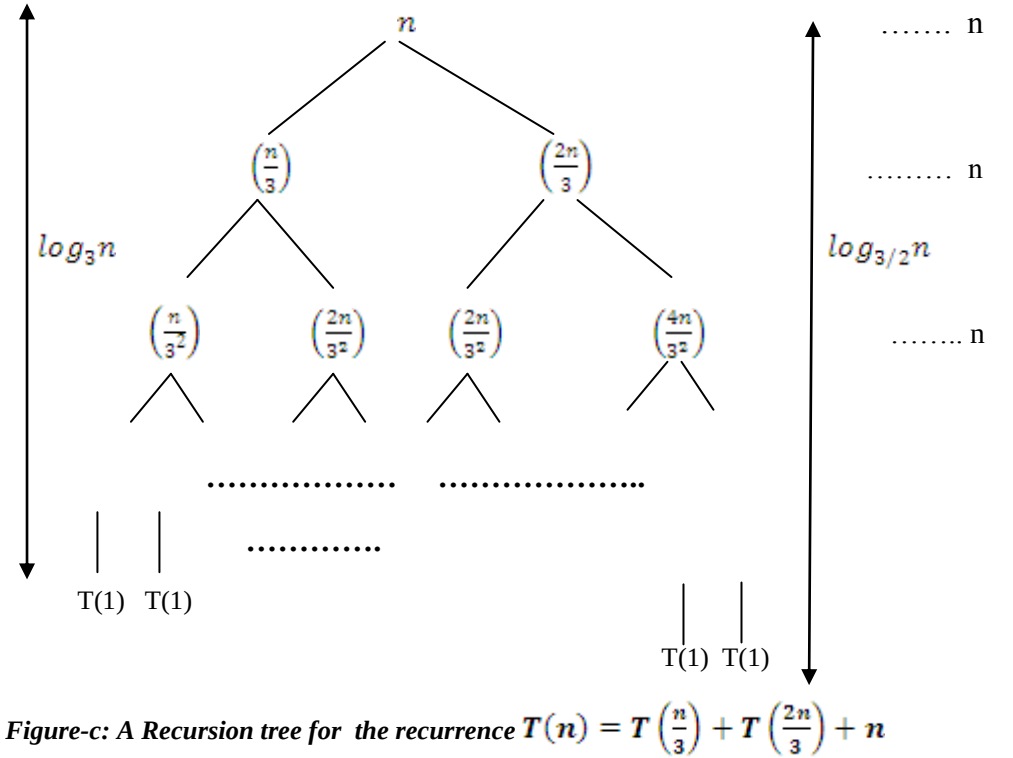


Figure-c: A Recursion tree for the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

Here the longest path from root to the leaf is:

$$n \to \left(\frac{2}{3}\right)n \to \left(\frac{2}{3}\right)^2 n \to \cdots 1$$

$$\left(\frac{2}{3}\right)^k = 1 \implies k = log_{3/2} n \implies Height\ of\ the\ tree.$$

$$n + n + \cdots \ldots \ldots \ldots + n \ (log_{3/2} n\ times)$$

$$\implies n log_{3/2} n = \frac{n log_2 n}{log_2 \frac{3}{2}} = O(n log_2 n) \quad \text{----------} \quad (*)$$

Here the smallest path from root to the leaf is:

$$n \to \frac{n}{3} \to \frac{n}{3^2} \to \cdots \ldots \ldots \to \frac{n}{3^k}$$

$$(n/3)^k = 1 \implies k = log_3 n \implies Height\ of\ the\ tree.$$

$$n + n + \cdots \ldots \ldots \ldots + n \ (log_3 n\ times)$$

$$\implies n log_3 n = \frac{n log_2 n}{log_2 3} = \Omega(n log_2 n) \quad \text{----------} \quad (**)$$

From equation (*) and (**), $since\ T(n) = O(nlog_2 n)\ and\ T(n) = \Omega(nlog_2 n)$, thus we can write:

$$T(n) = \Theta(nlog_2 n)$$

Remark: If $f(n) = O(g(n))\ and\ f(n) = \Omega(g(n))\ then\ f(n) = \Theta(g(n))$

Example3: A recurrence relation for Tower of Hanoi (TOH) problem is $T(n) = 2T(n-1) + 1\ with\ T(1) = 1\ and\ T(2) = 3$. Solve this recurrence to find the solution of TOH problem.

Solution:

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence $T(n) = 2T(n-1) + 1$

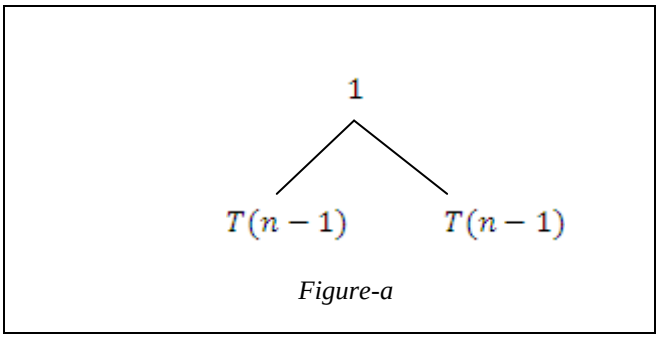


Figure-a

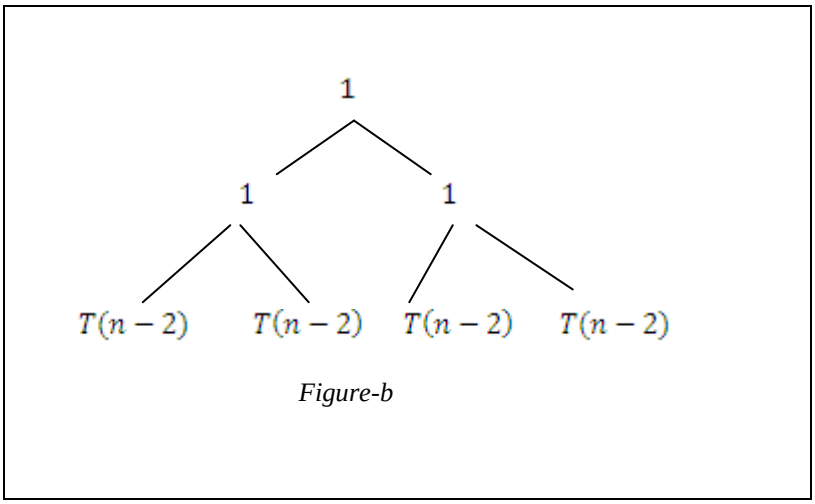


Figure-b

c) In this way, you can extend a tree up to Boundary condition (when problem size becomes 1). That is

$$n \to (n-1) \longrightarrow (n-2) \longrightarrow \cdots \ldots \ldots \longrightarrow \big(n-(n-1)\big)$$

$$\Rightarrow n \to (n-1) \longrightarrow (n-2) \longrightarrow \cdots \ldots \ldots \longrightarrow 2 \longrightarrow 1$$

So the final tree will be looks like:

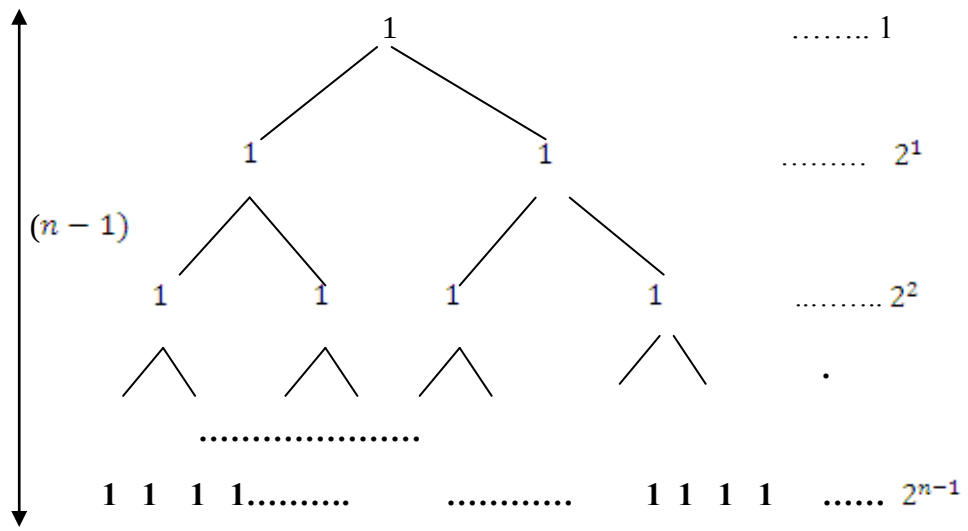


Figure-c: *A Recursion tree for the recurrence* $\boldsymbol{T(n) = 2T(n-1) + 1}$

At last level problem size will be equal to 1 if
$(n-(n-1)) = 1 \Rightarrow Hight\ of\ the\ tree \Rightarrow (n-1)$.

Hence Total Cost of the tree in figure (c) can be obtained by taking column sum upto the height of the tree.

$$T(n) = 1 + 2^1 + 2^2 + \cdots \ldots + 2^{n-1} = \frac{1(2^n-1)}{2-1} = 2^n - 1.$$

Hence the solution of TOH problem is $T(n) = (2^n - 1)$

☞ Check Your Progress 2

Q1: write a recurrence relation for the following recursive functions:

a)

```
Fast_Power(x, n)
{  if (n == 0)
     return 1;
   elseif (n == 1)
     return x;
   elseif ((n%2) == 0)      //if n is even
     return Fast_power(x, n/2) * Fast_power(x, n/2);
  else
     return x * Fast_power(x, n/2) * Fast_power(x, n/2);
}
```

b)

```
Fibnacci(n)
  { if (n == 0)
      return 0;
    if (n == 1)
      return 1;
    else
      return fibnacci(n − 1) + fibnacci(n − 2); }
```

Q.2: Solve the following recurrence using Iteration Method:

a) $T(n) = 3T\left(\frac{n}{2}\right) + n$

b) Recurrence obtained in Q.1 a) part

c) Recurrence obtained in Q.1 b) part

Q.3: Solve the following recurrence Using Recursion tree method

a. $T(n) = 4T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$

b. $T(n) = 3T\left(\frac{n}{2}\right) + n$

c. $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

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

1.9.4 MASTER METHOD

Definition 1: A function $f(n)$ is ***asymptotically positive*** if and only if there exists a real number n such that $f(x) > 0 \; for \; all \; x > n$.

The master method provides us a straightforward and "cookbook" method for solving recurrences of the form
$T(n) = aT\left(\frac{n}{b}\right) + f(n), where \; a \geq 1 \; and \; b > 1$ are constants and $f(n)$ is an asymptotically positive function. This recurrence gives us the running time of an algorithm that divides a problem of size n into a **subproblems** of size $\left(\frac{n}{b}\right)$. The a **subproblems** are solved recursively, each in time $T\left(\frac{n}{b}\right)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. This recurrence is technically correct only when $\left(\frac{n}{b}\right)$ is an integer, so the assumption will be made that $\left(\frac{n}{b}\right)$ is either $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$ since such a replacement does not affect the asymptotic behavior of the recurrence. The value of $a \; and \; b$ is a positive integer since one can have only a whole number of subproblems.

Theorem1: Master Theorem

The Master Method requires memorization of the following 3 cases; then the solution of many recurrences can be determined quite easily, often without using pencil & paper.

Let $T(n)$ be defined on the non negative integers by:

$T(n) = aT\left(\frac{n}{b}\right) + f(n), where\ a \geq 1,\ b > 1\ and\ \frac{n}{b}\ is$ treated as above -----(1)

Then $T(n)$ can be bounded asymptotically as follows:

Case1: If $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$

Case2: If $(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}.logn)$.

Case3: If $f(n) = O(n^{log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Remark: To apply Master method you always compares $n^{log_b a}$ and $f(n)$. If these functions are in the same Θ class, then you multiply by a logarithmic factor to get the run time of recurrence (1) [**Case 2**]. If $f(n)$ is polynomially smaller than $n^{log_b a}$ (by a factor of n^{ϵ}) then $T(n)$ is in the same Θ class as $n^{log_b a}$ (**case 1**). If $f(n)$ is polynomially larger than $n^{log_b a}$ (by a factor of $1/n^{\epsilon}$) then $T(n)$ is in the same Θ class as $f(n)$ (**case 3**). In case 1 and 3, the functions $f(n)$ must be polynomially larger or smaller then $n^{log_b a}$. If $f(n)$ is not polynomially larger or smaller then $n^{log_b a}$, then master method fail to solve a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Thus **master method fails** in following two cases:

a) **If $f(n)$ is asymptotically larger than $n^{log_b a}$, but not pollynomially larger (**by a factor of $1/n^{\epsilon}$) **than $n^{log_b a}$.**
For example $T(n) = 2T\left(\frac{n}{2}\right) + nlogn,$
in which $a = 2, b = 2\ and\ f(n) = nlogn.\ \ n^{log_b a} = n$. Here $f(n)$ is asymptotically larger than
$n^{log_b a} = \mathrm{n}$ (faster growth rate), but it is not polynomially larger than $n^{log_b a} = \mathrm{n}$. In other words
$\frac{f(n)}{n^{log_b a}} = logn\ , which\ is\ asymptotically\ less\ then\ n^{\epsilon}\ for\ some\ \epsilon > 0.$

b) **If $f(n)$ is asymptotically smaller than $n^{log_b a}$, but not pollynomially smaller (**by a factor of n^{ϵ}) **than $n^{log_b a}$:**
For example: $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{logn}$. Here $a = 2, b = 2$ and
$f(n) = \frac{n}{logn}\ and\ \ n^{log_b a} = n.$

Examples of Master Theorem

Example1: Consider the recurrence of $T(n) = 9T\left(\frac{n}{3}\right) + n$, in which a = 9, b=3, $f(n) = n$, $n^{log_b a} = n^2$ and $f(n) = O(n^{log_b a - \epsilon})$, where $\epsilon = 1$. By Master Theorem (case1), we get $T(n) = \Theta(n)$.

Example2: Consider the recurrence of $T(n) = T\left(\frac{2n}{3}\right) + 1$, in which $a = 1, b = \frac{3}{2}, f(n) = n^{log_3 2} = 1$. Since $f(n) = \Theta(n^{log_3 2})$. By Master Theorem (case 2), we get $T(n) = \Theta\left(n^{log_b a}.logn\right) = \Theta(logn)$.

Example3: Consider the recurrence of $T(n) = 3T(n/4) + nlogn$, in which

$a = 3, b = 4, f(n) = nlogn$. Since $n^{log_b a} = n^{log_4 3} \approx n^{0.79}$, Since $f(n) = nlogn = \Omega\left(n^{log_4 3 + \epsilon}\right)$ where $\epsilon \approx 0.21$. Case3 of Master method may be applied. Now check (Regularity Condition)

$af\left(\frac{n}{4}\right) \leq c.f(n) \Rightarrow 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq c.nlogn$, *which is* satisfied for

$c = \frac{3}{4}$, since $c < 1$, so now we can apply mater method (case 3).

$T(n) = \Theta(nlogn)$.

Example4: *Can the Master method be applied to solve recurrence* $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$? Why or why not?

Solution: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$

Here a = 2 b = 2 $f(n) = nlogn$
Now $n^{\log_b a} = n^{\log_2 2} = n$

since $f(n) = nlogn$ is asymptotically larger than $n^{\log_b a} = n$, so it might seem that case 3 should apply. The problem is that it is not polynomially larger.

The ratio $\frac{f(n)}{n^{log_b a}} = \frac{n\log n}{n} = \log n$; is asymptotically less than n^{ϵ} *for any* positive constant $\epsilon > 0$.
So, the recurrence falls into the gap between case 2 and case 3. Thus, the master theorem can not apply for this recurrence.

Example 5: *"The recurrence T(n)=7T*$\left(\frac{n}{2}\right) + n^2$ *describes the running* time of an algorithm A. A competing algorithm A' has a running time of $T'(n) = T\left(\frac{n}{4}\right) + n^2$. What is the largest integer value for '**a**' such that **A'** is asymptotically faster than A?"

Solution: $T(n) = 7T\left(\frac{n}{2}\right) + n^2$

The master method gives us $a = 7, b = 2$ $f(n) = n^2$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.8}$$

It is the first case because $f(n) = n^2 = O(n^{\log 7 - \epsilon})$

where $\epsilon = 0.8$ which gives $T(n) = \Theta(n^{\log_2 7})$

The other recurrence $T'(n) = T\left(\frac{n}{4}\right) + n^2$ is a bit more difficult to analyze because when **a** is unknown it is not so easy to say which of the three cases applied in the master method. But *f(n)* is same in both algorithms which leads us to try the case1.

Applying case1 for the recurrence $T'(n) = T\left(\frac{n}{4}\right) + n^2$ gives

$T(n) = \Theta(n^{\log_4 a})$

For getting the value of **a,** so that A' is asymptotically faster than A:

$$\log_4 a < \log_2 7$$
$$\Rightarrow \frac{\log_2 a}{\log_2 4} < \log_2 7 \quad \left[Using \log \quad property: \log_b a = \frac{\log_c a}{\log_c b}\right]$$
$$\Rightarrow \frac{\log_2 a}{2} < \log_2 7$$
$$\Rightarrow \log_2 a < 2\log_2 7$$
$$\Rightarrow \log_2 a < \log_2 49$$
$$\Rightarrow a < 49$$

In other words, A' asymptotically faster than A as long as $a < 49$ (hence $a = 48$). The other cases in the master method do not apply for *a > 48.*

Hence **A'** is asymptotically faster than A up to *a=48.*

☞ Check Your Progress 3

(Objective Questions)

Q.1: Which of the following recurrence can't be solved by Master method?

1) $T(n) = 3T(n/2) + n \log n$

2) $T(n) = 4T(n/2) + n^2 \log n$

a) Only 2 b) only 1 c) both 1) and 2) d) both 1) and 2) can be solved

Q.2: suppose $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(0) = T(1) = 1.$ Which of the following is false?

a) $T(n) = O(n \log n)$ b) $T(n) = \Theta(n \log n)$

c) $T(n) = \Omega(n^2)$ d) $T(n) = O(n^2)$

Q.3 The time complexity of the following function is (assume $n > 0$)

$int\ fib(int\ n)$

$\{ if(n == 1)\ return\ 1;$
$else\ return(fib(n-1) + fib(n-1));$
$\}$

a) $O(n)$ b) $O(nlogn)$ c) $TO(n^2)$ d) $O(2^n)$

Q.4: The solution of the recurrence $T(n) = T\left(\left\lceil\frac{n}{2}\right\rceil\right) + 1$ is

a) $O(n)$ b) $O(logn)$ c) $TO(nlogn)$ d) $O(n^2)$

Q.5: Write all the 3 cases of Master method to solve a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n).$

Q.6: Use Mater Theorem to give the tight asymptotic bounds of the following recurrences:

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

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

c. $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

d. $T(n) = 2T\left(\frac{n}{2}\right) + n\sqrt{n}$

e. $T(n) = 4T\left(\frac{n}{3}\right) + n^2$

f. $T(n) = 8T\left(\frac{n}{2}\right) + 3n^2$

Q.7: Write a condition when Master method fails to solve a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n).$

Q8: Can Master Theorem be applied to the recurrence of

$T(n) = 4T\left(\frac{n}{2}\right) + n^2logn$? Why and why not? Give an asymptotic upper bound of the recurrence.**?**

1.10 SUMMARY

1. "**Analysis of algorithm**" is a field in computer science whose overall goal is an understanding of the complexity of algorithms (in terms of time Complexity), also known as ***execution time*** & storage (or space) requirement taken by that algorithm.
2. **An Algorithm** is a well defined computational procedure that takes input and produces output.

3. An *algorithm* is a finite sequence of steps, each of which has a clear meaning and can be performed with a fine amount of effort in an finite length of time.

4. Two important ways to characterize the effectiveness of an algorithm are its *space complexity* and *time complexity*.

5. *Space complexity* of and algorithm is the number of elementary objects that this algorithm needs to store during its execution. The space occupied by an algorithm is determined by the number and sizes of the variables and data structures used by the algorithm.

6. Number of machine instructions which a program executes during its running time is called *time complexity*.

7. There are 3 cases, in general, to find the time complexity of an algorithm:
   1. **Best case:** The minimum value of $f(n)$ for any possible input.
   2. **Worst case:** The maximum value of $f(n)$ for any possible input.
   3. **Average case:** The value of $f(n)$ which is in between maximum and minimum for any possible input. Generally the Average case implies the ***expected value*** of $f(n)$.

8. *Asymptotic analysis* of algorithms is a means of comparing relative performance.

9. There are 3 Asymptotic notations used to express the time complexity of an algorithm *O, Ω and Θ* notations.

10. ***O*-notation:** Given functions *f(n)* and *g(n)*, we say that *f(n)* is *O(g(n))* if there exist two positive constants c > 0 and $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$. Big-Oh notation gives an upper bound on the growth rate of a function.

11. ***Ω*-notation**: Given functions *f(n)* and *g(n)*, we say that *f(n)* is *Ω(g(n))* if there exist 2 positive constants c > 0 and $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for all n, $n \leq n_0$. Big-Omega notation gives a lower bound on the growth rate of a function.

12. ***Θ*-notation**: Let *f(n)* and *g(n)* be two asymptotically positive real-valued functions. We say that *f(n)* is *Θ(g(n))* if there is an integer n_0 and two positive real constants c_1 and c_2 such that $c_1\, g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

13. When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A *recurrence relation* is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, a recurrence relation for Binary Search procedure is given by:

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & if\ n > 1 \end{cases}$$

14. There are three basic methods of solving the recurrence relation:
    1. The Substitution Method
    2. The Recursion-tree Method
    3. The Master Theorem
15. *Master method* provides a "cookbook" method for solving recurrences of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n) where\ a\ \geq 1\ and\ b > 1$ are constants and $f(n)$ is an asymptotically positive function.
16. In master method you have to always compare the value of $f(n)$ with $n^{log_b a}$ to decide which case is applicable. If $f(n)$ is asymptotically smaller than $n^{log_b a}$, then case1 is applied. If $f(n)$ is asymptotically same as $n^{log_b a}$, then case2 is applied. If $f(n)$ is asymptotically larger than $n^{log_b a}$, and if
    $af\left(\frac{n}{b}\right) \leq c.f(n)\ for\ some\ c < 1,$ then case3 is applied.
17. Master method is sometimes fails (either case1 or case 3) to solve a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n),$ as discussed above.

1.11 SOLUTIONS/ANSWERS

Check Your Progress 1

Answer 1-b, 2-a, 3-d, 4-c

Answer 5:

An Algorithm is a well defined computational procedure that takes input and produces output. Or we can say that an Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output. Any Algorithm must satisfy the following criteria (or Properties)

1. **Input**: It generally requires finite no. of inputs.
2. **Output:** It must produce at least one output.
3. **Uniqueness**: Each instruction should be clear and unambiguous
4. **Finiteness:** It must terminate offer a finite no. of steps.

Answer 6: There are basically 5 fundamental techniques are used to design an algorithm efficiently:

Algorithm design techniques	**Examples**
Divide and Conquer	Binary search, Merge sort
Greedy Method	Knapsack problem, Minimum cost spanning tree problem (Kruskal's and Prim's Algorithm)

Dynamic Programming	All Pair Shortest Path Problem (Floyed Algorithm), Chain Matrix multiplication.
Backtracking	N-Queen's problem, Sum-of-subset problem
Branch and Bound	Assignment problem, TSP (Travelling salesman problem)

Answer 7:

Testing a program consists of two phases: debugging and profiling (or performance measurement). Debugging is the process of executing programs on sample data sets to find whether wrong results occur and, if so, to correct them. Debugging can only point to the presence of errors.

Profiling is the process of executing a correct program on data sets and measuring its time and space. These timing figures are used to confirm the previous analysis and point out logical errors. These are useful to judge a better algorithm.

Answer 8:

There are 3 basic asymptotic notations (O,Ω and Θ) used to express the time complexity of an algorithm.

O (Big-Oh) notation	$O(g(n) = \{f(n)$: there exist a positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. It express upper bound of a function $f(n)$.
Ω (Big omega) notation	$\Omega(g(n) = \{f(n)$: there exist a positive constants c and n_0 such that $0 \leq f(n) \geq cg(n)$ for all $n \geq n_0\}$. It express lower bound of a function $f(n)$.
Θ (Theta) notation	$\Theta(g(n) = \{f(n)$: there exist a positive constants c_1, c_2, and and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$. It express both upper and lower bound of a function $f(n)$.

Solution 9:

Time complexity: The number of machine instructions which a program executes during its running time is called its *time complexity*. This number depends primarily on the size of the program's input.

Time taken by a program is the sum of the compile time and the run time. In time complexity, we consider run time only. The time required by an algorithm is determined by the number of elementary operations.

The following primitive operations that are independent from the programming language are used to calculate the running time:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation
- Comparing two variables
- Indexing into a array of following a pointer reference
- Returning from a function

The following fragment shows how to count the number of primitive operations executed by an algorithm.

```
int sum(int n)
{
int i,sum;
1. sum = 0;                          //add 1 to the time count
2. for(i = 0; i < n;i ++)            //add n + 1 to the time count
   {
3.   sum = sum + i * i;              //add n to the time count
   }
4.   return sum;                     //add 1 to the time count
```

This function returns the sum from $i = 1$ to n of i squared, $i.e. sum = 1^2 + 2^2 + \cdots \ldots + n^2$.

To determine the running time of this program, we have to count the number of statements that are executed in this procedure. The code at line 1 executes 1 time, at line 2 the **for loop** executes $(n + 1) time$, Line 3 executes n times, and line 4 executes 1 time. Hence the sum is $= 1 + (n + 1) + n + 1 = 2n + 3$.

In terms of O-notation this function is $O(n)$.

Solution 10:

(a) $(4n^2 + 7n + 12) \leq c.n^2 \ldots \ldots (1)$

for c = 5 and $n \leq 9$; the above inequality (1) is satisfied.

Hence $4n^2 + 7n + 12 = O(n^2)$.

(b) By using basic definition of Big – "oh" Notation:

$$logn + \log(logn) \leq C.\log n \quad \ldots \ldots (1)$$

For C = 2 and $n_0 = 2$, we have

$$log_2 2 + \log(log_2 2) \leq 2.log_2 2$$

$\Rightarrow \quad 1 + \log(1) \leq 2$

$\Rightarrow \quad 1 + 0 \leq 2$

$\Rightarrow$ satisfied for c = 2 and $n_{0=}2$

$\Rightarrow \quad \log n + \log(\log n) = O(\log n).$

(c) $3n^2 + 7n - 5 = Q(n^2)$;

To show this, we have to show:

$C_1.n^2 \leq 3n^2$+7n – 5 $\leq C_1.n^2$ …….. (*)

(i) L.H.S inequality:

$C_1.n^2 \leq 3n^2$+7n – 5 …..(1)

This is satisfied for $C_1 = 1$ and n ≥ 2

(ii) R.H.S inequality:

$3n^2$ + 7*n* – 5 ≤ $C_2 \, . \, n^2$…..(2)

This is satisfied for $C_1 = 1$ and n ≥ 1

$\Rightarrow$ inequality $(*)$ is simultaneously satisfied for

$C_1 = 1, C_2 = 10$ and $n \geq 2$

(d) $2^{n+1} \leq C.2^n \Rightarrow \; 2^{n+1} \leq 2.2^n$

(e) $2^{2n} \leq C.2^n$

$\Rightarrow \quad 4^n \leq 2.2^n \ldots\ldots(1)$

No value of C and n_0 Satisfied this in equality (1)

$\Rightarrow \quad 2^{2n} \neq O(2^n).$

(f) No; *f(n) = O(g(n))* does not implies *g(n) = O(f(n)*

Clearly *n = O*(n^2), but $n^2 \neq O(n)$

(g) To prove this, we have to show that

$$C_1.(f(n)+g(n) \leq \max\{f(n), g(n)\} \leq C_2(f(n)+g(n) \ldots\ldots\ldots\ldots (*)$$

1) L.H.S inequality:

$C_1.(f(n)+g(n) \leq \max\{f(n), g(n)\} \ldots\ldots\ldots\ldots (1)$

Let *h(n)* = max {*f(n), g(n)*} = $\begin{cases} f(n) & if\ f(n) > g(n) \\ g(n) & if\ g(n) > f(n) \end{cases}$

$\therefore C_1.(f(n)+g(n) \leq f(n) \ldots\ldots\ldots (1)$

[Assume$\max\{f(n), g(n)\} = f(n)$]

$for\ C_1 = \frac{1}{2}\ and; n \geq 1$, this inequality (1)is satisfied:

$since\ \frac{1}{2}(f(n)+g(n) \leq f(n)$

$\Rightarrow \quad f(n)+g(n) \leq 2f(n)$

$\Rightarrow \quad f(n)+g(n) \leq f(n)+f(n) \quad [\because f(n) > g(n)]$

$\Rightarrow$ satisfied for $C_1 = \frac{1}{2}$ and n ≥ 1

2) R.H.S. inequality

$\max\{f(n), g(n)\} \leq C_1.(f(n)+g(n) \ldots\ldots (2)$

This inequality (2) is satisfied for $C_2 = 1\ and\ n \geq 1$

$\Rightarrow$ inqualtiy $(*)$is simultaneously satisfied for

$C_1 = \frac{1}{2}, C_2 = 1$ and n ≥ 1

Remark : Let *f(n)* = *n* and *g(n)* = n^2 ;

then $\max\{n, n^2\} = \Theta(n+n^2)$

$\Rightarrow n^2 = \Theta(n^2)$; which is TRUE(by definition of Θ)

h) $NO;\ f(n) = O(g(n)$does not implies $2^{f(n)} = O(2^{g(n)})$;

we can prove this by taking a counter Example;

Let $f(n) = 2n$ and g(n) = n, we have

$2^{2n} = O(2^n)$; which is not TRUE [since $2^{2n} = 4^n \neq O(2^n)$].

i) $No, f(n) + g(n) \neq \Theta(\min\{f(n), g(n)\})$ We can prove this by taking counter example.

Let $f(n) = 2n$ and $g(n) = n^2$, then $(n + n^2) \neq \Theta(n)$

j) $(33n^3 + 4n^2) \geq C.n^4$; There is no positive integer for C and n_0 which satisfy this inequality. Hence $(33n^3 + 4n^2) \neq C.n^4$.

k) $\boldsymbol{f(n) + g(n) = 3n^2 - n + 4 + nlogn + 5 = h(n)}$

$By\ O - notation\ \ h(n) \leq cn^2;$

This is true for $\boldsymbol{c = 4}$ and $\boldsymbol{n_0 \geq 4}$

Check Your Progress 2:

Solution 1: a)

At every step the problem size reduces to half the size. When the power is an odd number, the additional multiplication is involved. To find a time complexity of this algorithm, let us consider the ***worst case,*** that is we assume that at every step additional multiplication is needed. Thus total number of operations T(n) will reduce to number of operations for n/2, that is T(n/2) with three additional arithmetic operations(In odd power case: 2 multiplication and one division). Now we can write:

$T(n) = 1 \ \ if\ n = 0\ or\ 1$

$T(n) = T\left(\frac{n}{2}\right) + 3 \ \ if\ n \geq 2$

Instead of writing exact number of operations needed by the algorithm, we can use some constants. The reason for writing this constant is that we are always interested to find "asymptotic complexity" instead of finding exact number of operations needed by algorithm, and also it would not affect our complexity also.

$$T(n) = \begin{cases} T(1) = a & if\ n = 0\ or\ n = 1 \quad (\boldsymbol{base\ case}) \\ T\left(\frac{n}{2}\right) + b & if\ n \geq 2 \quad (\boldsymbol{Recursive\ step}) \end{cases}$$

b)

$$T(n) = \begin{cases} a & if\ n = 0\ or\ n = 1 \quad (\textbf{\textit{base case}}) \\ T(n-1) + T(n-2) + b & if\ n \geq 2 \quad (\textbf{\textit{Recursive step}}) \end{cases}$$

Solution2: a)

$$T(n) = n + 3T\left(\frac{n}{2}\right)$$

$$= n + 3\left\{\frac{n}{2} + 3T\left(\frac{n}{4}\right)\right\} = n + \frac{3n}{2} + 3^2T\left(\frac{n}{4}\right)$$

$$= n + 3.\frac{n}{2} + 3^2.\frac{n}{4} + T\left(\frac{n}{8}\right)$$

(By substituting $T\left(\frac{n}{4}\right)$ $in\ equation\ 3)$; In this way we get the final GP series as:

$$= \underbrace{(n + \frac{3n}{2} + \frac{3^2n}{4} + \cdots \ldots \frac{3^{k-1}n}{2^{k-1}}}_{k=log_2n\ terms\ (total)} + T(\tfrac{n}{2^k})$$

$$= n + \frac{3}{2}n + \left(\frac{3}{2}\right)^2 n + \cdots \ldots \ldots + \left(\frac{3}{2}\right)^{log_2n-1} n + T(1)$$

$$= n\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots \ldots \ldots + \left(\frac{3}{2}\right)^{log_2n-1}\right) + a$$

$$= n.\frac{[(\frac{3}{2})^{log_2 n} - 1]}{(\frac{3}{2}-1)}$$ [By using GP series sum formula $S_n = \frac{a(x^n-1)}{x-1}$

$$= \frac{n[n^{log_2\frac{3}{2}}-1]}{(1/2)}$$ [Using log property $\boldsymbol{a^{log_b n} = n^{log_b a}}$]

$$= 2n[n^{log_2 3 - log_2 2} - 1] = 2n[n^{log_2 3 - 1} - 1] = 2n\left[\frac{n^{log_2 3}}{n} - 1\right]$$

$$= 2n^{log_2 3} - 2n = O(n^{log_2 3})$$

Thus $T(n) = O(n^{log_2 3})$

b) $T(n) = O(log_2 n)$

c) $T(n) = O(\frac{1+\sqrt{5}}{2})^n$

Solution 3 (a) : The recursion tree for the given recurrence is:

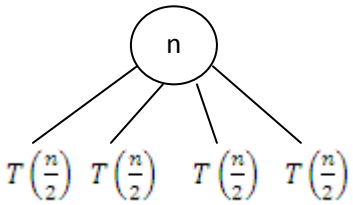


Figure a

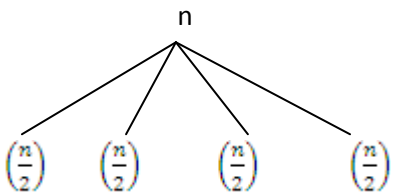


Figure b

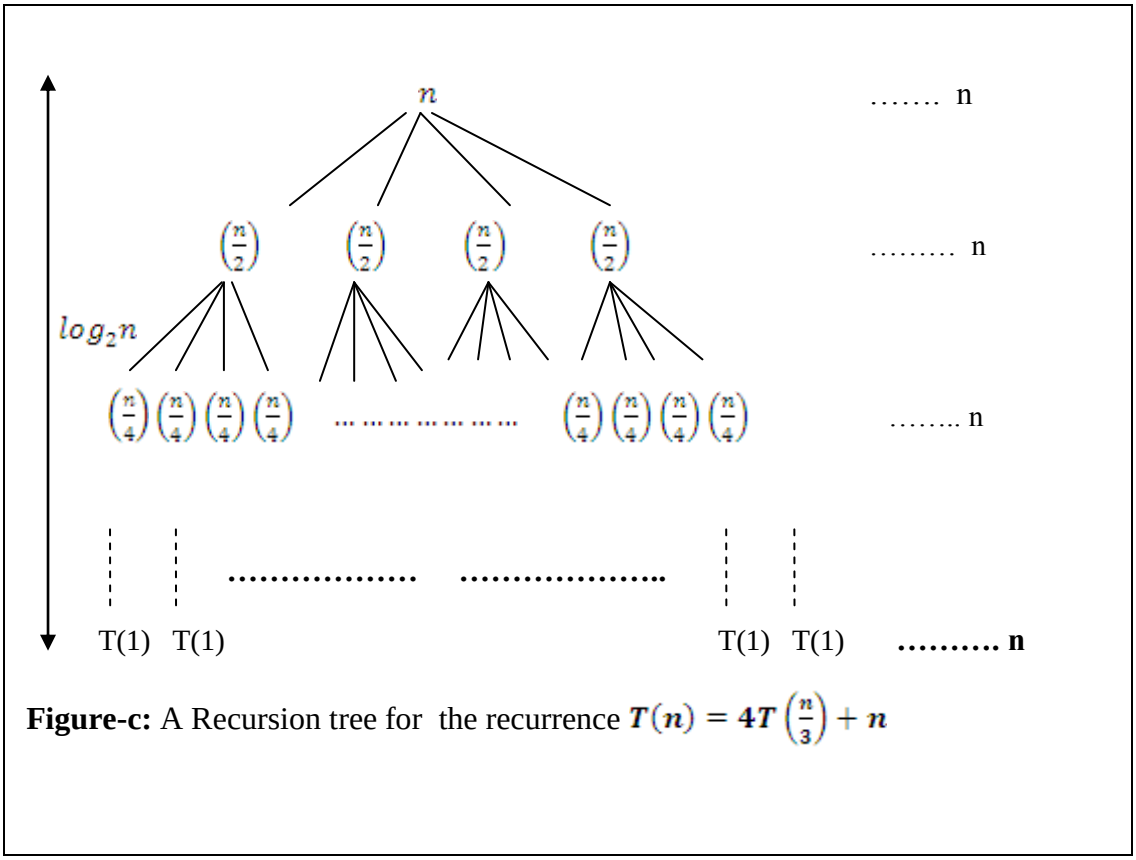


Figure-c: A Recursion tree for the recurrence $\boldsymbol{T(n) = 4T\left(\frac{n}{3}\right) + n}$

We have $Total = n + 2n + 4n + \cdots \log_2 n \text{ times}$

$= n(1 + 2 + 4 + \cdots \log_2 n \text{ times})$

$$= n\frac{(2^{\log_2 n} - 1)}{2-1} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

$$\therefore T(n) = \theta(n^2)$$

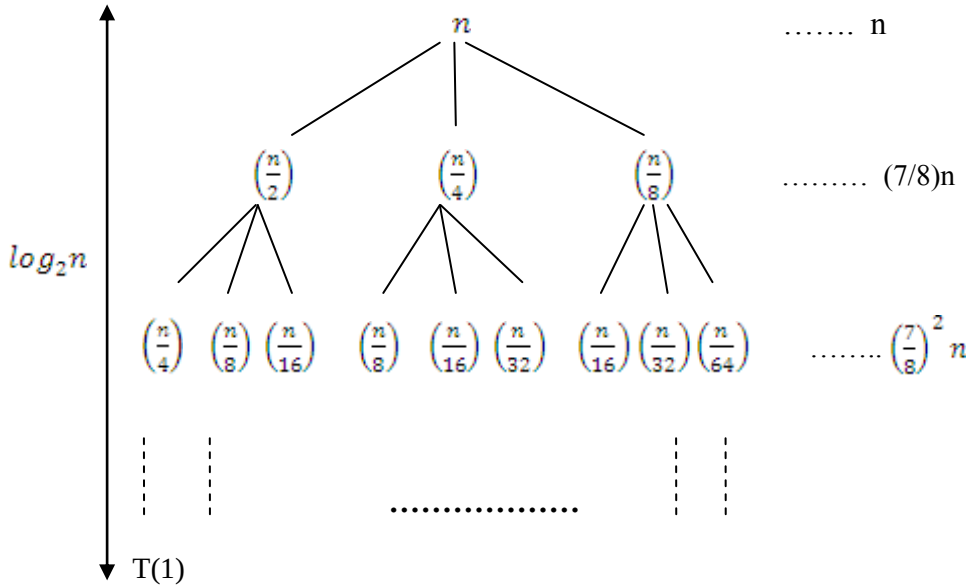
Solution (b) : Refer solution 3(a)

Solution (c): Refer solution 3(a)

Solution (d)

$T=(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$ *by recursion tree method*

Solution. The given recurrence has the following recursion tree:



$$T(n) \leq n + \frac{7}{8}n + \left(\frac{7}{8}\right)^2 n + \cdots$$

$$\leq n\left[1 + \frac{7}{8} + \left(\frac{7}{8}\right)^2 + \cdots\right]$$

$$\leq n.\frac{1}{1-\frac{7}{8}}$$

$$\leq 8n$$

$$T(n) = \Theta(n)$$

Check Your Progress 3:

(Objective Questions)

Answers: 1-a, 2-c,3-d, 4-b

Solution5: The following 3 cases are used to solve a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n),\ where\ a \geq 1\ and\ b > 1.$

Case1: If $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$

Case2: If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}.logn)$.

Case3: If $f(n) = O(n^{log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Solution6:

a) In a recurrence $T(n) = 4T\left(\frac{n}{2}\right) + n$, a = 4, b=2, $f(n) = n$, $n^{log_b a} = n^2$. Now compare $f(n)$ $with$ $n^{log_b a}$. $since$ $f(n) = O(n^{log_b a - \epsilon})$, where $\epsilon = 1$. By Master Theorem case1 we get $T(n) = \Theta(n^2)$.

b): $T(n) = 4T\left(\frac{n}{2}\right) + n^2$; in which $a = 4, b = 2,$ $f(n) = n^2\ and\ n^{log_b a} = n^2$. Now compare $f(n)$ $with$ $n^{log_b a}$; since $f(n) = n^2 = \Theta(n^{log_b a})$. Thus By Master Theorem (case 2), we get $T(n) = \Theta(n^{log_b a}.logn) = \Theta(n^2 logn)$.

c) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$; in which $a = 4, b = 2,$ $f(n) = n^3\ and\ n^{log_b a} = n^2$. Now compare $f(n)$ $with$ $n^{log_b a}$; Since $f(n) = n^3 = \Omega(n^{2+\epsilon})$ where ϵ=1. $\Longrightarrow$Case3 of Master method may be applied.

Now check (Regularity Condition)

$af\left(\frac{n}{b}\right) \leq c.f(n) \Rightarrow 4\left(\frac{n}{2}\right)^3 \leq c.n^3, \Longrightarrow \frac{1}{2}n^3 \leq cn^3;$ $which\ is$ satisfied for $c = \frac{1}{2};$ since $c < 1$, so now we can apply mater method (case 3).

$T(n) = \Theta(f(n)) = \Theta(n^3)$.

d) *T(n)*= $4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$

Here a = 4 b = 2 *f(n)*= $n^2\sqrt{n} = n^{5/2}$

$n^{log_b a} = n^{log_2 2} = n$; Now compare $f(n)$ $with$ $n^{log_b a}$.

Since *f(n)*= $n^2\sqrt{n} = n^{5/2} = \Omega(n^{1+\epsilon})$. *Hence* Case3 of Master method may be applied.

Now check (Regularity Condition):

$2f\left(\frac{n}{2}\right) \leq c.f(n) \Rightarrow 2\left(\frac{n}{2}\right)^{\frac{5}{2}} \leq c.n^{5/2}$

$\Rightarrow \frac{2}{2\sqrt{2}}n^{5/2} \leq c.n^{5/2}$; *which is* satisfied for

$c = \frac{1}{\sqrt{2}}$, since $c < 1$, so now we can apply mater method (case 3).

$T(n) = \Theta(n2\sqrt{n})$.

e) $T(n) = 4T(n/3) + n^2$

Here a = 4 b = 3 f(n) = n^2

$n^{\log_b a} = n^{\log_3 4} = n^{1.26}$

$\therefore\ n^2 = \Omega(n^{\log_3 4+\epsilon})\ for\ \epsilon > 0$

Now check **(regularity condition) :**

$4f\left(\frac{n}{3}\right) \leq c.f(n)$ for c < 1

$\frac{4.n^2}{9} \leq cn^2$; here c=$c = \frac{4}{9} < 1$. Hence Case3 of master method is applied. So *T(n) = Θ(f(n)) = Θ(n^2).*

f) $T(n) = 8T(n/2) + 3n^2$

Where n is an integer power of 2 and greater than 1.

Here *a = 8 b* = 2 f(n) = $3n^2$

Now $n^{\log_b a} = n^{\log_2 8} = n^3$

$3n^2 = O(n^{\log_b a-\epsilon}) = O(n^{3-\epsilon})\ \ for\ \epsilon > 0$; Case1 is applied.

$\therefore$ *T(n) = Θ*($n^{\log_b a}$) = $\Theta(n^3)$

Solution7: Refer two fail conditions given of master method.

Solution8 . T(n)4T(n/2) + n^2log n

Here a = 4 b = 2 *f(n) =* n^2 log n

Now $n^{\log_b a} = n^{\log_2 4} = n^2$

since f(n) = n^2log *n* is asymptotically larger than $n^{\log_b a} = n^2$, so it might seem that case 3 should apply. The problem is that it is not polynomially larger.

The ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \log n}{n^2}$ = log *n*

Is asymptotically less than n^{ϵ} *for any* positive constant ϵ.

So, the recurrence falls into the gap between case 2 and case 3. Thus, the master theorem can not apply for this recurrence.

1.12 FURTHER READINGS

1. *Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson (PHI)*
2. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).
3. *Algoritmics: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).
4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).
5. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).
6. *Fundamentals of Computer Algorithms,* E. Horowitz & S. Sahni: (Galgotia Publications).
7. *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).
8. *Programming Languages* (*Second Edition*) ─ *Concepts and Constructs,* Ravi Sethi: (Pearson Education, Asia, 1996).