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: $i = j$ *or* $i \leftarrow j$

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

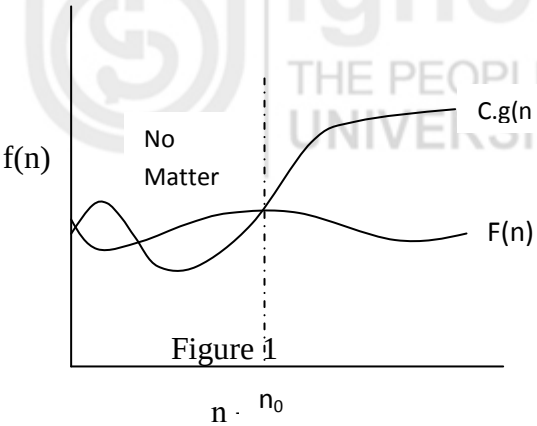


Figure 1

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

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

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

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

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

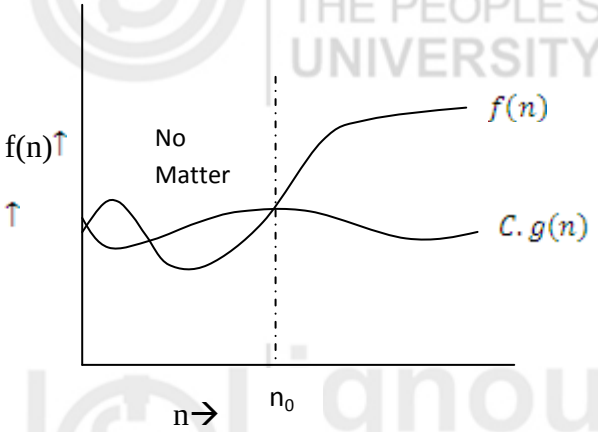
1.7.2 THE NOTATION Ω (BIG 'Omega')

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

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

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



f(n) = Ω g(m)

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

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

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

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

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

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

Solution:

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

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

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

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

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

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

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

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

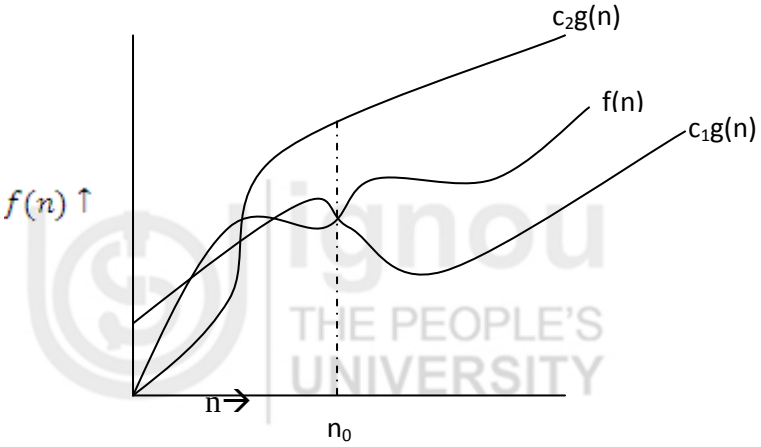
We say that the function $f(n) = \Theta(g(n))$ [read as "f of n is Theta" of g of n"], if and only if there exist three positive constants $C_1,\ C_2\ and\ n_0$ such that

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

(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

$f(n) = \Theta(n^2) \Rightarrow C_1.n^2 \leq 2n^3 + 3n^2 + 1 \leq C_2.n^2$ $for\ all\ n \geq n_0$ ...(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, Ω and Θ.

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

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

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:

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

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

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

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: $Example\left(\frac{n}{2}\right)$
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 $\left\lfloor \frac{n}{2} \right\rfloor$, which causes $x = x + 1$ *to be* executed $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ additional times. Thus we obtain the recurrence relation as:

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

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: $return\ 3 * f\left(\frac{n}{2}\right) + f\left(\frac{n}{2}\right) + 5;$
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:

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

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:

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

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

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

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

$\leq cnlogn - cn + n$

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

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

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

Solution: Here

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

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

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)

$= \cdots\cdots\cdots$

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

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

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

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

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

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:

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

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:

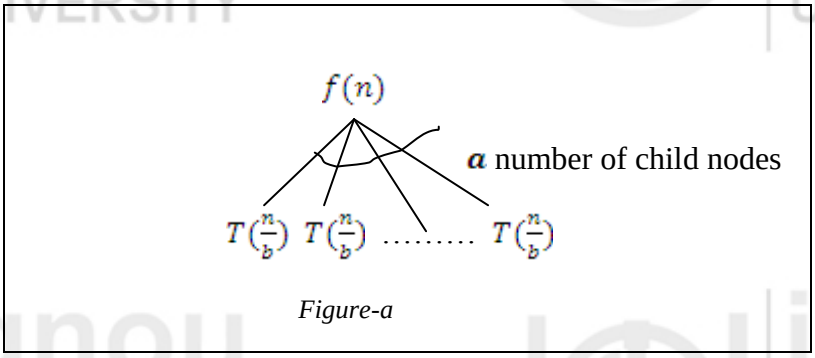


Figure-a

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:

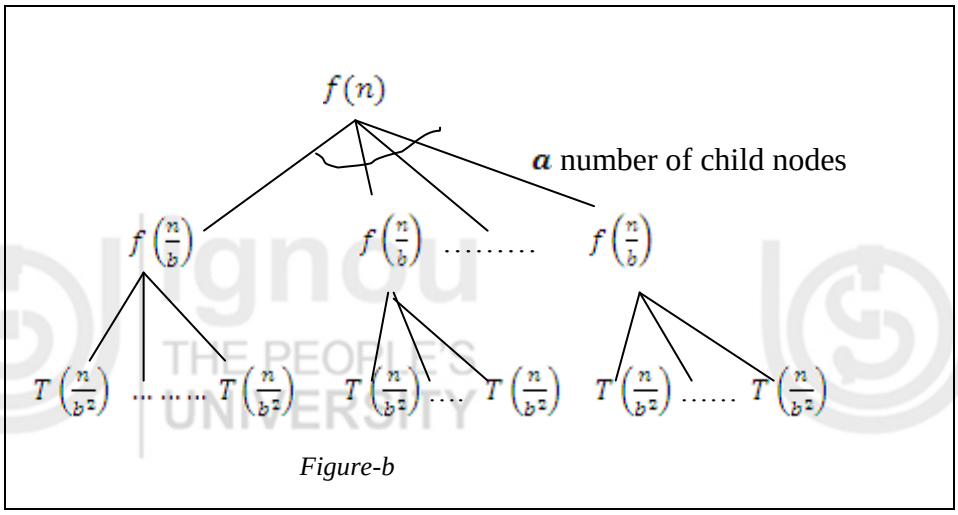


Figure-b

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:

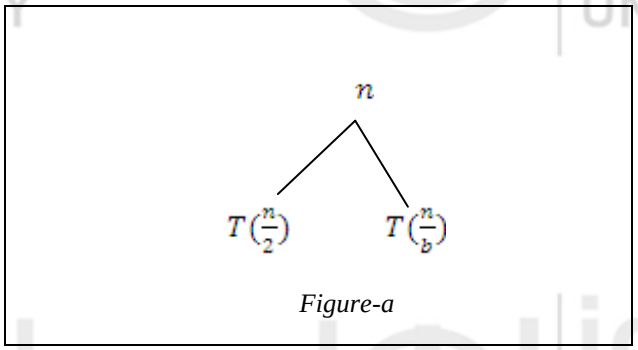


Figure-a

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:

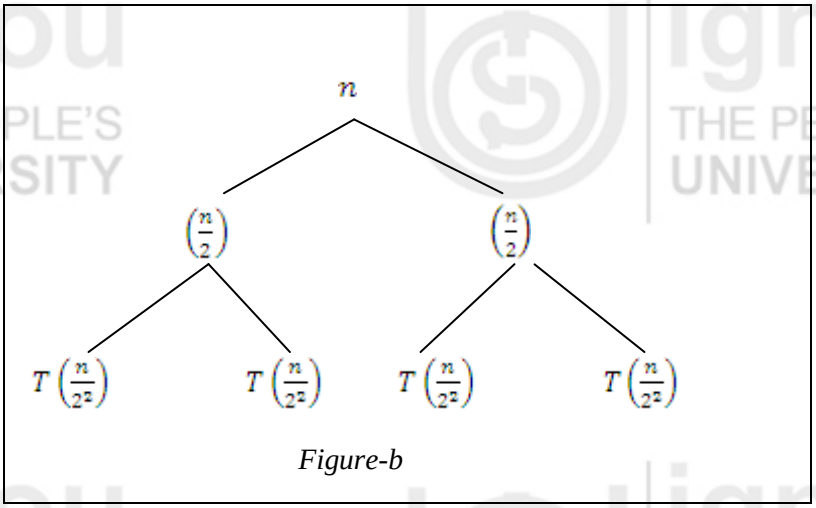


Figure-b

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:

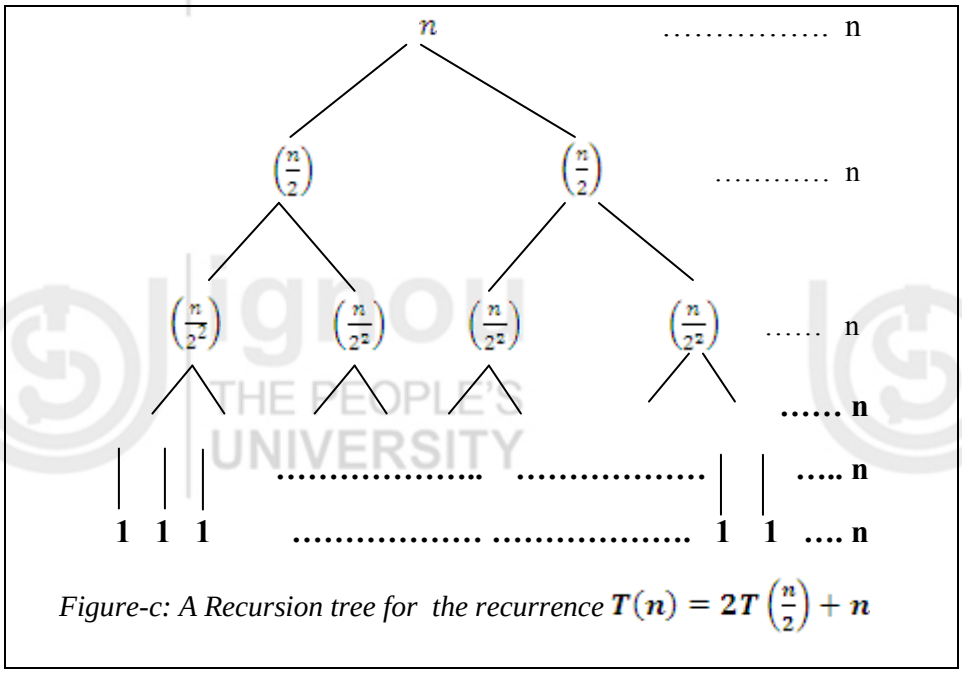


Figure-c: A Recursion tree for the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$

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).

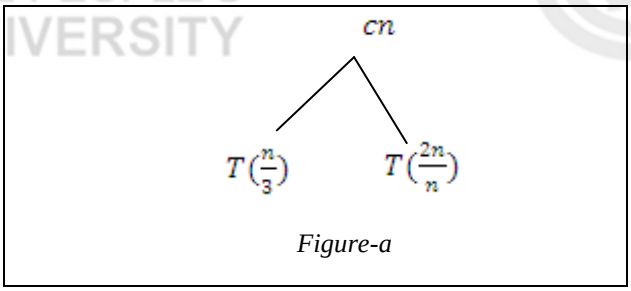


Figure-a

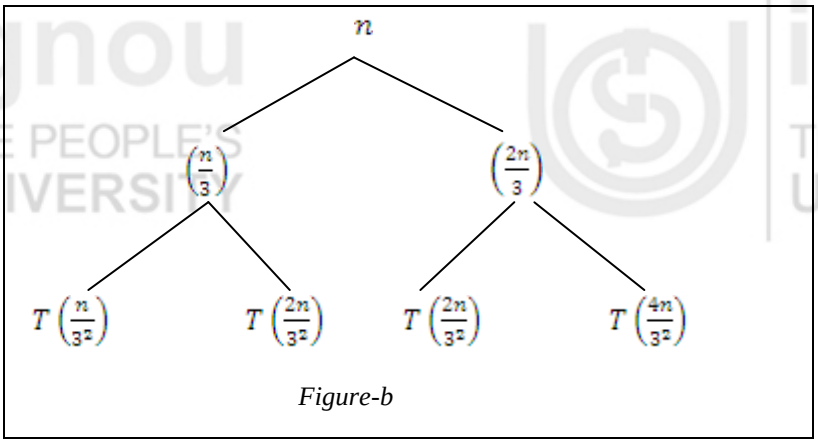


Figure-b

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:

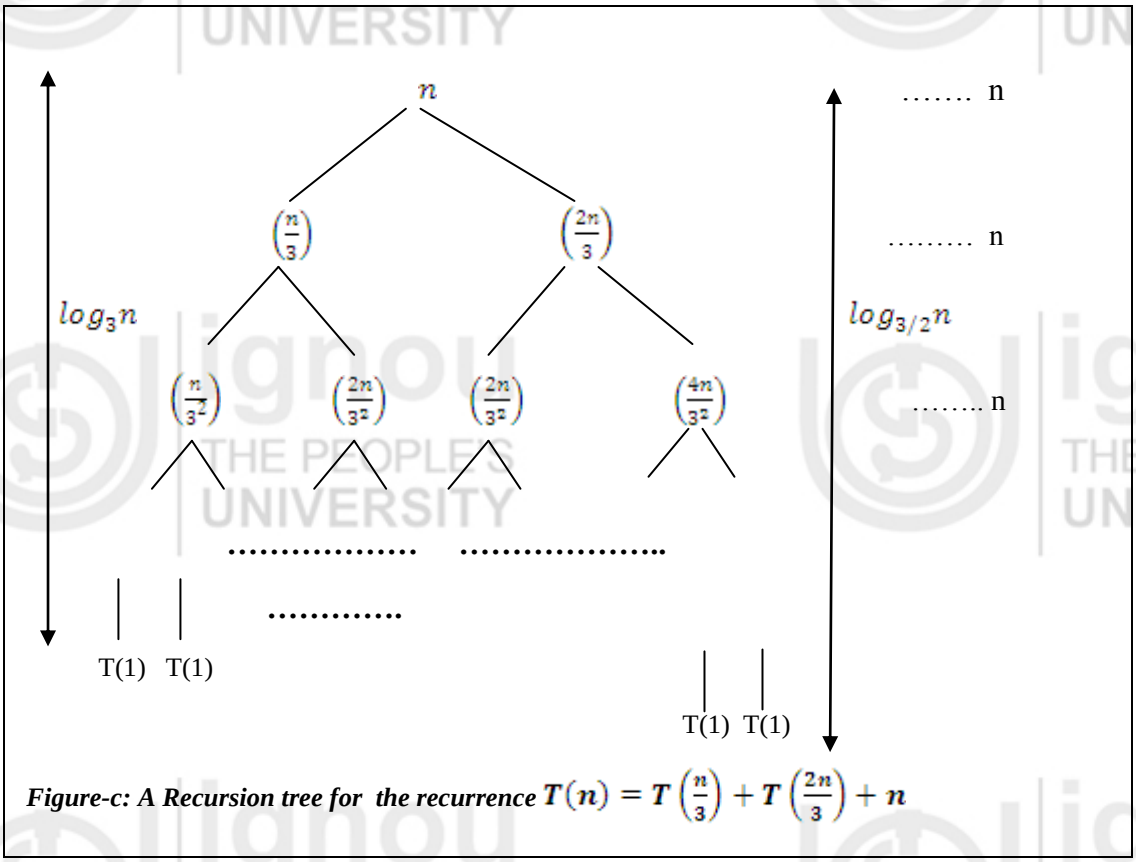


Figure-c: A Recursion tree for the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

Here the longest path from root to the leaf is:

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

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

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

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

Here the smallest path from root to the leaf is:

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

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

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

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

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

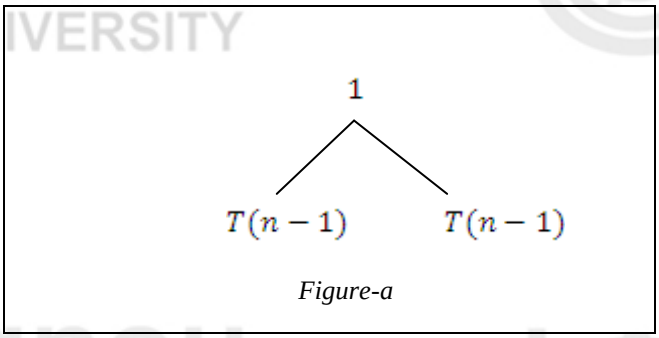


Figure-a

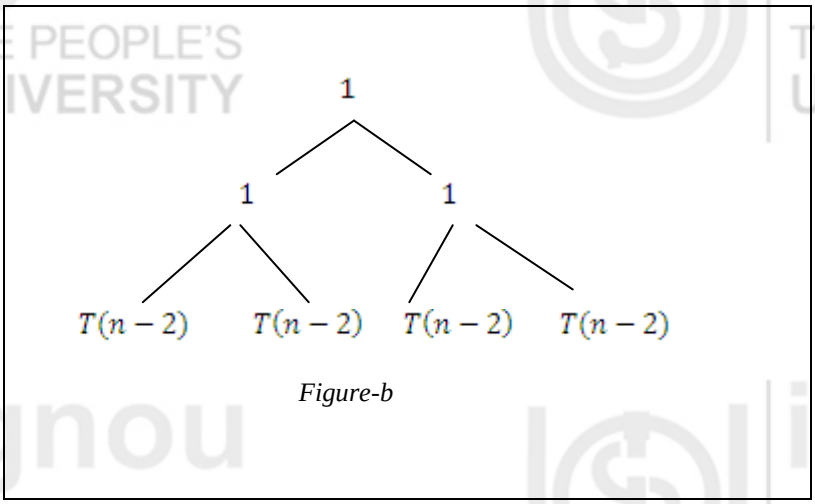


Figure-b

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

$$n \to (n-1) \to (n-2) \to \cdots\cdots\cdots \to (n-(n-1))$$

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

So the final tree will be looks like:

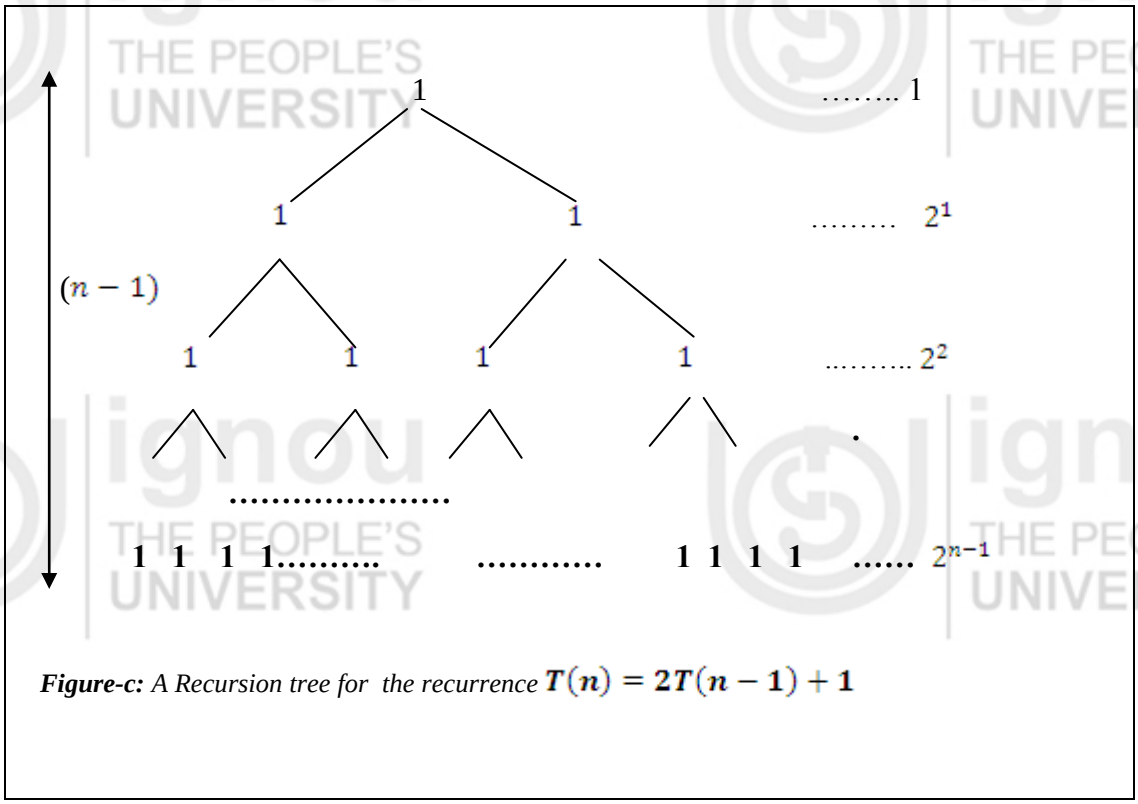


Figure-c: *A Recursion tree for the recurrence* $\boldsymbol{T(n) = 2T(n-1) + 1}$

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
$n^{log_b a} = n$ (faster growth rate), but it is not polynomially larger than $n^{log_b a} = n$. 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 $f(n) = n^2 = O(n^{\log 7 - \varepsilon})$

where $\varepsilon = 0.8$ 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

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

$3n^2 + 7n - 5 \leq C_2 . n^2$.....(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

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

1) L.H.S inequality:

$C_1.(f(n)+g(n) \le \max\{f(n), g(n)\} \dots\dots\dots\dots (1)$

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

$\therefore C_1.(f(n)+g(n) \le f(n) \dots\dots\dots\dots (1)$

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

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

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

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

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

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

2) R.H.S. inequality

$\max\{f(n), g(n)\} \le C_1.(f(n)+g(n) \dots\dots (2)$

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

$\Rightarrow$ 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.

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

b)

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

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:

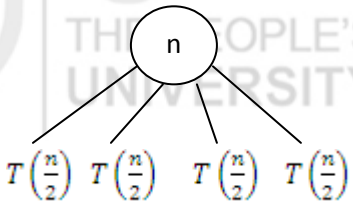


Figure a

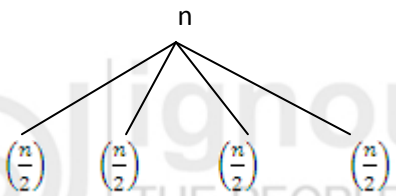


Figure b

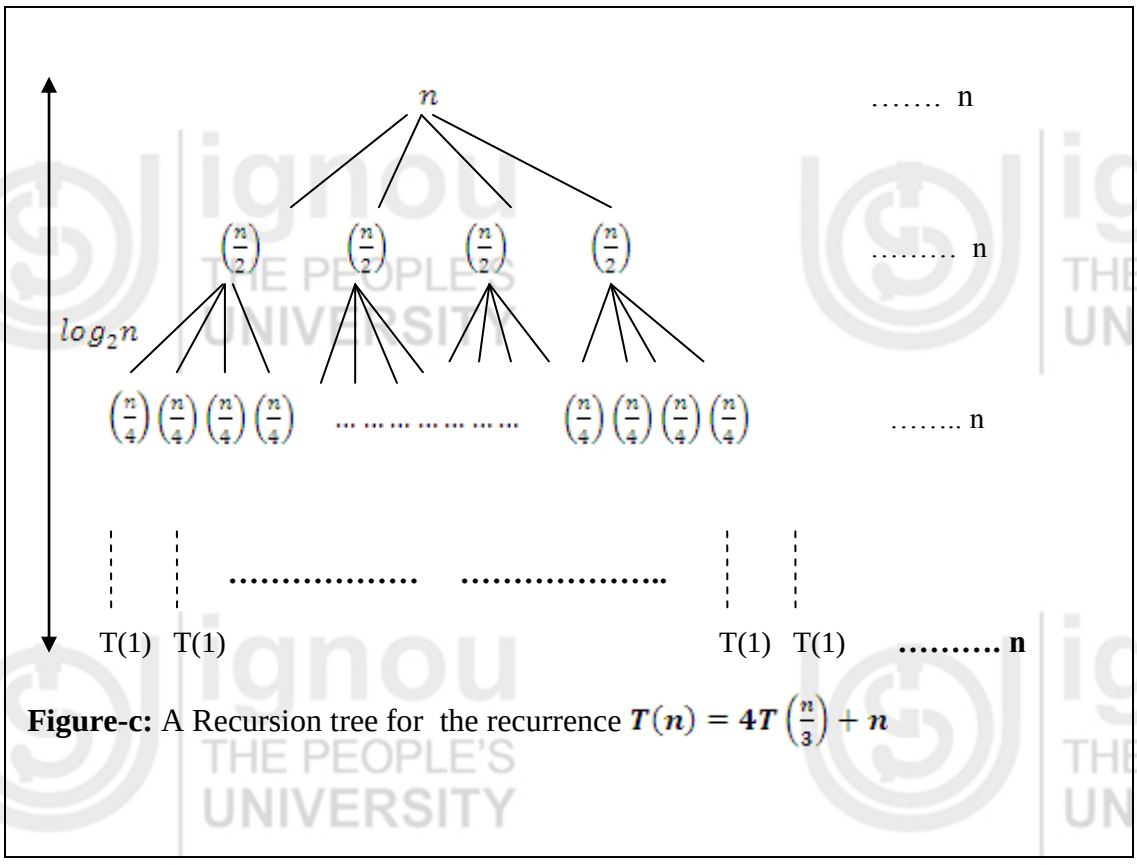


Figure-c: A Recursion tree for the recurrence $\boldsymbol{T(n) = 4T\left(\frac{n}{3}\right) + n}$

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

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

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

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

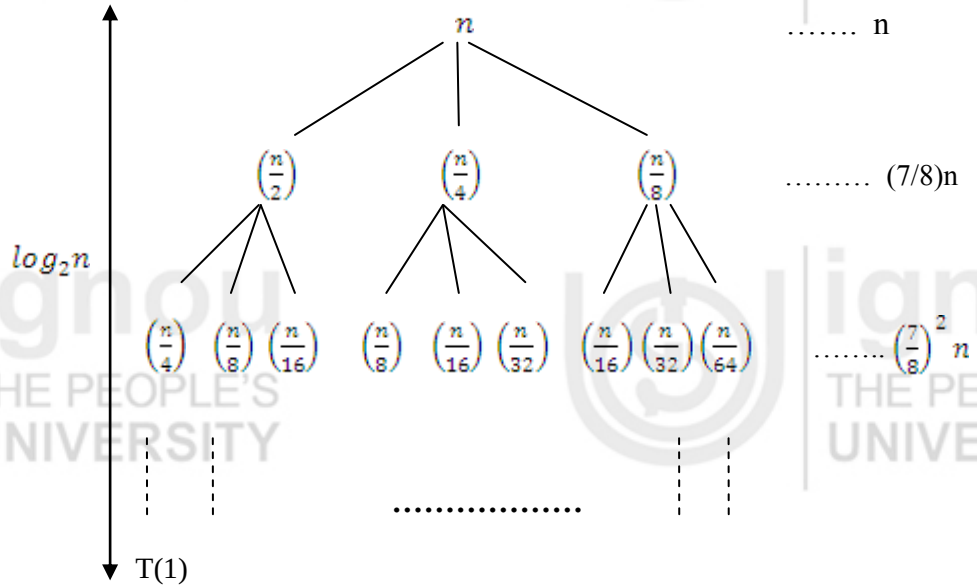
Solution (b) : Refer solution 3(a)

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

Now check (Regularity Condition):

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

$\Longrightarrow \frac{2}{2\sqrt{2}}n^{5/2} \leq c.n^{5/2}$; *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.

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

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^2 log *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).

UNIT 2: ASYMPTOTIC BOUNDS

2.0 INTRODUCTION

In previous unit of the block, we have discussed definition of an algorithm and several characteristics to describe an algorithm. An algorithm provides an approach to solve a given problem. The key components of an algorithm are input, processing and output. Generally all algorithms works well for small size input irrespective of the complexity. So we need to analyze the algorithm for large value of input size. It is also possible that one problem have many algorithmic solutions. To select the best algorithm for an instance of task or input we need to compare the algorithm to find out how long a particular solution will take to generate the desired output. We will determine the behavior of function and running time of an algorithm as a function of input size for large value of n. This behavior can be expressed using asymptotic notations. To understand concepts of the asymptotic notations you will be given an idea of lower bound, upper bound and how to represent time complexity expression for various algorithms. This is like expressing cost component of an algorithm. The basic five asymptotic notations will be discussed here to represent complexity expression in this unit.

In the second section, analysis for efficiency of algorithm is discussed. Efficiency of algorithm is defined in terms of two parameters i.e time and space. Time complexity refers to running time of an algorithm and space complexity refers to the additional space requirement for an algorithm to be executed. Analysis will be focused on running time complexity as response time and computation time is more important as computer speed and memory size has been improved by many orders of magnitude. Time complexity depends on input size of the problem and type of input. Based on the type of data input to an algorithm complexity will be categorized as worst case, average case and best case analysis.

In the last section, linear, quadratic, polynomial and exponential algorithm efficiency will be discussed. It will help to identify that at what rate run time will grow with respect of size of the input

2.1 OBJECTIVES

After studying this unit, you should be able to:

- Asymptotic notations
- Worst case, best case and average case analysis
- Comparative analysis of Constant, Logarithmic, Linear, Quadratic and Exponential growth of an algorithm

2.2 ASYMPTOTIC NOTATIONS

Before staring the discussion of asymptotic notations, let us see the symbols that will be used through out this unit. They are summarized in the following table.

Symbol	Name
θ	Theta
Ω	Big Omega
ε	Belongs to
ω	Small Omega
$\forall$	for all
$\exists$	there exist
=>	Implies

An algorithm is set of instruction that takes some input and after computation it generates an output in finite amount of time. This can be evaluated by a variety of criteria and parameters. For performance analysis of an algorithm, following two complexities measures are considered:

- Space Complexity
- Time Complexity

Space complexity is amount of memory require to run an algorithm. This is sum of fixed part and variable part of a program. Here a fixed part refers to instruction space, constants and variables where as a variable part refers to instance characteristics i.e recursion, run time variables etc. Computer speed and memory size has been improved by many orders of magnitude. Hence for algorithm analysis major focus will be on time complexity.

Time complexity: Total time required to run an algorithm can be expressed as function of input size of problem and this is known as time complexity of algorithm. The limiting behavior of complexity as input size of a problem increases is called asymptotic time complexity. Total time required for completion of solving a problem is equal to sum of compile time and running time. To execute a program, always it is not mandatory that it must be compiled. Hence running time complexity will be under consideration for an evaluation and finding an algorithm complexity analysis.

Before starting with an introduction to asymptotic notations let us define the term asymptote. An asymptote provides a behavior in respect of other function for varying value of input size. An **asymptote** is a line or curve that a graph approaches but does not intersect. An asymptote of a curve is a line in such a way that distance between curve and line approaches zero towards large values or infinity.

The figure 1 will illustrate this.

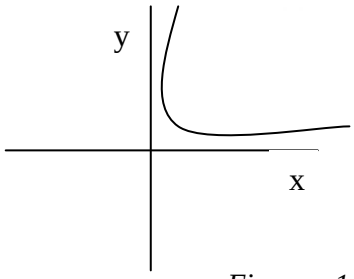


Figure:- 1

In the Figure 1, curve along x-axis and y axis approaches zero. Also the curve will not intersect the x-axis or y axis even for large values of either x or y.
Let us discuss one more example to understand the meaning of asymptote. For example x is asymptotic to x+1 and these two lines in the graph will never intersect as depicted in following Figure 2.

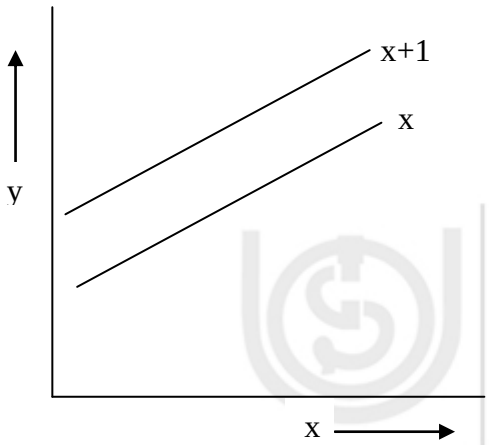


Figure:- 2

In Figure 2, x and x+1 are parallel lines and they will never intersect with each other. Therefore it is called as x is asymptotic to x+1.

The concept of asymptote will help in understanding the behavior of an algorithm for large value of input.
Now we will discuss the introduction to **bounds** that will be useful to understand the asymptotic notations.

Lower Bound: A non empty set A and its subset B is given with relation $\leq$. An element a ε A is called lower bound of B if $a \leq x \; \forall \; x \varepsilon B$ (read as if a is less than equal to x for all x belongs to set B). For example a non empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The lower bound of B= 1, 2 as 1, 2 in the set A is less than or equal to all element of B.

Upper Bound: An element a ε A is called upper bound of B if $x \leq a \; \forall \; x \varepsilon B$. For example a non empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The upper bound of B= 3,4,5,6 as 3,4,5,6 in the set A is greater than or equal to all element of B.

A bound (upper bound or lower bound) is said to be tight bound if the inequality is less than or equal to (≤) as depicted in Figure 3.

Similarly a bound (lower bound or upper bound) is said to be loose bound if the inequality is strictly less than (<) as depicted in Figure 4.

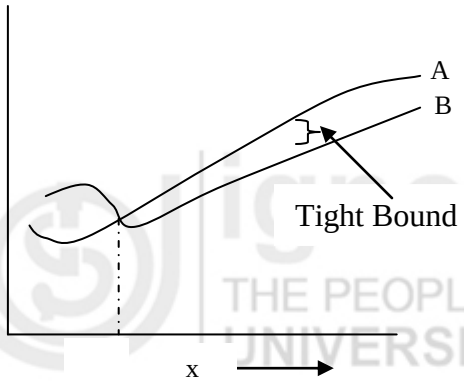


Figure:- 3 Tight Bound

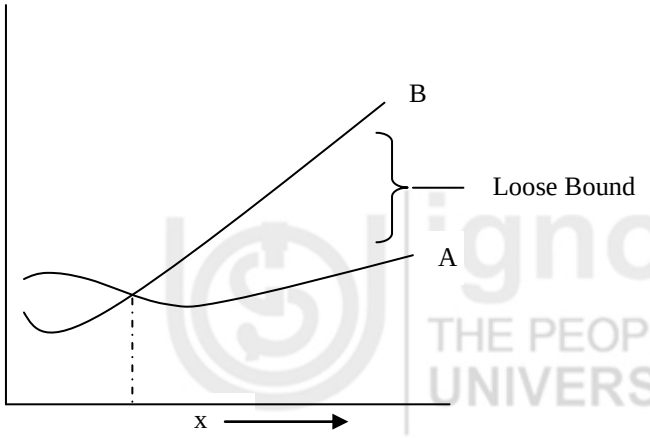


Figure:- 4 Loose Bound

For example in figure 3, distance between lines A and B is less as B ≤ A. For large value of x, B will approach to A as it is less than or equal to A.

In Figure 4, A < B i.e. distance between A and B is large. For example A < B, there will be distance between A and B even for large value of x as it is strictly less than only.

We also require the definition of bounded above or bounded below and bounded above & below both to understand the asymptotic notations.

Bounded above: Let A is non empty set and B is non empty subset of A. Bounded from above on B i.e supremum or least upper bound on B is defined as an upper bound of B which is less than or equal to all upper bounds of B. For example a non empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The upper bound of B= 3,4,5,6 as 3,4,5,6 in the set A is greater than or equal to all element of B. Least upper bound of B is 3 i.e 3 is less than equal to all upper bounds of B.

Bounded below: Let A is non empty set and B is non empty subset of A. Bounded from below on B i.e infimum or greatest lower bound on B is defined as a lower bound of B which is greater than or equal to all lower bounds of B. For example a non empty set A and its subset B is given as A={1,2,3,4,5,6} and B={2,3}. The lower bound of B= 1, 2 as 1, 2 in the set A is less than or equal to all element of B. Greatest lower bound of B is 2 i.e. 2 is greater than equal to all lower bounds of B.

To study the analysis of an algorithm and compute its time complexity we will be computing the total running time of an algorithm. Total running time of an algorithm is dependent on input size of the problem. Hence complexity expression will always be a function in term of input size. Hence we also require understanding the bounds in respect of function.

In respect of function defined on non empty set X , bounded above is written as $f(x) \leq A \ \forall \ x \ \varepsilon \ X$ then we say function is bounded above by A. It is read as function for all elements in the set X is less than or equal to A.

Similarly bounded below is written as $A \leq f(x) \ \forall \ x \ \varepsilon \ X$, it is said to be function is bounded below by A. It is read as function for all elements in the set X is greater than or equal to A.

A function is said to be bounded if it has both bounds i.e bounded above and below both. It is written as $A \leq f(x) \leq B \ \forall \ x \ \varepsilon \ X$.
The bounded above is depicted by figure 5 and bounded below by figure 6. Bounded above and below both is illustrated by figure 7.

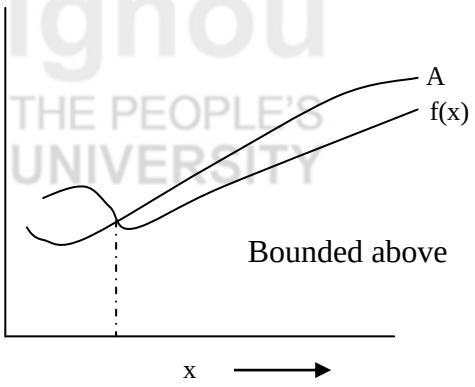


Figure:- 5 Bounded above

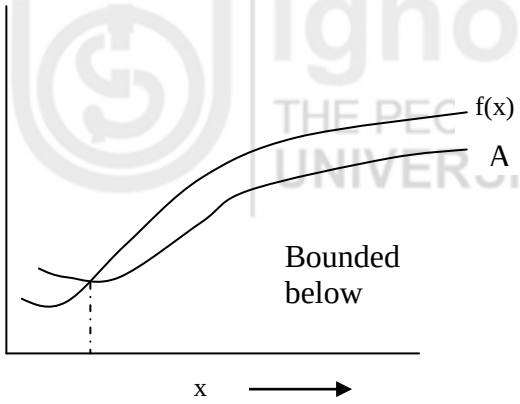


Figure:- 6 Bounded below

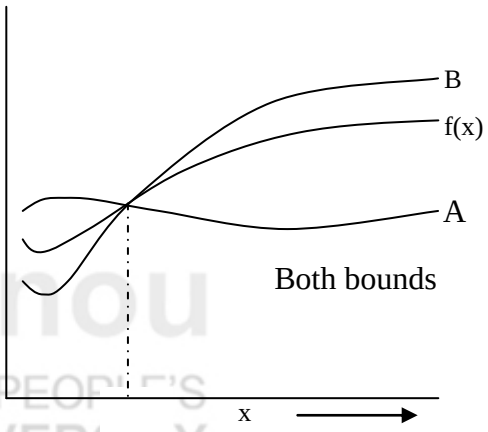


Figure:- 7 Bounded above & below

In figure 5, bounded above indicates that the value of f(x) will never exceed A. It means we know the largest value of function f(x) for any input value for x. Similarly in figure 6 bounded below provide the smallest value of function f(x) for any input value of x. In figure 7 we get the information for smallest and largest value both. The function f(x) will be in the range A and B i.e the smallest value for function f(x) is A and the largest value for f(x) is B. Here we know the both the values A and B i.e minimum and maximum value for f(x) for any input value of x.

Now, Let us discuss the formal definitions of basic asymptotic notation which are named as θ (Theta), O(Big Oh), Ω (Big Omega),o(Small Oh), ω(Small Omega).

Let g(n) be given function i.e a function in terms of input size n. In the following section we will be discussing various asymptotic notations to find the solution represented by function f(n) belongs to which one of basic asymptotic notations.

2.3.1 Theta (θ) Notation

It provides both upper and lower bounds for a given function.

θ (Theta) Notation: means `order exactly'. Order exactly implies a function is bounded above and bounded below both. This notation provides both minimum and maximum value for a function. It further gives that an algorithm will take this much of minimum and maximum time that a function can attain for any input size as illustrated in figure 7.

Let g(n) be given function. f(n) be the set of function defined as
θ (g(n)) = {f(n): if there exist positive constant c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$}

It can be written as f(n)= θ (g(n)) or f(n)ε θ (g(n)), here f(n) is bounded both above and below by some positive constant multiples of g(n) for all large values of n. It is described in the following figure 8.

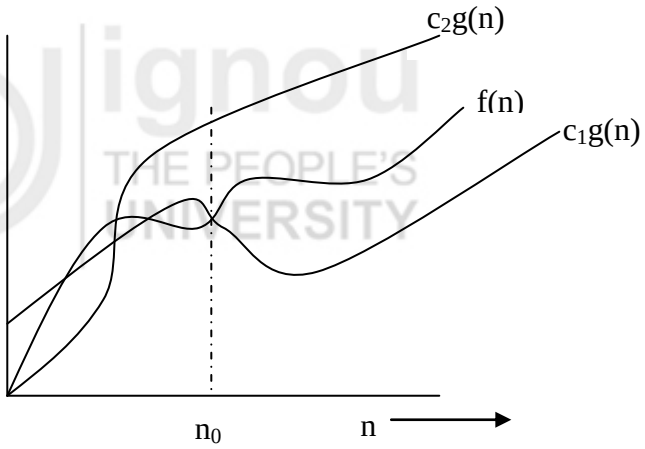


Figure:- 8 θ(n)

In the figure 8 function f(n) is bounded below by constant c1 times g(n) and above by constants c2 times g(n). We can explain this by following examples:

Example 1:

To show that 3n+3 = θ(n) or 3n+3 ε θ(n) we will verify that f(n) ε g(n) or not with the help of the definition i.e

θ (g(n)) = {f(n): if there exist positive constant c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$}

In the given problem f(n)= 3n+3 and g(n)=n to prove f(n) ε g(n) we have to find c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

=> to verify $f(n) \leq c_2 g(n)$

We can write $f(n)=3n+3$ as $f(n)=3n+3 \leq 3n+3n$ (write $f(n)$ in terms of $g(n)$ such that mathematically ineqaulity should be true)

$\leq 6n$ for all $n > 0$

$c_2=6$ for all $n > 0$ i.e $n_0=1$

To verify $0 \leq c_1 g(n) \leq f(n)$

We can write $f(n)=3n+3 \geq 3n$ (again write $f(n)$ in terms of $g(n)$ such that mathematically ineqaulity should be true)

$c_1=3$ for all n, $n_0=1$

$\Rightarrow 3n \leq 3n+3 \leq 6n$ for all $n \geq n_0$, $n_0=1$

i.e we are able to find, $c_1=3$, $c_2=6$ $n_0=1$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
So, $f(n)= \theta(g(n))$ for all $n \geq 1$

Example 2:

To show that $10n^2+4n+2=\theta(n^2)$ or $10n^2+4n+2 \in \theta(n^2)$ we will verify that $f(n) \in g(n)$ or not with the help of the definition i.e

$\theta(g(n)) = \{f(n)$: if there exist positive constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0\}$

In the given problem $f(n)= 10n^2+4n+2$ and $g(n)= n^2$ to prove $f(n) \in g(n)$ we have to find c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

$\Rightarrow$ to verify $f(n) \leq c_2 g(n)$

We can write $f(n)= 10 n^2+4n+2 \leq 10n^2+4n^2+2n^2$ (write $f(n)$ in terms of $g(n)$ such that mathematically ineqaulity should be true)

$\leq 16 n^2$

$c_2=16$ for all n

To verify $0 \leq c_1 g(n) \leq f(n)$ We can write $f(n)= 10 n^2+4n+2 \geq 10n^2$
(write $f(n)$ in terms of $g(n)$ such that mathematically ineqaulity should be true)

$c_1=10$ for all n, $n_0=1$

$\Rightarrow 10n^2 \leq 10 n^2+4n+2 \leq 16 n^2$ for all $n \geq n_0$, $n_0=1$
i.e we are able to find, $c_1=10$, $c_2=16$ $n_0=1$such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
So, $f(n)= \theta(g(n))$ for all $n \geq 1$

2.3.2 Big Oh (O) Notation

This notation provides upper bound for a given function.

O(Big Oh) Notation: mean `order at most' i.e bounded above or it will give maximum time required to run the algorithm.

For a function having only asymptotic upper bound, Big Oh 'O' notation is used.

Let a given function g(n), O(g(n))) is the set of functions f(n) defined as

O(g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$}
f(n)= O(g(n)) or f(n)ε O(g(n)), f(n) is bounded above by some positive constant multiple of g(n) for all large values of n. The definition is illustrated with the help of figure 9.

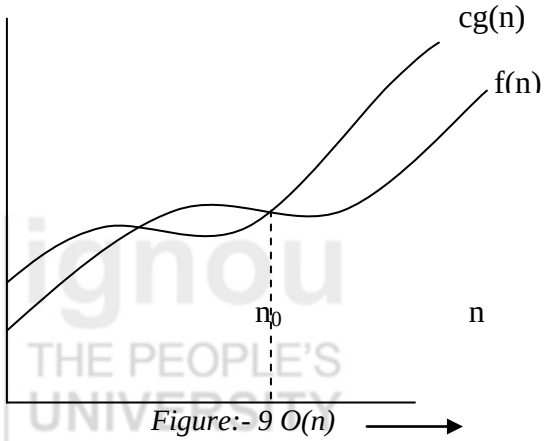


Figure:- 9 O(n)

In this figure9, function f(n) is bounded above by constant c times g(n). We can explain this by following examples:

Example 3:

To show $3n^2+4n+6=O(n^2)$ we will verify that f(n) ε g(n) or not with the help of the definition i.e O(g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$}

In the given problem

$f(n)= 3n^2+4n+6$
$g(n)= n^2$

To show $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$
$f(n)= 3n^2+4n+6 \leq 3n^2+n^2$ for $n \geq 6$
$\leq 4n^2$
c=4 for all $n \geq n_0$, $n_0=6$

i.e we can identify , c=4, $n_0=6$
So, $f(n)=O(n^2)$

Example 4:

To show 5n+8=O(n) we will verify that f(n) ε g(n) or not with the help of the definition i.e O(g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$}

In the given problem

f(n)=5n+8
g(n)=n

To show $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$
$f(n)=5n+8 \leq 5n+8n$

$\leq 13n$

$c=13$ for all $n \geq n_0$, $n_0=1$

i.e we can identify, $c=13$, $n_0=1$

So, $f(n)=O(g(n))$ i.e $f(n)=O(n)$

2.3.3 Big Omega (Ω) Notation

This notation provides lower bound for a given function.

Ω (Big Omega): mean 'order at least' i.e minimum time required to execute the algorithm or have lower bound

For a function having only asymptotic lower bound, Ω notation is used.
Let a given function g(n). Ω (g(n))) is the set of functions f(n) defined as
Ω (g(n)) ={f(n): if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$}

$f(n)= \Omega (g(n))$ or $f(n) \varepsilon \Omega (g(n))$, f(n) is bounded below by some positive constant multiple of g(n) for all large values of n. It is described in the following figure 10.

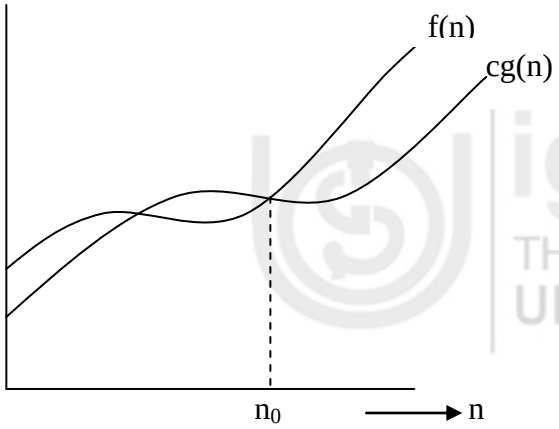


Figure:- 10 Ω (n)

In this figure 10, function f(n) is bounded below by constant c times g(n). We can explain this by following examples:

Example 5:

To show $2n^2+4n+6=\Omega (n^2)$ we will verify that $f(n) \varepsilon g(n)$ or not with the help of the definition i.e Ω (g(n)) ={f(n): if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$}

In the given problem

$f(n)= 2n^2+4n+6$
$g(n)= n^2$

To show $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$

We can write $f(n)= 2n^2+4n+6$

$0 \leq 2n^2 \leq 2n^2+4n+6$ for $n \geq 0$
$c=2$ for all $n \geq n_0$, $n_0=0$
i.e we are able to find, $c=2$, $n_0=0$

So, $f(n)= \Omega (n^2)$

Example 6:

To show $n^3=\Omega (n^2)$ we will verify that $f(n) \varepsilon g(n)$ or not with the help of the definition
i.e $\Omega (g(n)) =\{f(n)$: if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0\}$

In the given problem

$f(n)= n^3$
$g(n)= n^2$

To show $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$

We can write

$f(n)= n^3$
$0 \leq n^2 \leq n^3$ for $n \geq 0$
$c=1$ for all $n \geq n_0$, $n_0=0$

i.e we can select, $c=1$, $n_0=0$

So, $f(n)= \Omega (n^2)$

2.3.4 Small o (o) Notation

o(small o) Notation:

For a function that does not have asymptotic tight upper bound, o (small o) notation is used. i.e. It is used to denote an upper bound that is not asymptotically tight.

Let a given function g(n), o (g(n))) is the set of functions f(n) defined as
o (g(n)) = {f(n): for any positive constant c there exist a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$

f(n)= o(g(n)) or f(n)ε o (g(n)), f(n) is loosely bounded above by all positive constant multiple of g(n) for all large n. It is illustrated in the following figure 11.

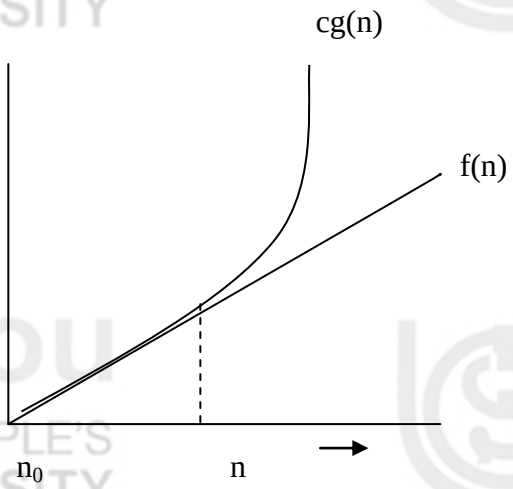


Figure:-11 o(n)

In this figure 11, function f(n) is loosely bounded above by constant c times g(n). We can explain this by following example:

Example 7:

To show $2n+4=o(n^2)$ we will verify that f(n) ε g(n) or not with the help of the definition i.e o (g(n)) = {f(n): for any positive constant c there exist a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$}

In the given problem

$f(n)=2n+4$, $g(n)= n^2$

To show $0 \le f(n) < cg(n)$ for all $n \ge n_0$ We can write as
$f(n)=2n+4 < cn^2$
for any $c > 0$, for all $n \ge n_0$, $n_0=1$

i.e we can find, $c=1$, $n_0=1$

Hence, f(n)=o(g(n))

Example 8:

To show $2n=o(n^2)$ we will verify that f(n) ε g(n) or not with the help of the definition i.e o (g(n)) = {f(n): for any positive constant c there exist a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$}

In the given problem

$f(n)=2n$, $g(n)= n^2$

To show $0 \le f(n) < cg(n)$ for all $n \ge n_0$ We can write as

$f(n)=2n < cn^2$

for any c >0 , for all $n \geq n_0$, $n_0=1$

i.e we can find, c=1, $n_0=1$

Hence, f(n)=o(g(n))

2.3.5 Small Omega (ω)Notation

ω(Small Omega) Notation:

For a function that does not have asymptotic tight lower bound, ω notation is used. i.e. It is used to denote a lower bound that is not asymptotically tight.

Let a given function g(n). ω (g(n))) is the set of functions f(n) defined as
ω (g(n)) = {f(n): for any positive constant c > 0 there exist a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$}

f(n)= ω (g(n)) or f(n)ε ω (g(n)), f(n) is loosely bounded below by all positive constant multiple of g(n) for all large n. It is described in the following figure 11.

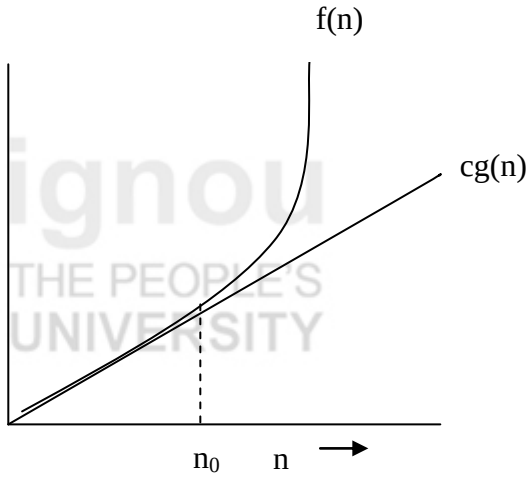


Figure:-11 ω (n))

In this figure function f(n) is loosely bounded below by constant c times g(n).
Following example illustrate this notation:

Example 9:

To show $2n^2+4n+6= \omega (n)$ we will verify that f(n) ε g(n) or not with the help of the definition i.eω (g(n)) = {f(n): for any positive constant c > 0 there exist a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$}

In the given problem

$f(n)= 2n^2+4n+6$
$g(n)= n$

To show $0 \leq cg(n) < f(n)$ for all $n \geq n_0$We can write as

$f(n)= 2n^2+4n+6$

$cn < 2n^2+4n+6$ for any $c > 0$, for all $n \geq n_0$, $n_0=1$
i.e we can find, $c=1$, $n_0=1$

Hence, $f(n)= \omega (g(n))$ i.e $f(n)= \omega (n)$

Example 10:

To show $2n^3+3n^2+1= \omega (n)$ we will verify that $f(n) \varepsilon g(n)$ or not with the help of the definition i.e$\omega (g(n)) = \{f(n)$: for any positive constant $c > 0$ there exist a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

In the given problem

$f(n)= 2n^3+3n^2+1$
$g(n)= n$

To show $0 \leq cg(n) < f(n)$ for all $n \geq n_0$We can write as

$f(n)= 2n^3+3n^2+1$
$cn < 2n^3+3n^2+1$ for any $c > 0$, for all $n \geq n_0$, $n_0=1$
i.e we can find, $c=1$, $n_0=1$

Hence, $f(n)= \omega (g(n))$ i.e $f(n)= \omega (n)$

Let us summarize the above asymtotic notations in the following table.

Notation Name	**Mathematical inequality**	**Meaning**
θ	+ve constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$	Minimum and maximum time that a function f can take
O	+ve constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$	Maximum time that a function f can take
Ω	+ve constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$	Minimum time that a function f can take
o	$c > 0$ there exist $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$	Function f will take strictly less than Maximum time
ω	$c > 0$ there exist $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$	Function f will take strictly greater than Maximum time

An algorithm complexity can be written in the form of asymptotic notations discussed in this section depending upon algorithm will fall under which notation. For example let us take a part of any algorithm where we read n element of an array.

```
1.    scanf("%d",&n);
2.    printf("Enter element for an array");
3.    for(i=0;i<n;i++)
4.       scanf("%d",&a[i]);
```

Line Number	Number of times
1	1
2	1
3	n
4	n-1

f(n)= 1+1+n+(n-1)
f(n)=2n+1

Now to compute the complexity for the above construct of an algorithm, let us find the bounds for above function f(n) i.e 2n+1.

Let us verify whether f(n) is O(n) , Ω (n) and θ (n).

To show f(n)= O(n)
f(n)=2n+1
g(n)=n
f(n)=2n+1≤2n+n for all n ≥ 1
≤3n
c=3 for all $n \geq n_0$, $n_0=1$
i.e we can identify, c=3, $n_0=1$

So, f(n)=O(g(n)) i.e f(n)=O(n)

To show f(n)= Ω (n)

f(n)= 2n+1
g(n)= n
f(n)= 2n+1
$0 \leq n \leq 2n+1$ for $n \geq 0$
c=1 for all $n \geq n_0$, $n_0=0$

i.e we can select, c=1, $n_0=0$

So, f(n)= Ω (n)

To show f(n)= θ (n)

f(n)= 2n+1 and g(n)=n
=> f(n)=2n+1 ≤ 2n+n for all n>=1
≤3n
c_2=3 for all n

Also f(n)=2n+1 ≥n for all n>=1
c_1=1 for all n

i.e we are able to find, $c_1=1$, $c_2=3$ $n_0=1$

So, f(n)= θ(g(n)) i.e f(n) = θ(n) for all n ≥1

For this construct complexity will be f(n)= O(n), f(n)=Ω (n), f(n)=θ (n).

However, we will generally be most interested in the Big Oh time analysis as this analysis can lead to computation of maximum time required for the algorithm to solve the given problem.

In the next section, we will discuss about concept of efficiency analysis of an algorithm.

☞ Check Your Progress 1

1. Define the following:
   a) Algorithm

   ……

   ……

   ……

   b) Time Complexity

   ……

   ……

   ……

   c) Space Complexity

   ……

   ……

   ……

2. Define basic five asymptotic notations.

   ……

   ……

   ……

   ……

   ……

3. Give an example for each asymptotic notations as defined in Q2

   ……

   ……

   ……

   ……

2.3 CONCEPT OF EFFICIENCY ANALYSIS OF ALGORITHM

If we are given an input to an algorithm we can exactly compute the number of steps our algorithm executes. We can also find the count of the processor instructions. Usually, we are interested in identifying the behavior of our program w.r.t input supplied to the algorithm. Based on type of input, analysis can be classified as following:

- Worst Case
- Average Case
- Best Case

In the **worst case** - we need to look at the input data and determine an upper bound on how long it will take to run the program. Analyzing the efficiency of an algorithm in the worst case scenario speaks about how fast the maximum runtime grow when we increase the input size. For example if we would like to sort a list of n numbers in ascending order and the list is given in descending order. It will lead to worst case scenario for the sorting algorithm.

In **average case** – we need to look at time required to run the algorithm where all inputs are equally likely. Analyzing the efficiency of an algorithm speaks about probabilistic analysis by which we find expected running time for an algorithm. For example in a list of n numbers to be sorted in ascending order, some numbers may be at their required position and some may be not in order.

In **Best case**- Input supplied to the algorithm will be almost similar to the format in which output is expected. And we need to compute the running time of an algorithm. This analysis will be referred as best case analysis. For example we would like to sort the list of n numbers in ascending order and the list is already in ascending order.

During efficiency analysis of algorithm, we are required to study the behavior of algorithm with varying input size. For doing the same, it is not always required to execute on a machine, number of steps can be computed by simulating or performing dry run on an algorithm.

For example: Consider the linear search algorithm in which we are required to search an element from a given list of elements, let's say size of the list is n.

Input: An array of n numbers and an element which is required to be searched in the given list

Output: Number exists in the list or not.

Algorithm:

1. Input the size of list i.e. n
2. Read the n elements of array A
3. Input the item/element to be searched in the given list.
4. for each element in the array i=1 to n
5. if A[i]==item
6. Search successful, return
7. if i==n+1
8. Search unsuccessful.
9. Stop

Efficiency analysis of the above algorithm in respect of various cases is as follows:

Worst Case: In respect of example under consideration, the worst case is when the element to be searched is either not in the list or found at the end of the list. In this case algorithm runs for longest possible time i.e maximum running time of the algorithm depends on the size of an array so, running time complexity for this case will be O(n).

Average case: In this case expected running time will be computed based on the assumption that probability of occurrence of all possible input is equal i.e array elements could be in any order. This provides average amount of time required to solve a problem of size n. In respect of example under consideration, element could be found at random position in the list. Running time complexity will be O(n).

Best Case: In this the running time will be fastest for given array elements of size n i.e. it gives minimum running time for an algorithm. In respect of example under consideration, element to be searched is found at first position in the list. Running time complexity for this case will be O(1).

In most of the cases, average case analysis and worst case analysis plays an important role in comparison to best case. Worst case analysis defines an upper bound on running time for any input and average case analysis defines expected running time for input of given size that are equally likely.

For solving a problem we have more than one solution. Comparison among different solutions provides which solution is much better than the other i.e which one is more efficient to solve a problem. Efficiency of algorithm depends on time taken to run the algorithm and use of memory space by the algorithm. As already discussed, focus will be on time efficiency rather than space.

Execution time of an algorithm will be computed on different sizes of n. For large values of n, constant factor will not affect the complexity of an algorithm. Hence it can be expressed as a function of size n.

For example O(n)= O(n/2)= O(n+2) etc. It is read as order will be defined in terms of n irrespective of the constant factor like divide by 2 or plus 2 etc. As while discussing complexity analysis we are interested in order of algorithm complexity.

Some algorithm solution could be quadratic function of n. Other solution may be linear or exponential function of n. Different algorithm will fall under different complexity classes.

Behavior of quadratic, linear and exponential in respect of n will be discussed in next section.

☞ Check Your Progress 2

1. Define Best case Time Complexity.

   ……

   ……

   ……

2. Define Worst case Time Complexity.

   ……

   ……

   ……

3. Define Average case time complexity.

   ……

   ……

   ……

4. Write an algorithm for bubble sort and write its worst case, average case and best case analysis.

   ……

   ……

2.4 COMPARASION OF EFFICIENCIES OF AN ALGORITHM

Running time for most of the algorithms falls under different efficiency classes.

a) 1 Constant Time When instructions of program are executed once or at most only a few times , then the running time complexity of such algorithm is know as constant time. it is independent of the problem's size. It is represented as O(1). For example, linear search best case complexity is O(1)

b) log n Logarithmic The running time of the algorithm in which large problem is solved by transforming into smaller sizes sub problems is said to be Logarithmic in nature. In this algorithm becomes slightly slower as n grows. It does not process all the data element of input size n. The running time does not double until n increases to n^2. It is represented as O(log n). For example binary search algorithm running time complexity is O(log n).

c) n Linear In this the complete set of instruction is executed once for each input i.e input of size n is processed. It is represented as O(n). This is the best option to be used when the whole input has to be processed. In this situation time requirement increases directly with the size of the problem. For example linear search Worst case complexity is O(n).

d) n^2 Quadratic Running time of an algorithm is quadratic in nature when it process all pairs of data items. Such algorithm will have two nested loops. For input size n, running time will be $O(n^2)$. Practically this is useful for problem with small input size or elementary sorting problems. In this situation time requirement increases fast with the size of the problem. For example insertion sort running time complexity is $O(n^2)$.

e) 2^n Exponential Running time of an algorithm is exponential in nature if brute force solution is applied to solve a problem. In such algorithm all subset of an n-element set is generated. In this situation time requirement increases very fast with the size of the problem. For input size n, running time complexity expression will be $O(2^n)$.For example Boolean variable equivalence of n variables running time complexity is $O(2^n)$. Another familiar example is Tower of Hanoi problem where running time complexity is $O(2^n)$.

For large values of n or as input size n grows, some basic algorithm running time approximation is depicted in following table. As already discussed, worst case

analysis is more important hence O Big Oh notation is used to indicate the value of function for analysis of algorithm.

n	Constant	Logarithmic	Linear	Quadratic	Exponential
	O(1)	O(log n)	O(n)	$O(n^2)$	$O(2^n)$
1	1	1	1	1	2
2	1	1	2	4	4
4	1	2	4	16	16
8	1	3	8	64	256
10	1	3	10	10^2	10^3
10^2	1	6	10^2	10^4	10^{30}
10^3	1	9	10^3	10^6	10^{301}
10^4	1	13	10^4	10^8	10^{3010}

The running time of an algorithm is most likely to be some constant multiplied by one of above function plus some smaller terms. Smaller terms will be negligible as input size n grows. Comparison given in above table has great significance for analysis of algorithm.

☞ Check Your Progress 3

1. Define basic efficiency classes.

   ……

   ……

   ……

2. Write a function for implementing binary search. Also give an expression for running time complexity in terms of input size n for Worst case, Best case and average case.

   ……

   ……

   ……

   ……

   ……

2.5 SUMMARY

Analysis of algorithms means to find out an algorithm's efficiency with respect to resources: running time and memory space. Time efficiency indicates how fast the algorithm executes; space efficiency deals with additional space required running the algorithm. Algorithm running time will depend on input size. This will be the number of basic operation executed for an algorithm. For the algorithm we can define worst case efficiency, best case efficiency and average case efficiency. Worst case efficiency means the algorithm runs the longest time among all possible inputs of size n. Best case efficiency the algorithm runs the fastest among all possible inputs of size n. Average case efficiency means running time for a typical/random input of size n. For example for sorting a set of element in ascending order , input given in descending order is referred as worst case and input arranged in ascending order referred as best case. Input data of mixed type/random type i.e some elements are in order and some are not in order is referred as average case.

Among all worst case analysis is important as it provides the information about maximum amount of time an algorithm requires for solving a problem of input size n. The efficiency of some algorithm may differ significantly for input of the same size.

In this unit, five basic Asymptotic notations are defined: θ (Theta), O(Big Oh), Ω (Big Omega),o(Small Oh), ω(Small Omega).

These notations are used to identify and compare asymptotic order of growth of function in respect of input size n to express algorithm efficiency.

For visualization of growth of function with respect to input size, comparison among values of some functions for analysis of algorithm is provided. In this comparison input size is taken as 2^0,2^1,2^2,2^3,10^1,10^2,10^3, and 10^4 for constant, logarithmic, linear, quadratic and exponential functions.

These notation and comparison for growth of function defined and used here will be used throughout the design and analysis of an algorithm.

2.6 MODEL ANSWERS

Check Your Progress 1:

Answers:

1)
(a) Algorithm: An algorithm is set of instructions to be executed for a given input to solve a problem or generate the required out put in finite amount of time. The algorithm should solve the problem correctly.

(b) Time complexity: Time complexity of an algorithm tells the amount of time required to run the algorithm as a function of input size n. Generally it is expressed using O Big Oh notation which ignores constant and smaller terms.

(c) Space complexity: Space complexity of an algorithm speaks about additional space required to run the algorithm. Good algorithm will keep this amount of additional memory used as small as possible.

2). Asymptotic Notation: It is the formal way to speak about function and classify them. Basic five notations are:

θ (Theta), O(Big Oh), Ω (Big Omega),o(Small Oh), ω(Small Omega).

θ (Theta): Let g(n) be given function. f(n) is the set of function defined as θ (g(n))={f(n): if there exist positive constant c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$}

O(Big Oh): For a given function g(n), O(g(n))), f(n) is the set of functions defined as O(g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, $n \geq n_0$}

Ω (Big Omega): For a given function g(n), Ω (g(n))), f(n) is the set of functions defined as Ω (g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0$}

o(Small Oh): For a given function g(n), o (g(n))), f(n) is the set of functions defined as o (g(n)) = {f(n): for any positive constant c there exist a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$}

ω(Small Omega): For a given function g(n), ω (g(n))), f(n) is the set of functions defined as ω (g(n)) = {f(n): for any positive constant $c > 0$ there exist a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0$}

3) Example for above define basic asymptotic notation:

θ (Theta): $10n^3+5n^2+17 \varepsilon \theta(n^3)$

$10\ n^3 \le 10n^3+5n^2+17 \le (10+5+17)\ n^3$
$= 32\ n^3$
$c_1=10, c_2=32, n_0=1$
$10\ n^3 \le 10n^3+5n^2+17 \le 32\ n^3$ for all $n \ge n_0=1$

O(Big Oh): $10n^3+5n+17 \varepsilon O(n^2)$
$10n^3+5n+17 \le (10+5+17)\ n^3$ for all $n \ge n_0=1$
$= 32\ n^3$
$c=32, n_0=1$
$10n^3+5n+17 \le 32\ n^3$ for all $n \ge n_0=1$

Ω (Big Omega):$2n^3 + 37 \varepsilon \Omega(n^3)$
$2n^3 \le 2n^3 + 37$ for all $n \ge n_0=1$
$c=2, n_0=1$
$2n^3 \le 2n^3 + 37$ for all $n \ge n_0=1$

o(Small Oh): $3n^2 \varepsilon o(n^3)$
$3n^2 < n^3$ for all $n \ge n_0=4$
$c=1, n_0=4$
$3n^2 < n^3$ for all $n \ge n_0=4$

ω(Small Omega): $3n^3 \varepsilon \omega(n^2)$
$cn^2 < 3n^3$ for all $n \ge n_0=1$
$c=1, n_0=1$
$n^2 < 3n^3$ for all $n \ge n_0=1$

4) The algorithm for bubble sort is as below:

```
// a is the list or an array of n elements to be sorted
function bubblesort(a,n)
{
        int i,j,temp,flag=true;
        for(i=0; i<n-1 && flag==true; i++)
        {
                flag=false
                for(j=0; j<n-i-1; j++)
                {
                        if(a[j]>a[j+1])
                        {
                                flag=true
```

```
                        temp = a[j];
                        a[j] = a[j+1];
                        a[j+1] = temp;
                    }
                }

            }
}
```

Complexity analysis of bubble sort is as follows.

Best-case:

When the given data set in an array is already sorted in ascending order the number of moves/exchanges will be 0, then it will be clear that the array is already in order because no two elements need to be swapped. In that case, the sort should end, which takes O(1). The total number of key comparisons will be (n-1) so complexity in best case will be O(n).

Worst-case:

In this case the given data set will be in descending order that need to be sorted in ascending order. Outer loop in the algorithm will be executed n-1 times. The number of exchanges will be $3*(1+2+...+n-1) = 3 * n*(n-1)/2$ i.e $O(n^2)$. The number of key comparison will be $(1+2+...+n-1)= n*(n-1)/2$ i.e $O(n^2)$. Hence complexity in worst case will be $O(n^2)$.

Average –case:

In this case we have to consider all possible initial data arrangement. So as in case of worst case ,outer loop will be executed n-1 times. The number of exchanges will be $O(n^2)$. The number of key comparison will be i.e $O(n^2)$.So the complexity will be $O(n^2)$.

Check Your Progress 3:

1) Basic efficiency classes is depicted in following table:

Running time	Function class
1	constant
log n	logarithmic
n	linear
n^2	quadratic
2^n	exponential

Fast and high time efficiency

Slow and low time efficiency

2) Function for binary search is given below:

```
int binarysearch(int a[], int size, int element)
{
    int beg =0;
    int end = size –1;
    int mid;            // mid will be the index of  target when it's found.
    while (beg <= end)
    {
            mid = (beg + end)/2;
            if (a[mid] < element)
                    beg = mid + 1;
            else if (a[mid] > element)
                    end  = mid – 1;
            else
                    return mid;
    }
    return –1;

}
```

For unsuccessful search running time complexity will be O(log n).

For Successful search that is element to be searched is found in the list, running time complexity for different cases will be as follows:

Worst Case- O(log n)
Best Case – O(1)
Average Case - O(log n)

2.7 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, "Introduction to Algorithms", 2 nd Ed., PHI, 2004.

2. Robert Sedgewick, "Algorithms in C", 3rd Edition, Pearson Education, 2004

3. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajasekaran, "Fundamentals of Computer algorithms", 2nd Edition, Universities Press, 2008

4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education, 2003.

UNIT 3: ANALYSIS OF SIMPLE ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

In the previous unit we have studied about asymptotic notation and efficiency analysis of algorithm. In the continuation this unit will provide an insight to various categories of algorithm and their complexity analysis. Algorithms considered for analyzing the complexity are Euclid's algorithm to compute GCD, Matrix Multiplication of square matrix. Variants of exponent evaluation algorithm and its brute force approach and the major difference in the order of complexity is discussed. Further searching algorithm is described and analyzed for various cases like best case, worst case and average case as described in the Unit2. Then sorting algorithm is categorized on the basis of storage of input data in primary memory or secondary memory. In this unit we have discussed only few internal sorting algorithms and their complexity analysis.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- Algorithm to compute GCD and its analysis
- An algorithm to evaluate polynomial by Horner's rule
- Analysis of Matrix Multiplication algorithm
- Exponent evaluation in logarithmic complexity
- Linear search and its complexity analysis
- Basic sorting algorithm and their analyis

3.2 EUCLID ALGORITHM FOR GCD

Let us take some basic algorithm construct that will be helpful through out this unit for computing execution time of any algorithm. To compute the same we will count the number of basic operation and its cost to find total cost of an algorithm.
For example: Sequence of statement

	Cost	Time
x=x+2;	c1	1
a=b;	c2	1

Total cost for sequence statement will be = 1 x c1 + 1 x c2
= c1+c2 i.e proportional to constant 1

For example : Looping construct

	Cost	time
a=b;	c1	1
for(i=0;i<n;i++)	c2	n+1
x= x+2	c3	n

Total cost for above looping construct will be = 1 x c1 + c2 (n+1) + c3 x n
= (c1+c2)+ n(c2+c3) i.e proportional to n

For example : Nested Looping construct

	Cost	time
a=b;	c1	1
z=2;	c2	1
for(i=0;i<n;i++)	c3	n+1
{		
x= x+2	c4	n
for(j=0;j<n;j++)	c5	n x (n+1)
y=y+1	c6	n x n
}		

Total cost for above looping construct will be =
c1+c2+c3 x (n+1) + c4 x n + c5 x n x (n+1) + c6 x n x n
i.e proportional to n^2

The algorithm for calculating GCD will be explained in two steps. In the first step we will write pseudo code and in the second step the algorithms will be discussed. This algorithm can be easily coded into a programming language. Further explanation of the algorithm is supported through an example.

Let us define GCD (Greatest Common divisor) Problem that you might have already read earlier or referred somewhere during your school days.

GCD of two non negative, non zero (both) integers i.e. m and n, is the largest integer that divides both m and n with a remainder of zero. Complexity analysis of an algorithm for computing GCD depends on which algorithm will be used for GCD computation. In this section Euclid's Algorithm is used to find GCD of two non negative, both non zero integers, m and n.

Step I : Pseudo code for Computing GCD(m,n) by Euclid's Method
// m and n are two positive numbers where m is dividend and n is divisor
1. If n=0 , return m and exit else proceed to step 2.
2. Divide m by n and assign remainder to r.
3. Assign the value of n to m and value of r to n. Go back to step 1.

Step II : Algorithm for Computing GCD(m,n) by Euclid's Method

Input: Two non negative, non zero integers m and n

Output : GCD of m and n

```
function gcd(m,n)
{
    while (n≠0)
    {
            r= m mod n
            m=n
            n=r
    }

    return m
}
```

Example: Find the GCD of 662 and 414

Let m= 662 and n=414
Divide m by n to obtain quotient and remainder.

662=414 · 1+248---------(1) // here 1 is quotient and 248 is remainder

In subsequent iterations dividend and divisor are based on what number we get as a divisor and as a remainder respectively of previous iteration.

So, subsequent iterations are as follows:

414=248 · 1+166 ----------(2) // now m is 414 and n is 166
248=166 · 1+82 ----------(3) // now m is 248 and n is 82
166=82 · 2 + 2 ----------(4) // now m is 166 and n is 2
82= 2 · 41 + 0 ----------(5) // now m is 82 and n is 0

According to Euclid's algorithm, step (1) gcd(662,414)=gcd(414,248)
step(2) gcd(414,248)=gcd(248,166)
step(3) gcd(248,166)=gcd(166,82)
step (4) gcd(166,82)=gcd(82,2)
step (5) gcd (82,2)= gcd (2,0)

Combining above all gives gcd(662,414)=2 which is the last divisor that gives remainder 0.

Complexity Analysis

In function gcd(m,n), each iteration of while loop has one test condition, one division and two assignment that will take constant time. Hence number of times while loop will execute will determine the complexity of the algorithm. Here it can be observed that in subsequent steps, remainder in each step is smaller than its divisor i.e smaller than previous divisor. The division process will definitely terminate after certain number of steps or until remainder becomes 0.

Best Case

If m=n then there will be only one iteration and it will take constant time i.e O(1)

Worst Case

If n=1 then there will be m iterations and complexity will be O(m)
If m=1 then there will be n iterations and complexity will be O(n)

Average Case

By Euclid's algorithm it is observed that

GCD(m,n) = GCD(n,m mod n) = GCD(m mod n, n mod (m mod n))

Since m mod n = r such that m = nq + r, it follows that r < n, so m > 2r. So after every two iterations, the larger number is reduced by at least a factor of 2 so there are at most O(log n) iterations.

Complexity will be O(log n), where n is either the larger or the smaller number.

☞ **Check Your Progress 1**

1. Find GCD(595,252) by Euclid's algorithm.

   ……

   ……

   ……

2. Why Euclid's algorithm's to compute GCD stops at remained 0?

   ……

   ……

   ……

3.3 HORNER'S RULE FOR POLYNOMIAL EVALUATION

The algorithm for evaluating a polynomial at a given point using Horner's rule will be explained in two steps. In the first step we will write pseudo code and in the second step the algorithm will be discussed. This algorithm can be easily coded into any programming language. Further explanation of the algorithm is supported through an example.

In this section we will discuss the problem of evaluating a polynomial using Horner's rule. This problem is also a familiar problem. Before discussing the algorithm , let us define the problem of polynomial evaluation. Consider the polynomial

$p(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots..+a_1x^1+a_0x^0$

where $a_0,a_1,\ldots..a_{n-1},a_n$ are real numbers and we have to evaluate polynomial at a specific value for x.

//For Example: $p(x)=x^2+3x+2$

At x=2, $p(x) = 2^2+3.2+2 = 4+6+2=12$//

Now, we will discuss Horner's rule method for polynomial evaluation.

Consider a polynomial $p(x) = ax^2 + bx + c$ which can be written as x(x(a)+b)+c by using Horner's simplification.

Now cconsidering a general polynomial p(x), $p(x) = a_n x^n + a_{(n-1)}x^{(n-1)} + \ldots + a_1 x^1 + a_0 x^0$

Which can be rewritten as $p(x) = ((a_n + a_{(n-1)})x + a_{(n-2)})x + \ldots + a_1)x + a_0$ by using Horner's simplification.

Step I. Pseudo code for polynomial evaluation using Horner method, Horner(a,n,x)

//In this a is an array of n elements which are coefficient of polynomial of degree n

1. Assign value of polynomial p= coefficient of nth term in the polynomial
2. set i= n-1
4. compute p = p * x + a[i];
5. i=i-1
6. if i is greater than or equal to 0 Go to step 4.
7. final polynomial value at x is p.

Step II. Algorithm to evaluate polynomial at a given point x using Horner's rule:

Input: An array a[0..n] of coefficient of a polynomial of degree n and a point x

Output: The value of polynomial at given point x

```
Evaluate_Horner(a,n,x)
{
p  = a[n];
for (i = n-1; i≥0;i--)
        p =  p * x + a[i];
return p;
}
```

For Example: $p(x)=x^2+3x+2$ using Horner 's rule can be simplified as follows

At x=2,

p(x) = (x+3)x+2

p(2)=(2+3).2+2

= (5).2+2

=10+2

=12

Complexity Analysis

Polynomial of degree n using horner's rule is evaluated as below:

Initial Assignment, $p = a_n$

after iteration 1, $p = x\, a_n + a_{n-1}$

after iteration 2, $p = x(x\, a_n + a_{n-1}) + a_{n-2}$

$= x^2 a_n + x\, a_{n-1} + a_{n-2}$

Every subsequent iteration uses the result of previous iteration i.e next iteration multiplies the previous value of p then adds the next coefficient, i.e.

$p = x(x^2 a_n + x\, a_{n-1} + a_{n-2}) + a_{n-2}$

$= x^3 a_n + x^2 a_{n-1} + x a_{n-2} + a_{n-3}$ etc.

Thus, after n iterations, $p = x^n a_n + x^{n-1} a_{n-1} + \ldots + a_0$, which is the required correct value.

In above function
First step is one initial assignment that takes constant time i.e O(1).

For loop in the algorithm runs for n iterations, where each iteration cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence total time complexity of the algorithm will be O(n) for a polynomial of degree n.

☞ **Check Your Progress 2**

1. Evaluate $p(x)= 3x^4+2x^3-5x+7$ at x=2 using Horne's rule. Discuss step wise iterations.

   ……
   ……
   ……

2. Write basic algorithm to evaluate a polynomial and find its complexity. Also compare its complexity with complexity of Horner's algorithm.

   ……

   ……

   ……

3.4 MATRIX (N X N) MULTIPLICATION

Matrix is very important tool in expressing and discussing problems which arise from real life cases. By managing the data in matrix form it will be easy to manipulate and obtain more information. One of the basic operations on matrices is multiplication.

In this section matrix multiplication problem is explained in two steps as we have discussed GCD and Horner's Rule in previous section. In the first step we will brief pseudo code and in the second step the algorithm for the matrix multiplication will be discussed. This algorithm can be easily coded into any programming language. Further explanation of the algorithm is supported through an example.

Let us define problem of matrix multiplication formally , Then we will discuss how to multiply two square matrix of order n x n and find its time complexity. Multiply two matrices A and B of order nxn each and store the result in matrix C of order nxn.

A square matrix of order nxn is an arrangement of set of elements in n rows and n columns.

Let us take an example of a matrix of order 3 x 3 which is represented as

$$A= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}_{3x3}$$

This matrix A has 3 rows and 3 columns.

Step I : Pseudo code: For Matrix multiplication problem where we will multiply two matrices A and B of order 3x3 each and store the result in matrix C of order 3x3.

1. Multiply first row first element of first matrix with first column first element of second matrix.
2. Similarly perform this multiplication for first row of first matrix and first column of second matrix. Now take the sum of these values.
3. The sum obtained will be first element of product matrix C
4. Similarly Compute all remaining element of product matix C.

i.e $c_{11}= a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31}$

C= A x B

Step II : Algorithm for multiplying two square matrix of order n x n and find the product matrix of order n x n

Input: Two n x n matrices A and B
Output: One n x n matrix C = A x B

```
Matrix_Multiply(A,B,C,n)
{

for i = 0 to n-1  //outermost loop
      for j = 0 to n-1
      {
      C[i][j]=0                  //assignment statement
                  for k = 0 to n-1  // innermost loop
                  C[i][j] = C[i][j] + A[i][k] * B[k][j]
      }
}
```

For Example matrix A (3 x 3) , B(3 x3)

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 2 \\ 3 & 2 & 1 \end{pmatrix}$$

To compute product matrix C= A x B

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} 1x1+2x2+3x3 & 1x1+2x3+3x2 & 1x1+2x2+3x1 \\ 2x1+3x2+4x3 & 2x1+3x3+4x2 & 2x1+3x2+4x1 \\ 4x1+5x2+6x3 & 4x1+5x3+6x2 & 4x1+5x2+6x1 \end{pmatrix}$$

$$= \begin{pmatrix} 14 & 13 & 8 \\ 20 & 19 & 12 \\ 32 & 31 & 20 \end{pmatrix}$$

Complexity Analysis

First step is, for loop that will be executed n number of times i.e it will take O(n) time. The second nested for loop will also run for n number of time and will take O(n) time.

Assignment statement inside second for loop will take constant time i.e O(1) as it includes only one assignment.

The third for loop i.e innermost nested loop will also run for n number of times and will take O(n) time . Assignment statement inside third for loop will cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence, total time complexity of the algorithm will be $O(n^3)$ for matrix multiplication of order nxn.

☞ **Check Your Progress 3**

1. Write a program in 'C' to find multiplication of two matrices A[3x3] and B[3 x3].

……

……

……

3.5 EXPONENT EVALUATION

In section 3.4 of this unit we have discussed Horner's rule to evaluate polynomial and its complexity analysis. But computing x^n at some point x = a i.e a^n tends to brute force multiplication of a by itself n times. So computing x^n is most important operation. It has many applications in various fields for example one of well known field is cryptography and encryption methods. In this section we will discuss binary exponentiation methods to compute x^n. In this section first we will discuss pseudo code then we will explain algorithm for computing x^n. In this binary representation of exponent n is used for computation of exponent. Processing of binary string for exponent n to compute x^n can be done by following methods:

- left to right binary exponentiation
- right to left binary exponentiation

3.6.1 Left to right binary exponentiation

In this method exponent n is represented in binary string. This will be processed from left to right for exponent computation x^n at x=a i.e a^n. First we will discuss its pseudo code followed by algorithm.

Step I : Pseudo code to compute a^n by left to right binary exponentiation method
// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set result =a
2. set i=s-2
3. compute result = result * result
4. if A[i] = 1 then compute result = result * a
5. i=i-1 and if i is less than equal to 0 then go to step 4.
6. return computed value as result.

Step II : Algorithm to compute a^n by left to right binary exponentiation method is as follows:

Input: a^n and binary string of length s for exponent n as an array A[s]

Output: Final value of a^n .

1. result =a
2. for i=s-2 to 0
3. result = result * result
4. if A[i]= 1 then
5. result= result * a
6. return result (i.e a^n)

Let us take an example to illustrate the above algorithm to compute a^{17}
In this exponent n=17 which is equivalent to binary string 10001
Step by step illustration of the left to right binary exponentiation algorithm for a^{17} :
s=5
result=a

Iteration 1:

i=3
result=a *a= a^2
A[3] $\neq$ 1

Iteration 2:

i=2
result= $a^2 * a^2 = a^4$
A[2] $\neq$ 1

Iteration 3

i=1
result= $a^4 * a^4 = a^8$
A[1] $\neq$ 1

Iteration 4

i=0
result= $a^8 * a^8 = a^{16}$
A[0] = 1
result = $a^{16} * a = a^{17}$

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplications in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications in each iteration of a for loop in line no. 2 of the algorithm.

Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So

complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

3.6.2 Right to left binary exponentiation

In right to left binary exponentiation to compute a^n , processing of bits will start from least significant bit to most significant bit.

Step I : Pseudo code to compute a^n by right to left binary exponentiation method

// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set x =a
2. if A[0]= 1 then set result =a
3. else set result =1
4. Initialize i=1
5. compute x = x * x
6. if A[i] = 1 then compute result = result * x
7. Increment i by 1 as i=i+1 and if i is less than equal to s-1 then go to step 4.
8. return computed value as result.

Step II : Algorithm to compute a^n by right to left binary exponentiation method algorithm is as follows:

Input: a^n and binary string of length s for exponent n as an array A[s]

Output: Final value of a^n.

1. x=a
2. if A[0]=1 then
3. result = a
4. else
5. result=1
6. for i= 1 to s-1
7. x= x * x
8. if A[i]=1
9. result= result * x
10. return result (i.e a^n)

Let us take an example to illustrate the above algorithm to compute a^{17}
In this exponent n=17 which is equivalent to binary string 10001

Step by step illustration of the right to left binary exponentiation algorithm for a^{17} : s=5, the length of binary string of 1's and 0's for exponent n

Since A[0] =1 , result=a

Iteration 1:

i=1
x=a *a= a^2
A[1]≠ 1

Iteration 2:

i=2
x= $a^2 * a^2 = a^4$

$A[2] \neq 1$

Iteration 3

i=3
$x = a^4 * a^4 = a^8$
$A[3] \neq 1$

Iteration 4

i=4
$x = a^8 * a^8 = a^{16}$
$A[4] = 1$
$result = result * x = a * a^{16} = a^{17}$

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplication in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications in each iteration of for loop as shown in line no. 6.
Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

From the above discussion we can conclude that the complexity for left to right binary exponentiation and right to left binary exponentiation is logarithmic in terms of exponent n.

☞ **Check Your Progress 4**

1. Compute a^{283} using left to right and right to left binary exponentiation.

   ……

   ……

   ……

3.6 SEARCHING

Computer system is generally used to store large amount of data. For accessing a data item from a large data set based on some criteria/condition searching algorithms are required. Many algorithms are available for searching a data item from large data set stored in a computer viz. linear search, binary search. In this section we will discuss the performance of linear search algorithm. Binary search will be discussed in the Block-2. In the next section we will examine how long the linear search algorithm will take to find a data item/key in the data set.

3.7.1 Linear Search

Linear searching is the algorithmic process of finding a particular data item/key element in a large collection of data items. Generally search process will return the value as true/false whether the searching item/element is found/present in data set or not?

We are given with a list of items. The following table shows a data set for linear search:

7	17	3	9	25	18

In the above table of data set, start at the first item/element in the list and compared with the key. If the key is not at the first position, then we move from the current item to next item in the list sequentially until we either find what we are looking for or run out of items i.e the whole list of items is exhausted. If we run out of items or the list is exhausted, we can conclude that the item we were searching from the list is not present.

The key to be searched=25 from the given data set

In the given data set key 25 is compared with first element i.e 7 , they are not equal then move to next element in the list and key is again compared with 17 , key 25 is not equal to 17. Like this key is compared with element in the list till either element is found in the list or not found till end of the list. In this case key element is found in the list and search is successful.

Let us write the algorithm for the linear search process first and then analyze its complexity.

```
// a is the list of n elements, key is an element to be searched in the list
function linear_search(a,n,key)
{
        found=false // found is a boolean variable which will store either true or false
        for(i=0;i<n;i++)
        {
                if (a[i]==key)
                        found = true
                        break;
        }
        if (i==n)
                found = false
        return found
}
```

For the complexity analysis of this algorithm, we will discuss the following cases:

a. best case time analysis
b. worst-case time analysis
c. average case time analysis

To analyze searching algorithms, we need to decide on a basic unit of computation. This is the common step that must be repeated in order to solve the problem. For

searching, comparison operation is the key operation in the algorithm so it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. If the item is not in the list, the only way to know it is to compare it against every item present.

Best Case:

The best case - we will find the key in the first place we look, at the beginning of the list i.e the first comparison returns a match or return found as true. In this case we only require a single comparison and complexity will be O(1).

Worst Case:

In worst case either we will find the key at the end of the list or we may not find the key until the very last comparison i.e *nth* comparison. Since the search requires n comparisons in the worst case, complexity will be O(n).

Average Case:

On average, we will find the key about halfway into the list; that is, we will compare against *n*/2 data items. However, that as *n* gets larger, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the linear search, is *O*(*n*). The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility.

If the performance of the system is crucial, i.e. it's part of a life-critical system, and then we must use the worst case in our design calculations and complexity analysis as it tends to the best guaranteed performance.

The following table summarizes the above discussed results.

Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2) = O(n)
item is not present	O(n)	O(n)	O(n)

However, we will generally be most interested in the worst-case time calculations as worst-case times can lead to guaranteed performance predictions.
Most of the times an algorithm run for the longest period of time as defined in worst case. Information provide by best case is not very useful. In average case, it is difficult to determine probability of occurrence of input data set. Worst case provides an upper bound on performance i.e the algorithm will never take more time than computed in worse case. So, the worst-case time analysis is easier to compute and is useful than average time case.

3.7 SORTING

Sorting is the process of arranging a collection of data into either ascending or descending order. Generally the output is arranged in sorted order so that it can be easily interpreted. Sometimes sorting at the initial stages increases the performances of an algorithm while solving a problem.

Sorting techniques are broadly classified into two categories:

- Internal Sort: - Internal sorts are the sorting algorithms in which the complete data set to be sorted is available in the computer's main memory.

- External Sort: - External sorting techniques are used when the collection of complete data cannot reside in the main memory but must reside in secondary storage for example on a disk.

In this section we will discuss only internal sorting algorithms. Some of the internal sorting algorithms are bubble sort, insertion sort and selection sort. For any sorting algorithm important factors that contribute to measure their efficiency are the size of the data set and the method/operation to move the different elements around or exchange the elements. So counting the number of comparisons and the number of exchanges made by an algorithm provides useful performance measures. When sorting large set of data, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of time. For sorting a simple array of integers, the number of comparisons will be more important.

Let us discuss some of internal sorting algorithm and their complexity analysis in next section.

3.8.1 Bubble Sort

In this we will discuss the bubble sort algorithm and study its complexity analysis. A list of numbers is given as input that needs to be sorted. Let us explain the process of sorting via bubble sort with the help of following Tables

23	18	15	37	8	11
18	23	15	37	8	11
18	15	23	37	8	11
18	15	23	37	8	11
18	15	23	8	37	11
18	15	23	8	11	**37**

18	15	23	8	11	**37**
15	18	23	8	11	**37**
15	18	23	8	11	**37**
15	18	8	23	11	**37**
15	18	8	11	**23**	**37**

15	18	8	11	**23**	**37**
15	18	8	11	**23**	**37**
15	8	18	11	**23**	**37**
15	8	11	**18**	**23**	**37**

15	8	11	**18**	**23**	**37**
8	15	11	**18**	**23**	**37**
8	11	**15**	**18**	**23**	**37**

8	11	**15**	**18**	**23**	**37**
8	**11**	**15**	**18**	**23**	**37**

8	**11**	**15**	**18**	**23**	**37**

In this the given list is divided into two sub list sorted and unsorted. The largest element is bubbled from the unsorted list to the sorted sub list. After each

iteration/pass size of unsorted keep on decreasing and size of sorted sub list gets on increasing till all element of the list comes in the sorted list. With the list of n elements, n-1 pass/iteration are required to sort. Let us discuss the result of iteration shown in above tables.

In iteration 1, first and second element of the data set i.e 23 and 18 are compared and as 23 is greater than 18 so they are swapped. Then second and third element will be compared i.e 23 and 15 , again 23 is greater than 15 so swapped. Now 23 and 37 is compared and 23 is less than 37 so no swapping take place. Then 37 and 8 is compared and 37 is greater than 8 so swapping take place. At the end 37 is compared with 11 and again swapped. As a result largest element of the given data set i.e 37 is bubbled at the last position in the data set. Similarly we can perform other iterations of bubble sort and after n-1 iteration we will get the sorted list.

The algorithm for above sorting method is as below:

```
// a is the list of n elements to be sorted
function bubblesort(a,n)
{
        int i,j,temp,flag=true;

        for(i=0; i<n-1 && flag==true; i++) // outer loop

        {
                flag=false
                for(j=0; j<n-i-1; j++) // inner loop
                {

                        if(a[j]>a[j+1])

                        {
                                flag=true
                                temp = a[j];            // exchange
                                a[j] = a[j+1]; // exchange
                                a[j+1] = temp; // exchange


                        }

                }

        }
}
```

Complexity analysis of bubble sort is as follows.

Best-case:

When the given data set in an array is already sorted in ascending order the number of moves/exchanges will be 0, then it will be clear that the array is already in order because no two elements need to be swapped. In that case, the sort should end, which takes O(1). The total number of key comparisons will be (n-1) so complexity in best case will be O(n).

Worst-case:

In this case the given data set will be in descending order that need to be sorted in ascending order. Outer loop in the algorithm will be executed n-1 times (as i ranges from 0 to n-2, when i will be n-1 it will exit from the loop).

The number of comparison and exchanges is depicted below:

i	j ranges between (no. of comparisons)	No. of exachange
0	0 to n-2	3(n-1)
1	0 to n-3	3(n-2)
2	0 to n-4	3(n-3)
---	---	---
---	---	---w
n-2	0 to1	3(1)

Total number of exchanges will be the following summation $3*(1+2+...+n-1) = 3 * n*(n-1)/2$ i.e $O(n^2)$. The number of key comparison will be $(1+2+...+n-1)= n*(n-1)/2$ i.e $O(n^2)$. Hence complexity in worst case will be $O(n^2)$.

Average –case:

In this case we have to consider all possible initial data arrangement. So as in case of worst case, outer loop will be executed n-1 times. The number of exchanges will be $O(n^2)$. The number of key comparison will be i.e $O(n^2)$.So the complexity will be $O(n^2)$.

3.8.2 Insertion Sort

This sort is usually applied by card players or for the insertion of new elements into a sorted sequence. It is more appropriate for small input size of data/list. Let us consider a data set to discuss the method of insertion sort as follows:

23	18	15	37	8	11
23	**18**	15	37	8	11
18	**23**	**15**	37	8	11
15	**18**	**23**	**37**	8	11
15	**18**	**23**	**37**	**8**	11
8	**15**	**18**	**23**	**37**	**11**
8	**11**	**15**	**18**	**23**	**37**

In insertion sort, the list will be divided into two parts sorted and unsorted. In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub list, and inserted at the appropriate place. In each pass the algorithm inserts each element of the array into its proper position. A list of n elements will take at most n-1 passes to sort the given data data in ascending order.

For the input data set under consideration, let us discuss iterations of insertion sort algorithm. In first iteration first element of the list and second element of the list are compared i.e 23 and 18. As 23 is greater than 18 , so they are exchanged. In second iteration third element of the list i.e 15 is compared with second element and it is less than that so second element 23 is shifted down then it is compared with first element

and it is less than that also so first element will also be shifted down. Now there is no more element above that so third element 15 appropriate position is first. This way rest of the element will get their right position and sorted list will be obtained.

The algorithm for the insertion sort is as below:
// a is an array or the list of n element to be sorted in ascending order

```
function insertionSort(a,n)
{
        int i,j,key;

        for (i=1;i< n;i++)              //outer loop
        {
                key= a[i]  //exchange
                j = i-1
                while (j >= 0 and a[j] > key)   // inner loop
                {
                        a[j+1] = a[j]  //exchange
                        j = j-1
                }
                a[j+1] = key            //exchnage
        }
}
```

Running time depends not only on the size of the array but also the contents of the array i.e already data is sorted or in descending order. Complexity analysis of insertion sort algorithm is as follows.

Best-case:

In best case array data is already sorted in ascending order. Then inner loop will not be executed at all and the number of moves/exchanges will be 2*(n-1) i.e O(n). The number of key comparisons will be n-1 i.e O(n). So complexity in best case will be O(n).

Worst case:

In worst case data element of the array will be given in descending order. In the outer loop of above algorithm i range from 1 to n-1. So, the inner loop in the algorithm will be executed n-1 times.

The number of moves for outer loop exchanges will be 2(n-1).

i	Outerloop exchange	Innerloop exchange	Inner loop comparison
1	2	1	1
2	2	2	2
3	2	3	3
n-1	2	n-1	n-1

The number of exchanges will be 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 i.e $O(n^2)$.

The number of key comparison will be (1+2+...+n-1)= n*(n-1)/2 i.e $O(n^2)$. Hence complexity in worst case will be $O(n^2)$.

Average case:

In this case we have to consider all possible initial data arrangement. It is difficult to figure out the average case. i.e. what will be probability of data set either in mixed / random input. We can not assume all possible inputs before hand and all cases will be equally likely. For most algorithms average case is same as the worst case. So as in case of worst case, outer loop will be executed n-1 times. The number of moves/assignment will be $O(n^2)$. The number of key comparison will be i.e $O(n^2)$. So the complexity in average case will be $O(n^2)$.

3.8.3 Selection Sort

Now we will discuss the selection sort algorithm and its complexity analysis. A list of numbers is given as input that needs to be sorted. Let us explain the process of sorting via selection sort with the help of following Tables

23	18	15	37	8	11

8	18	15	37	23	11

8	**11**	15	37	23	18

8	**11**	**15**	37	23	18

8	**11**	**15**	37	23	18

8	**11**	**15**	**18**	23	37

8	**11**	**15**	**18**	**23**	37

In this algorithm the list will be divided into two sub lists, sorted and unsorted. Here we find the smallest element of the list and replace it by the first element of the list i.e beginning element of the given list. Then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted. After each selection and swapping, the two sub lists will be there where first sub list move one element ahead, increasing the number of sorted elements and second sub list decreasing the number of unsorted elements by one. In one pass we move one element from the unsorted sublist to the sorted sublist. A list of n elements requires n-1 passes to completely rearrange the data in sorted i.e ascending order.

For given data set, in first iteration minimum from the complete list is obtained i.e 8 so this will be exchanged with first position in the list i.e 23. Then in second iteration minimum from the remaining list will be found out i.e 11 and exchanged with second position element of the list i.e 18. This process will be continued for rest of the list also and finally we will get sorted list.

The algorithm for the insertion sort is as below:
// a is an array or the list of n element to be sorted

```
function selectionsort (a,n)
{
```

```
        int i, j;
        int min, temp;

        for (i = 0; i < n-1; i++)        //outer loop
        {
                min = i;
                for (j = i+1; j < n; j++)          //inner loop
                {
                        if (a[j] < a[min])
                                min = j;
                }
                temp = a[i];
                a[i] = a[min];          } Swap operation
                a[min] = temp;
        }
}
```

In selectionsort function, the outer for loop executes n-1 times. Swap operation once at each iteration of outer loop. Total number of Swaps will be n-1 and in each swap operation three moves or assignment are performed. This gives the total moves/assignment for outer loop as 3*(n-1). The inner loop executes the size of the unsorted list minus 1 i.e from i+1 to n-1 for every iterations of outer loop. Number of key comparison for each iteration of inner loop is one. Total number of key comparisons will be equal to 1+2+...+n-1 = n*(n-1)/2 So, Selection sort complexity is $O(n^2)$.

The following table will describe about number of moves and comparison.

i	j range	No. of moves in outer loop	No. of comparison in inner loop
0	1 to n-1	3	1
1	2 to n-1	3	1
2	3 to n-1	3	1
n-2	n-1 to n-1	3	1

Let us summarize the number of moves and comparison for selection sort algorithm.

Total moves/assignment for outer loop = 3*(n-1).

Total number of key comparisons = 1+2+...+n-1 = n*(n-1)/2 = $O(n^2)$

So, Selection sort algorithm complexity is $O(n^2)$.

The best case, the worst case, and the average case complexity of the selection sort algorithm are same that is $O(n^2)$. As none of the loop in the algorithm is dependent on the type of data either it is already sorted or in reverse order or mixed. It indicates that behavior of the selection sort algorithm does not depend on the initial organization of data.

Following Table summarizes the above discussed results for different sorting algorithms.

Algorithm	Best Case	Worst Case	Average Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

☞ Check Your Progress 5

1. Write advantages and disadvantages of linear search algorithm.

……
……
……

2. Write the iterations for sorting the following list of numbers using bubble sort, selection sort and insertion sort:
   45, 67, 12, 89, 1, 37, 25, 10

……
……
……
……
……
……

3.8 SUMMARY

In this unit various categories of algorithm and their analysis is described like GCD, matrix multiplication, polynomial evaluation, searching and sorting. For GCD computation Euclid's algorithm is explained and complexity in best case O(1) , worst case O(m) or O(n) depending upon input and in average case it is O(log n). Horner's rule is discussed to evaluate the polynomial and its complexity is O(n) where n will be the degree of polynomial. Basic matrix multiplication is explain for finding product of two matrices of order nxn with time complexity in the order of $O(n^3)$. For exponent evaluation both approaches i.e left to right binary exponentiation and right to left binary exponentiation is illustrated. Time complexity of these algorithms to compute x^n is O(log n). In large data set to access an element searching algorithm are required. Here linear search algorithm and its analysis are discussed. Sorting is the process of arranging a collection of data into either ascending or descending order. Classification of sorting algorithm based on data storage in primary memory or secondary memory. Internal sorting algorithms are applied where data to be sorted is stored in primary memory. Otherwise if input data can not be stored in primary memory and stored in secondary memory, external sorting techniques are used. In this unit few internal sorting algorithms like bubble, selection and insertion and their complexity analysis in worst case, best case and average are discussed.

3.9 MODEL ANSWERS

Check Your Progress 1:

Answers:

1. 595 = 2 x 252 + 91
   252 = 2 x 91 +70
   91 = 1 x 70 + 21
   70 = 3 x 21 + 7
   21 = 3 x 7 + 0

   GCD(595,252)= 7

   According to Euclid's algorithm, step (1) gcd(595,252)=gcd(252,91)
   step(2) gcd(252,91)=gcd(91, 70)
   step(3) gcd(91,70)=gcd(70,21)
   step (4) gcd(70,21)=gcd(21,7)
   step (5) gcd (21,7)= gcd (7,0)
   Combining above all gives gcd(595, 252)=7 which is the last divisor that gives remainder 0.

2. In Euclid's algorithm, at each step remainder decreases at least by 1. So after finite number of steps remainder must be 0. Non zero remained gives GCD of given two numbers.

Check Your Progress 2:

Answers:

1. Show the steps of Horner's rule for $p(x)= 3x^4+2x^3-5x+7$ at x=2

poly=0 , array a[5]={7,-5,0,2,3}

Iteration 1,
poly = x * 0 + a[4] = 3

Iteration 2,
poly = x * 3 + a[3]
= 2 * 3 + 2 = 6 +2 = 8

Iteration 3,
poly = x * 8 + a[2]
= 2 * 8 + 0 = 16 + 0 = 16

Iteration 4,
poly = x * 16 + a[1]
= 2 * 16 + (-5) = 32 -5 = 27

Iteration 5,
poly = x * 27 + a[0]
= 2 * 27 + 7 = 54 + 7 = 61

2. A basic (general) algorithm:

/* a is an array with polynomial coefficient, n is degree of polynomial, x is the point at which polynomial will be evaluated */

```
function(a[n], n, x)
{
        poly = 0;

        for ( i=0; i <= n; i++)
        {
                result =1;
                for (j=0; j<i; j++)
                {
                        result= result * x;
                }
                poly= poly + result *a[i];
        }
        return poly.
}
```

Time Complexity of above basic algorithm is $O(n^2)$ where n is the degree of the polynomial. Time complexity of the Horner's rule algorithm is O(n) for a polynomial of degree n. Basic algorithm is inefficient algorithm in comparison to Horner's rule method for evaluating a polynomial.

Check Your Progress 3:

Answers:

1. C program to find two matrices A[3x3] and B[3x3]

```
#include<stdio.h>
int main()
{
         int a[3][3],b[3][3],c[3][3],i,j,k,sum=0;


        printf("\nEnter the First matrix->");
        for(i=0;i<3;i++)
                for(j=0;j<3;j++)
                        scanf("%d",&a[i][j]);
        printf("\nEnter the Second matrix->");
        for(i=0;i<3;i++)
                for(j=0;j<3;j++)
                        scanf("%d",&b[i][j]);
        printf("\nThe First matrix is\n");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                {
                        printf("%d\t",a[i][j]);
                }
        }
        printf("\nThe Second matrix is\n");
        for(i=0;i<3;i++)
```

```
        {
                printf("\n");
                for (j=0;j<3;j++)
                        printf("%d\t",b[i][j]);

        }
        for(i=0;i<3;i++)
                for(j=0;j<3;j++)
                        c[i][j]=0;

        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                sum=0;
                        for(k=0;k<3;k++)
                                sum=sum+a[i][k]*b[k][j];
                        c[i][j]=sum;
                }
        }

        printf("\nThe multiplication of two matrix is\n");
        for(i=0;i<3;i++)
        {
                printf("\n");
                for(j=0;j<3;j++)
                        printf("%d\t",c[i][j]);

        }
        return 0;
}
```

Check Your Progress 4

Answers:

1. Left to right binary exponentiation for a^{283} is as follows:
   n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
   result = a

Iteration no.	i	Bit	result
1	7	0	a^2
2	6	0	a^4
3	5	0	a^8
4	4	1	$(a^8)^2*a= a^{17}$
5	3	1	$(a^{17})^2*a= a^{35}$
6	2	0	$(a^{35})^2= a^{70}$
7	1	1	$(a^{70})^2*a= a^{141}$
8	0	1	$(a^{141})^2*a= a^{283}$

   Right to left binary exponentiation for a^{283} is as follows:
   n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
   result = a (since A[0]=1)

Iteration no.	i	Bit	x	result
1	1	1	a^2	$a *a^2 = a^3$

2	2	0	a^4	a^3
3	3	1	a^8	$a^3 * a^8 = a^{11}$
4	4	1	$(a^8)^2$	$a^{16} * a^{11} = a^{27}$
5	5	0	$(a^{16})^2$	(a^{27})
6	6	0	$(a^{32})^2$	a^{27}
7	7	0	$(a^{64})^2$	a^{27}
8	8	1	$(a^{128})^2$	$(a^{256})* a^{27} = a^{283}$

Check Your Progress 5:

Answers:

1. Linear search algorithm is easy to write and efficient for short list i.e input data list is small in size. It does not have any prerequisite like data should be sorted or not sorted.

However, it is lengthy and time consuming where data set is large in size. There is no quicker method to identify whether the item to be searched is present in the list or not. The linear search situation will be in worst case if the element is at the end of the list. In case of element is not present in the list then also whole list is required to be searched.

2. List of numbers to be sorted 45, 67, 12, 89, 1, 37, 25, 10.

Bubble sort:

Iteration 1:

45,67,12,89,1,37,25,10
45,12,67,89,1,37,25,10
45,12,67,89,1,37,25,10
45,12,67,1,89,37,25,10
45,12,67,1,37,89,25,10
45,12,67,1,37,25,89,10
45,12,67,1,37,25,10,**89**

Iteration 2:

45,12,67,1,37,25,10
12,45,67,1,37,25,10
12,45,67,1,37,25,10
12,45,1,67,37,25,10
12,45,1,37,67,25,10
12,45,1,37,25,67,10
12,45,1,37,25,10,**67**

Iteration 3:

12,45,1,37,25,10
12,45,1,37,25,10
12,1,45,37,25,10
12,1,37,45,25,10
12,1,37,25,45,10
12,1,37,25,10,**45**

Iteration 4:

12,1,37,25,10
1,12,37,25,10
1,12,37,25,10
1,12,25,37,10
1,12,25,10,**37**

Iteration 5:

1,12,25,10,
1,12,25,10,
1,12,25,10,
1,12,10,**25**

Iteration 6:

1,12,10,
1,12,10
1,10,**12**

Iteration 7:

1,10
1,**10**

Iteration 8:

1

Sorted list: 1,10,12,25,45,67,89

Selection Sort:

45, 67, 12, 89, 1, 37, 25, 10

1,67,12,89,45,25,10
1,10,12,89,45,25,67
1,10,12,89,45,25,67
1,10,12,25,45,89,67
1,10,12,25,45,89,67
1,10,12,25,45,67,89
1,10,12,25,45,67,89

Insertion Sort:

45, 67, 12, 89, 1, 37, 25, 10

45,67,12,89,1,37,25,10
12,45,67,89,1,37,25,10
12,45,67,89,1,37,25,10
1, 12,45,67,89,37,25,10
1,12,37,45,67,89,25,10
1,12,25,37,45,67,89,10
1,10,12, 25,37,45,67,89

3.10 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, "Introduction to Algorithms", 2 nd Ed., PHI, 2004.

2. Robert Sedgewick, "Algorithms in C", 3rd Edition, Pearson Education, 2004

3. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajasekaran, "Fundamentals of Computer algorithms", 2nd Edition, Universities Press, 2008

4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education, 2003.

UNIT 1 GREEDY TECHNIQUES

Structure **Page Nos.**

1.0 INTRODUCTION

Greedy algorithms are typically 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 w. r. t. some constraints or conditions. Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it ***solution set***, S where S$\subseteq C$) that satisfies the given constraints or conditions. Any subset S $\subseteq C$, which satisfies the given constraints, is called a ***feasible*** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called an **optimal solution**.

A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say ***select***). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. The input we tried and rejected is never considered again. When a greedy algorithm works correctly, the first solution found in this way is always optimal.
In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say$x$, from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{x\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.
   (Note: From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function (either maximize or minimize, as the case may be), is called ***optimal solution***.

Characteristics of greedy algorithm

- Used to solve optimization problem
- Most general, straightforward method to solve a problem.
- Easy to implement, and if exist, are efficient.
- Always makes the choice that looks best at the moment. That is, it makes a ***locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.***
- Once any choice of input from C is rejected then it never considered again.
- Do not always yield an optimal solution; but for many problems they do.

In this unit, we will discuss those problems for which greedy algorithm gives an optimal solution such as Knapsack problem, Minimum cost spanning tree (MCST) problem and Single source shortest path problem.

1.1 OBJECTIVES

After going through this Unit, you will be able to:

- Understand the basic concept about Greedy approach to solve Optimization problem.
- Understand how Greedy method is applied to solve any optimization problem such as Knapsack problem, Minimum-spanning tree problem, Shortest path problem etc.

1.2 SOME EXAMPLES TO UNDERSTAND GREEDY TECHNIQUES

In order to better understand the greedy algorithms, let us consider some examples: Suppose we are given Indian currency notes of all denominations, e.g. {1,2,5,10,20,50,100,500,1000}. The **problem** is to ***find the minimum number of currency notes to make the required amount A, for payment***. Further, it is assumed that currency notes of each denomination are available in sufficient numbers, so that one may choose as many notes of the same denomination as are required for the purpose of using the minimum number of notes to make the amount A.

Now in the following examples we will notice that for a problem (discussed above) the greedy algorithm ***provides a solution*** (see example-1), some other cases, greedy algorithm does ***not provides a solution***, even when a solution by some other method exist (see example-2) and sometimes greedy algorithm does ***not provides an optimal solution*** Example-3).

Example 2

Solution: Intuitively, to begin with, we pick up a note of denomination D, satisfying the conditions.

i) $D \leq 289$ and
ii) if D1 is another denomination of a note such that $D1 \leq 289$, then $D1 \leq D$.

In other words, the picked-up note's denomination D is the largest among all the denominations satisfying condition (i) above.

The above-mentioned step of picking note of denomination D, satisfying the above two conditions, is repeated till either the amount of Rs.289/- is formed or we are clear that we can not make an amount or Rs.289/- out of the given denominations.

We apply the above-mentioned intuitive solution as follows:

To deliver Rs. 289 with minimum number of currency notes, the notes of different denominations are chosen and rejected as shown below:

Chosen-Note-Denomination	Total-Value-So far
100	0+100 ≤ 289
100	100+100= ≤ 289
~~100~~	~~200+100 > 289~~
50	200+50 ≤ 289
~~50~~	~~250+50 > 289~~
20	250 + 20 ≤ 289
~~20~~	~~270 + 20 > 289~~
10	270 + 10 ≤ 289
~~10~~	~~280 + 10 > 289~~
5	280 + 5 ≤ 289
~~5~~	~~285 + 5 > 289~~
2	285 + 2 < 289
2	287 + 2 = 289

The above sequence of steps based on Greedy technique, constitutes an algorithm to solve the problem.

To summarize, in the above mentioned solution, we have used the strategy of choosing, at any stage, the maximum denomination note, subject to the condition that the sum of the denominations of the chosen notes does not exceed the required amount A = 289.

The above strategy is the essence of greedy technique.

Example 2

Next, we consider an example in which for a given amount A and a set of available denominations, the greedy algorithm does not provide a solution, even when a solution by some other method exists.

Let us consider a hypothetical country in which notes available are of only the denominations 20, 30 and 50. We are required to collect an amount of 90.

Attempted solution through above-mentioned strategy of greedy technique:

i) First, pick up a note of denomination 50, because $50 \leq 90$. The amount obtained by adding denominations of all notes picked up so far is 50.

ii) Next, we can not pick up a note of denomination 50 again. However, if we pick up another note of denomination 50, then the amount of the picked-up notes becomes 100, which is greater than 90. Therefore, we do not pick up any note of denomination 50 or above.

iii) Therefore, we pick up a note of next denomination, viz., of 30. The amount made up by the sum of the denominations 50 and 30 is 80, which is less then 90. Therefore, we accept a note of denomination 30.

iv) Again, we can not pick up another note of denomination 30, because otherwise the sum of denominations of picked up notes, becomes 80+30=110, which is more than 90. Therefore, we do not pick up only note of denomination 30 or above.

v) Next, we attempt to pick up a note of next denomination, viz., 20. But, in that case the sum of the denomination of the picked up notes becomes 80+20=100, which is again greater than 90. Therefore, we do not pick up only note of denomination 20 or above.

vi) Next, we attempt to pick up a note of still next lesser denomination. However, there are no more lesser denominations available.

Hence greedy algorithm fails to deliver a solution to the problem.

However, by some other technique, we have the following solution to the problem: First pick up a note of denomination 50 then two notes each of denomination 20.

Thus, we get 90 and it can be easily seen that at least 3 notes are required to make an amount of 90. Another alternative solution is to pick up 3 notes each of denomination 30.

Example 3

Next, we consider an example in which the greedy technique, of course, leads to a solution, but the solution yielded by greedy technique is not optimal.

Again, we consider a hypothetical country in which notes available are of the only denominations 10, 40 and 60. We are required to collect an amount of 80.

Using the greedy technique, to make an amount of 80, first, we use a note of denomination 60. For the remaining amount of 20, we can choose note of only denomination 10. And , finally, for the remaining amount, we choose another note of denomination 10. Thus, greedy technique suggests the following solution using 3 notes: 80 = 60 + 10 + 10.

However, the following solution uses only two notes:

80 = 40 + 40

Thus, the solutions suggested by Greedy technique may not be optimal.

1.3 FORMALIZATION OF GREEY TECHNIQUE

In order to solve optimization problem using greedy technique, we need the following data structures and functions:

1) A candidate set from which a solution is created. It may be set of nodes, edges in a graph etc. call this set as:
   C: Set of given values or set of candidates

2) A solution set S (where S $\subseteq C$) , in which we build up a solution. This structure contains those candidate values, which are considered and chosen by the greedy technique to reach a solution. Call this set as:
   S: Set of selected candidates (or input) which is used to give optimal solution.

3) A function (say ***solution***) to test whether a given set of candidates give a solution (not necessarily optimal).

4) A selection function (say ***select***) which chooses the best candidate form C to be added to the solution set S,

5) A function (say ***feasible***) to test if a set S can be **extended** to a solution (not necessarily optimal) and

6) An objective function (say **ObjF**) which assigns a ***value*** to a solution, or a partial solution.

To better understanding of all above mentioned data structure and functions, consider the minimum number of notes problem of example1. In that problem:

1) C={1, 2, 5, 10,50,100,500,1000}, which is a list of available notes (in rupees). Here the set C is a multi-set, rather than set, where the values are repeated.

2) Suppose we want to collect an amount of Rs. 283 (with minimum no. of notes). If we allow a multi-set rather than set in the sense that values may be repeated, then S={100,100,50,20,10,2,1}

3) A function **solution** checks whether a solution is reached or not. However this function does not check for the optimality of the obtained solution. In case of minimum number of notes problem, the function ***solution*** finds the sum of all values in the multi-set S and compares with the fixed amount, say Rs. 283. If at any stage S={100,100, 50}, then sum of the values in the S is 250, which does not equal to the 283, then the function ***solution*** returns "solution not reached". However, at the later stage, when S={100,100,50,20,10,2,1}, then the sum of values in S equals to the required amount, hence the function ***solution*** returns the message of the form "solution reached".

4) A function ***select*** finds the "best" candidate value (say x) from C, then this value x is tried to add to the set S. At any stage, value x is added to the set S, if its addition leads to a partial (feasible) solution. Otherwise, x is rejected. For example, In case of minimum number of notes problem, for collecting Rs. 283, at the stage when S={100, 100,50}, then first the function ***select*** try to add the Rs 50 to S. But by using a function **solution**, we can found that the addition of Rs. 50 to S will lead us a infeasible solution, since the total value now becomes 300 which exceeds Rs. 283. So the value 50 is rejected. Next, the function ***select*** attempts the next lower denomination 20. The value 20 is added to the set S, since after adding 20, total sum in S is 270, which is less than Rs. 283. Hence, the value 20 is returned by the function ***select***.

5) When we select a new value (say x) using ***select*** function from set C, then before adding x to S we check its feasibility. If its addition gives a partial solution, then this value is added to S. Otherwise it is rejected. The feasibility checking of new selected value is done by the function ***feasible***. For example, In case of minimum number of notes problem, for collecting Rs. 283, at the stage when S={100, 100,50}, then first the function ***select*** try to add the Rs 50 to S. But by using a function **solution**, we can found that the addition of Rs. 50 to S will lead us an infeasible solution, since the total value now becomes 300 which exceeds Rs. 283. So the value 50 is rejected. Next, the function ***select*** attempts the next lower denomination 20. The value 20 is added to the set S, since after adding 20, total sum in S is 270, which is less than Rs. 283. Hence feasible.

6) The objective function (say ***ObjF***), gives the value of the solution. For example, In case of minimum number of notes problem, for collecting Rs. 283; and when S={100,100,50,20,10,2,1}, then the sum of values in S equals to the required amount 283; the function ***ObjF*** returns the number of notes in S, i.e., the number 7.

A general form for greedy technique can be illustrated as:

```
Algorithm Greedy(C, n)

    /* Input: A input domain (or Candidate set ) C of size n, from which solution is to be
              Obtained. */

   // function select (C: candidate_set)  return an element (or candidate).
   // function solution (S: candidate_set) return Boolean
   // function feasible (S: candidate_set) return Boolean
    /* Output: A solution set S, where S ⊆ C, which maximize or minimize the selection
               criteria w. r. t. given constraints */
{
        S ← ϕ                          // Initially a solution set S is empty.
       While ( not solution(S) and C ≠ ϕ)
         {
            x ← select(C)              /* A "best" element x is selected from C which
                                          maximize or minimize the selection criteria. */
            C ← C − {x}                   /* once x is selected , it is removed from C
            if ( feasible(S ∪ {x}) then   /* x is now checked for feasibility
               S ← S ∪ {x}
         }
       If (solution (S))
          return S;
       else
          return " No Solution"
    } // end of while
```

Now in the following sections, we apply greedy method to solve some optimization problem such as knapsack (fractional) problem, Minimum Spanning tree and Single source shortest path problem etc.

1.4 KNAPSACK (FRACTIONAL) PROBLEM

The fractional knapsack problem is defined as:

- Given a list of n objects say $\{I_1, I_2 \dots \dots, I_n\}$ and a Knapsack (or bag).
- Capacity of Knapsack is M.
- Each object I_i has a weight w_i and a profit of p_i .
- If a fraction x_i (where $x_i \in \{0, \dots, 1\}$) of an object I_i is placed into a knapsack then a profit of $p_i x_i$ is earned.

The **problem** (or Objective) is to fill a knapsack (up to its maximum capacity M) which maximizes the total profit earned.

Mathematically:

$$Maximize\ (the\ profit) \sum_{i=1}^{n} p_i x_i \ ;\ subjected\ to\ the\ constraints$$

$$\sum_{i=1}^{n} w_i x_i \ \leq \text{M and} \ \ x_i \in \{0, \dots, 1\}, 1 \leq \text{i} \leq \text{n}$$

Note that the value of x_i will be any value between 0 and 1 (inclusive). If any object I_i is completely placed into a knapsack then its value is 1 $(i.e. x_i = 1)$, if we do not pick (or select) that object to fill into a knapsack then its value is 0 $(i.e. x_i = 0)$. Otherwise if we take a fraction of any object then its value will be any value between 0 and 1.

To understand this problem, consider the following instance of a knapsack problem:

$Number\ of\ objects;\ n = 3$
Capacity of Knapsack; M=20
$(p_1, p_2, p_3) = (25,24,15)$
$(w_1, w_2, w_3) = (18,15,10)$

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i/w_i that fits into the knapsack.

Let us apply all above 3 approaches on the given knapsack instance:

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$(1, \frac{2}{15}, 0)$	18+2+0=20	28.2
2	$< 0, \frac{2}{3}, 1 >$	0+10+10=20	31.0
3	$< 0, 1, \frac{1}{2} >$	0+15+5=20	31.5

Approach 1: (selection of object in decreasing order of profit):

In this approach, we select those object first which has maximum profit, then next maximum profit and so on. Thus we select 1st object (since its profit is 25, which is maximum among all profits) first to fill into a knapsack, now after filling this object ($w_1 = 18$) into knapsack remaining capacity is now 2 (i.e. 20-18=2). Next we select the 2nd object, but its weight w_2=15, so we take a fraction of this object (i.e. $x_2 = \frac{2}{15}$). Now knapsack is full (i.e. $\sum_{i=1}^{3} w_i x_i = 20$) so 3rd object is not selected. Hence we get total profit $\sum_{i=1}^{3} p_i x_i$ = 28 units and the solution set $(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$

Approach 2: (Selection of object in increasing order of weights).
In this approach, we select those object first which has minimum weight, then next minimum weight and so on. Thus we select objects in the sequence 2nd then 3rd then 1st. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i$ = 31.0 units and the solution set $(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$.

Approach 3: (Selection of object in decreasing order of the ratio p_i/w_i).
In this approach, we select those object first which has maximum value of p_i/w_i , that is we select those object first which has maximum profit per unit weight .
Since (p_1/w_1 , p_2/w_2 , p_3/w_3)=(1.3, 1.6, 1.5). Thus we select 2nd object first , then 3rd object then 1st object. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i$ = 31.5 units and the solution set $(x_1, x_2, x_3) = (0,1, \frac{1}{2},)$.

Thus from above all 3 approaches, it may be noticed that

- Greedy approaches **do not always yield an optimal solution**. In such cases the greedy method is frequently the basis of a heuristic approach.
- Approach3 **(S**election of object in decreasing order of the ratio p_i/w_i) gives a optimal solution for knapsack problem.

A pseudo-code for solving knapsack problem using greedy approach is :

```
        Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
                /* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered such
                that
                X[1..n] is a solution set and M is the capacity of KnapSack*/
        {
1:              For i ← 1 to n do
2:                      X[i] ← 0
3:                      profit ← 0          //Total profit of item filled in Knapsack
4:                      weight ← 0          // Total weight of items packed in KnapSack
5:                      i←1
6:              While (Weight < M) // M is the Knapsack Capacity
                {
7:                      if (weight + W[i] ≤ M)
8:                              X[i] = 1
9:                              weight = weight + W[i]
10:                     else
11:                             X[i] = (M-wright)/w[i]
12:                             weight = M
13:             Profit = profit = profit + p [i]*X[i]
14:             i++;
        }//end of while
        }//end of Algorithm
```

Running time of Knapsack (fractional) problem:

Sorting of n items (or objects) in decreasing order of the ratio p_i/w_i takes $O(nlogn)\ time$. Since this is the lower bound for any comparison based sorting algorithm. Line 6 of ***Greedy Fractional-Knapsack*** takes $O(n)$ time. Therefore, the total time including sort is $O(nlogn)$.

Example: 1: Find an optimal solution for the knapsack instance n=7 and M=15 ,

$(p_1, p_2 \ldots\ldots, p_7) = (10, 5, 15, 7, 6, 18, 3)$
$(w_1, w_2, \ldots\ldots, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Solution:

Greedy algorithm gives a optimal solution for knapsack problem if you select the object in decreasing order of the ratio p_i/w_i . That is we select those object first which has maximum value of the ratio p_i/w_i ; $for\ all\ i = 1,..,7$. This ratio is also called profit per unit weight .

Since $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \ldots, \frac{p_7}{w_7}\right) = (5, 1.67, 3, 1, 6, 4.5, 3)$. Thus we select 5th object first , then 1st object, then 3rd (or 7th) object, and so on.

Approach	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
Selection of object in decreasing order of the ratio p_i/w_i	$(1, \frac{2}{3}, 1, 0, 1, 1, 1)$	1+2+4+5+1+2 =15	6+10+18+15+3+3.33 =55.33

1.5 MINIMUM COST SPANNING TREE (MCST) PROBLEM

Definition: **(Spanning tree):** Let G=(V,E) be an undirected connected graph. A ***subgraph*** T=(V,E') of G is a spanning tree of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.

Definition: **(Minimum cost Spanning tree):**

Suppose G is a **weighted connected graph**. A ***weighted graph*** is one in which every edge of G is assigned some positive weight (or length). A graph G is having several spanning tree.

In general, a *complete graph* (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if n=4 then total number of spanning tree is 16.
A ***minimum cost spanning tree*** (MCST) of a weighted connected graph G is that spanning tree whose sum of length (or weight) of all its edges is minimum, among all the possible spanning tree of G.

For example: consider the following **weighted connected graph** G (as shown in figure-1). There are so many spanning trees (say $T_1, T_2, T_3, \ldots\ldots$) are possible for G. Out of all possible spanning trees, four spanning trees T_1, T_2, T_3 and , T_4 of G are shown in figure a to figure d.

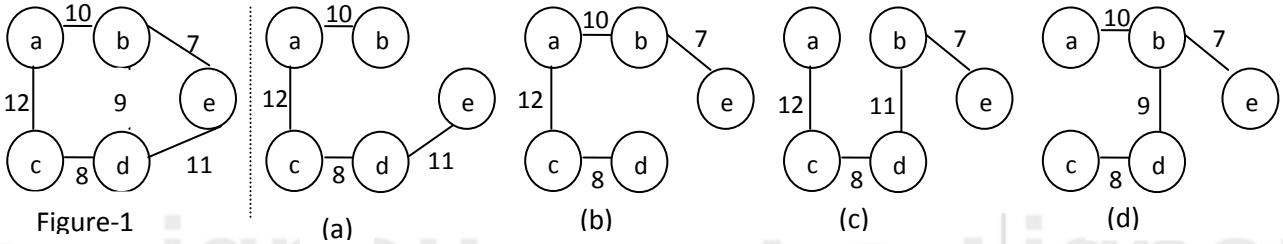


Figure-1 (a) (b) (c) (d)

A sum of the weights of the edges in T_1, T_2, T_3 and , T_4 is: 41, 37, 38 and 34 (some other spanning trees T_5, T_6, ... are also possible). We are interested to find that spanning tree, out of all possible spanning trees $T_1, T_2, T_3, T_4, T_5 \ldots\ldots$; whose sum of weights of all its edges are minimum. For a given graph G , T_3 is the MCST, since weight of all its edges is minimum among all possible spanning trees of G.

Application of spanning tree:

- ***Spanning trees are widely used in designing an efficient network***.
  For example, suppose we are asked to design a network in which a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. This problem can be converted into a graph problem in which nodes are telephones (or computers), undirected edges are potential links. The goal is to pick enough of these edges that the nodes are connected. Each link (edge) also has a maintenance cost, reflected in that edge's weight. Now question is "what is the cheapest possible network? An answer to this question is MCST, which connects everyone at a minimum possible cost.
- Another application of MCST is in the **designing of efficient routing algorithm**.
  Suppose we want to find a airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, when we travel more, the more it will cost. So MCST can be applied to optimize airline routes by finding the least costly paths with no cycle.

To find a MCST of a given graph G, one of the following algorithms is used:

1. **Kruskal's algorithm**
2. **Prim's algorithm**

These two algorithms use Greedy approach. A greedy algorithm selects the edges one-by-one in some given order. The next edge to include is chosen according to some optimization criteria. The simplest such criteria would be to choose an edge (u, v) that results in a minimum increase in the sum of the costs (or weights) of the edges so for included.

In General for constructing a MCST:

- We will build a set *A* of edges that is always a subset of some MCST.
- Initially, *A* has no edges (i.e. empty set).
- At each step, an edge (u, v) is determined such that $A \cup \{(u, v)\}$ is also a subset of a MCST. This edge *(u, v)* is called a ***safe edge.***
- At each step, we always add only ***safe edges*** to set A.
- **Termination:** when all *safe* edges are added to *A*, we stop. Now *A* contains a edges of spanning tree that is also an MCST.

Thus a general MCST algorithm is:

```
GENERIC_MCST(G, w)
  {
    A ← ϕ
    While A is not a spanning tree
     {
       find an edge (u, v) that is safe for A
       A ← A ∪ {(u, v)}
     }
    return A
  }
```

A main ***difference*** between ***kruskal's*** and ***Prim's*** algorithm to solve MCST problem is that the order in which the edges are selected.

Kruskal's Algorithm	Prim's algorithm
• Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST. • In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there are may be more than one connected components are possible. • Time complexity: $O(\|E\| log \|V\|)$	• Prim's algorithm always selects a vertex (say, v) to find MCST. • In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there will be only one connected components are possible • Time complexity: $O(\|V\|^2)$

For solving MCST problem using Greedy algorithm, we use the following data structure and functions, as mentioned earlier:

i) C: The set of candidates (or given values): Here C=E, the set of edges of $G(V, E)$.

ii) S: Set of selected candidates (or input) which is used to give optimal solution. Here the subset of edges, E' $(i.e. E' \subseteq E)$ is a solution, if the graph $T(V, E')$ is a spanning tree of $G(V, E)$.

iii) In case of MCST problem, the function ***Solution*** checks whether a solution is reached or not. This function basically checks :

a) All the edges in S form a tree.
b) The set of vertices of the edges in S equal to V.
c) The sum of the weights of the edges in S is minimum possible of the edges which satisfy (a) and (b) above.

1) ff selection function (say ***select***) which chooses the best candidate form C to be added to the solution set S,

iv) The ***select*** function chooses the best candidate from C. In case of **Kruskal's algorithm**, it selects an edge, whose **length is smallest** (from the remaining candidates). But in case of Prim's algorithm, it select a vertex, which is added to the already selected vertices, to minimize the cost of the spanning tree.

v) A function ***feasible*** checks the feasibility of the newly selected candidate (i.e. edge (u,v)). It checks whether a newly selected edge (u, v) form a cycle with the earlier selected edges. If answer is "yes" then the edge (u,v) is rejected, otherwise an edge (u,v) is added to the solution set S.

vi) Here the objective function **ObjF** gives the **sum of the edge lengths** in a **Solution.**

1.5.1 Kruskal's Algorithm

Let $G(V, E)$ is a connected, weighted graph.

Kruskal's algorithm finds a minimum-cost spanning tree (MCST) of a given graph G. It uses a ***greedy approach*** to find MCST, because at each step it adds an edge of least possible weight to the set A. In this algorithm:

- First we examine the edges of G in order of increasing weight.
- Then we select an edge $(u, v) \in E$ of minimum weight and checks whether its end points belongs to same component or different connected components.
- If *u* and *v* belongs to different connected components then we add it to set A, otherwise it is rejected because it create a cycle.
- The algorithm stops, when only one connected components remains (i.e. all the vertices of G have been reached).

Following pseudo-code is used to constructing a MCST, using Kruskal's algorithm:

```
   KRUSKAL_MCST(G, w)
       /* Input: A undirected connected weighted graph G=(V,E).
       /* Output:  A minimum cost spanning tree T(V, E') of G
   {
1.    Sort the edges of E in order of increasing weight
2.     A ← ϕ
3.     for (each vertex v ∈ V[G])
4.         do MAKE_SET(v)
5.    for (each edge (u, v)∈E, taken in increasing order of weight
          {
6.          if (FIND_SET(u) ≠ FIND_SET(v))
7.            A ← A ∪ {(u, v)}
8.            MERGE(u, v)
          }
9.    return A
   }
```

Kruskal's algorithm works as follows:

- First, we sorts the edges of E in order of increasing weight
- We build a set *A* of edges that contains the edges of the MCST. Initially A is empty.
- At line 3-4, the function **MAKE_SET(v),** make a new set {v} for all vertices of G. For a graph with n vertices, it makes n components of disjoint set such as $\{1\}, \{2\}, \ldots$ and so on.

- In line 5-8: An edge *(u, v)∈E,* of minimum weight is added to the set A, if and only if it joins two nodes which belongs to ***different*** components (to check this we use a $\boldsymbol{FIND\_SET}()$ function, which returns a same integer value, if u and v belongs to same components (In this case adding (u,v) to A creates a cycle), otherwise it returns a different integer value)
- If an edge added to A then the two components containing its end points are merged into a single component.
- Finally the algorithm stops, when there is just a single component.

Analysis of Kruskal's algorithm:

Let $|V| = n, the\ number\ of\ vertices$ and

$|E| = a, the\ number\ of Edges$

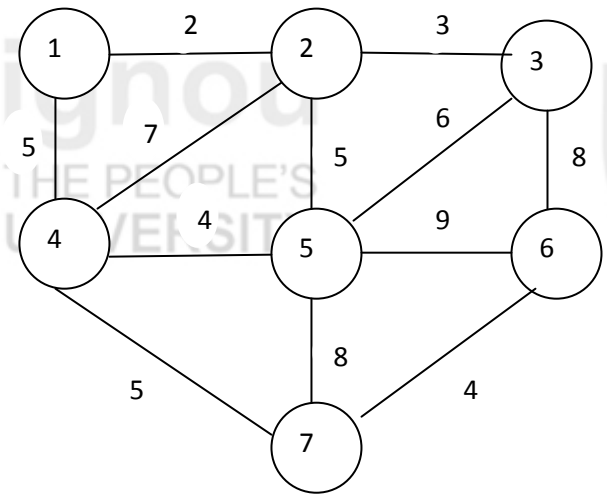
1. Sorting of edges requires $O(|E|log|E|) = O(aloga)\ time.$
2. Since, in any graph, minimum number of edges is $(n-1)$ and maximum number of edges (when graph is complete) is $n(n-1)/2$. Hence $n-1 \leq a \leq n(n-1)/2$. Thus $O(aloga) = O(alog\,(n(n-1)/2)) = O(alogn)$
3. Initializing n-disjoint set (in line 3-4) using **MAKE_SET** will requires O(n) time.
4. There are at most $2a$ **FIND_SET** operations (since there are a edges and each edge has 2 vertices) and $(n-1)$ **MERGE** operations. Thus we requires $O((2a + (n-1)logn)$ time.
5. At worst, O(a) time for the remaining operations.
6. For a connected graph, we know that $\boldsymbol{a \geq (n-1)}$**.** So the total time for Kruskal's algorithm is $O((2a + (n-1)logn) = O(alogn) = \boldsymbol{O(|E|\,log|V|)}$

Example: Apply Kruskal's algorithm on the following graph to find minimum-cost-spanning –

tree (MCST).

Solution: First, we sorts the edges of G=(V,E) in order of increasing weights as:

Edges	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5)	(2,4)	(3,6)	(5,7)	(5,6)
weights	2	3	4	4	5	5	5	6	7	8	8	9



The kruskal's Algorithm proceeds as follows:

STEP	EDGE CONSIDERED	CONNECTED COMPONENTS	SPANNING FORESTS (A)
Initialization	___	{1}{2}{3}{4}{5}{6}{7} (using line 3-4)	(1) (2) (3) (4) (5) (6) (7)
1.	(1, 2)	{1, 2},{3},{4},{5},{6},{7}	(1) – (2) (3) (4) (5) (6) (7)
2.	(2, 3)	{1,2,3},{4},{5},{6},{7}	(1)–(2)–(3) (4) (5) (6) (7)
3.	(4, 5)	{1,2,3},{4,5},{6},{7}	(1)–(2)–(3) (6) (7) (4)–(5)
4.	(6, 7)	{1,2,3},{4,5},{6,7}	(1)–(2)–(3) (4)–(5) (6) (7)
5.	(1, 4)	{1,2,3,4,5},{6,7}	(1)–(2)–(3) | (4)–(5) (6) (7)
6.	(2, 5)	Edge (2,5) is rejected, because its end point belongs to same connected component, so create a cycle.	
7.	(4, 7)	{1,2,3,4,5,6,7}	(1)–(2)–(3) | (4)–(5) (6) (7)

Total Cost of Spanning tree, T = 2+3+5+4+5+4=23

1.5.2 Prim's Algorithm

PRIM's algorithm has the property that the edges in the set A (this set A contains the edges of the minimum spanning tree, when algorithm proceed step-by step) always form a single tree, i.e. at each step we have only one connected component.

- We begin with one starting vertex (say v) of a given graph G(V,E).
- Then, in each iteration, we choose a minimum weight edge *(u, v)* connecting a vertex *v* in the set A to the vertices in the set$(V - A)$. That is, we always find an edge *(u, v)* of minimum weight such that $v \in A$ and $u \in V\text{-}A$. Then we modify the set A by adding *u* i.e.$A \leftarrow A \cup \{u\}$
- This process is repeated until $\neq V$, i.e. until all the vertices are not in the set A.

Following pseudo-code is used to constructing a MCST, using PRIM's algorithm:

```
   PRIMS_MCST(G, w)
      /* Input: A undirected connected weighted graph G=(V,E).
      /* Output:  A minimum cost spanning tree T(V, E') of G
   {
1.    T ← ϕ            // T contains the edges of the MST
2.    A ← { Any arbitrary menber of V}
3.    while (A ≠ V)
        {
4.        find an edge (u, v) of minimum weight s.t. u∈ V-A and v∈A
5.          A ← A ∪ {(u, v)}
6.          B ← B ∪ {u}
        }
7.    return T
   }
```

PRIM's algorithm works as follows:

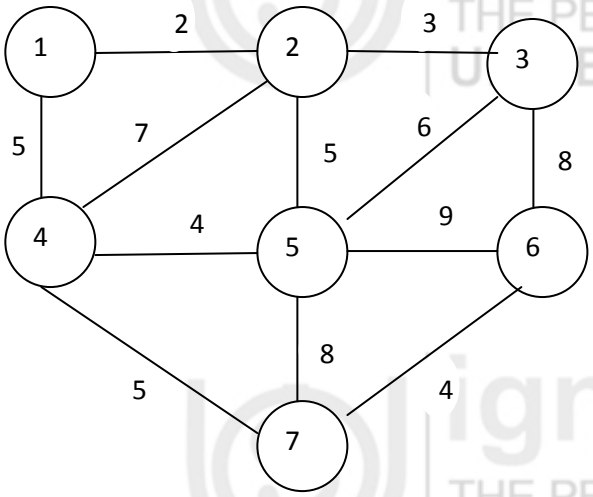
1) Initially the set A of nodes contains a single arbitrary node (i.e. starting vertex) and the set T of edges are empty.
2) At each step PRIM's algorithm looks for the shortest possible edge (u, v) such that
   $u \in V\text{-}A$ *and* $v \in A$
3) In this way the edges in T form at any instance a minimal spanning tree for the nodes in A. We repeat this process until $A \neq V$.

Time complexity of PRIM's algorithm

Running time of PRIM's algorithm can be calculated as follows:

- While loop at line-3 is repeated $|V| - 1 = (n - 1)$ times.
- For each iteration of while loop, the inside statements will require $O(n)$ time.
- So the overall time complexity is$O(n^2)$.

Example2: Apply PRIM's algorithm on the following graph to find minimum-cost-spanning – tree (MCST).



Solution: In PRIM's, First we select an arbitrary member of V as a starting vertex (say 1), then the algorithm proceeds as follows:

STEP	EDGE CONSIDERED (4, v)	CONNECTED COMPONENTS (Set A)	SPANNING FORESTS (set T)
Initialization	__	{1}	(1)
1	(1,2)	{1,2}	(1)−(2)
2	(2,3)	{1,2,3}	(1)−(2)−(3)
3	(1,4)	{1,2,3,4}	(1)−(2)−(3) | (4)
4.	(4,5)	{1,2,3,4,5}	(1)−(2)−(3) | (4)−(5)
5	(4,7)	{1,2,3,4,5,7}	(1)−(2)−(3) | (4)−(5) \ (7)
6	(6,7)	{1,2,3,4,5,6,7}	(1)−(2)−(3) | (4)−(5) (6) \ / (7)

Total Cost of the minimum spanning tree = 2+3+5+4+5+4
= 23

☞ Check Your Progress 1

Choose correct options from Q.1 to Q.7

Q.1: The essence of greedy algorithm is the …….. policy.
a) Maximization b) Minimization c) selection d) either a) or b)

Q2: The running time of KRUSKAL's **algorithm**, where |E| is the number of edges and |V| is the number of nodes in a graph:
a) $O(|E|)$ b) $O(|E|\log|E|)$ c) $O(|E|\log|V|)$ d) $O(|V|\log|V|)$

Q3: The running time of PRIM's algorithm, where |E| is the number of edges and |V| is the number of nodes in a graph:
a) $O(|E|^2)$ b) $O(|V|^2)$ c) $O(|E|\log|V|)$ d) $O(|V|\log|V|)$

Q.4: The optimal solution to the knapsack instance n=3, M=15, $(P_1, P_2, P_3) = (25,24,15)$ and $(W_1, W_2, W_3) = (18,15,10)$ is:
a) 28.2 b) 31.0 c) 31.5 d) 41.5

Q5: The solution set $X = (X_1, X_2, X_3)$ for the problem given in Q.4 is
a) $(\frac{1}{15}, \frac{2}{15}, 0)$ b) $(0, \frac{2}{3}, 1)$ c) $(0,1, \frac{1}{2})$ d) None of these

Q.6: Total number of spanning tree in a complete graph with 5 nodes are
a) 5^2 b) 5^3 c) 10 d) 100

Q.7: Let (i, j, C), where i and j indicates vertices of a graph & C denotes cost between edges. Consider the following edges & cost in order of increasing length: (b,e,3),(a,c,4),(e,f,4), (b,c,5),(f,g,5),(a,b,6), (c,d,,6),(e,f,6), (b,d,7), (d,e,7),(d,f,7),(c,f,7). Which of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

a) $(b,e),(e,f),(a,c),(b,c),(f,g),(c,d)$

b) $(b,e),(e,f),(a,c),(f,g),(b,c)(c,d)$

c) $(b,e),(a,c),(e,f),(b,c),(f,g),(c,d)$

d) $(b,e),(e,f),(b,c),(a,c),(f,g),(c,d)$

Q.8: Consider a question given in Q.7. Applying Kruskal's algorithm to find total cost of a Minimum spanning tree.

Q.9: State whether the following Statements are TRUE or FALSE. Justify your answer:

a) If e is a minimum edge weight in a connected weighted graph , it must be among the edges of at least one minimum spanning tree of the graph.

b) If e is a minimum edge weight in a connected weighted graph , it must be among the edges of each one minimum spanning tree of the graph.

c) If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.

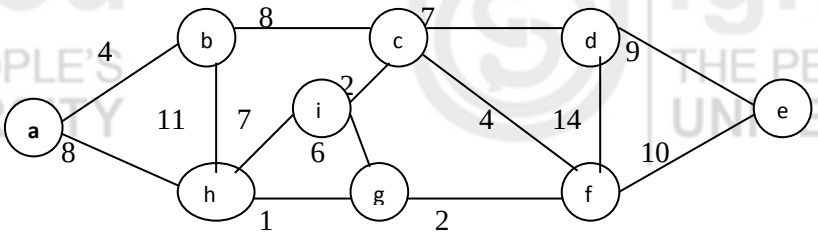
d) If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

e) If edge weights of a connected weighted graph are not all distinct, the minimum cost of each one minimum spanning tree is same.

Q.10: What is "*Greedy algorithm*" ? Write its pseudo code

Q.11: Differentiate between Kruskal's and Prim's algorithm to find a Minimum cost of a spanning tree of a graph G..

Q.12: Are the Minimum spanning tree of any graph is unique? Apply PRIM's algorithm to find a minimum cost spanning tree for the following.
Graph following using Prim's Algorithm. (a is a starting vertex).



Q.13: Find the optimal solution to the knapsack instance n=5, M=10,
$(P_1, P_2, ..., P_5) = (12,32,40,30,50)$ $(W_1, W_2, ..., W_5) = (4,8,2,6,1)$.

Q.14: Let S={a, b, c, d, e, f, g} be a collection of objects with Profit-Weight values as follows: a:(12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1) and g:(9,6). What is the optimal solution to the *fractional* knapsack problem for S, assuming we have a knapsack that can hold objects with total weight 18? What is the *complexity* of this method.

1.6 SINGLE SOURCE SHORTEST PATH PROBLEM (SSSPP)

Given: A directed graph $G(V, E)$ with weight edge $w(u, v)$.

- We define the **weight of path** $p =< v_0, v_1, \ldots, v_k >$ as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = sum\ of\ edge\ weights\ on\ path\ p.$$

- We can define the **shortest-path weight** from u to v as:

$$\delta(u, v) = \begin{cases} \min\{w(p): u \rightsquigarrow v\}\ ;\ if\ there\ exist\ a\ path\ u \rightsquigarrow v. \\ \infty\ ;\quad otherwise \end{cases}$$

Single-source-shortest path problem (SSSPP) problem:
Given a directed graph $G(V, E)$ with weight edge $w(u, v)$. We have to find a shortest path from source vertex $s \in V$ to every other vertex $v \in V - s$.

SSSP Problem can also be used to solve some other related problems:

- **Single-destination shortest path** problem(**SDSPP**): Find the transpose graph (i.e. reverse the edge directions) and use **single-source-shortest** path.

- **Single-pair shortest path (i.e.** a specific destination, say v): If we solve the SSSPP with source vertex s, we also solved this problem. Moreover, no algorithm for this problem is known that asymptotically faster than the SSSP in worst case.

- **All pair shortest path problem (APSPP):** Find a shortest path between every pair of vertices u and v. One technique is to use SSSP for each vertex, but there are some more efficient algorithm (known as Floyed-warshall's algorithm).

To find a SSSP for directed graphs $G(V, E)$, we have two different algorithms:

1. Bellman-Ford algorithm
2. Dijkstra's algorithm

- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$ or detect a negative weight cycles in G, hence ***no solution***. If there is no negative weight cycles are reachable (or exist) from source vertex s, then we can find a shortest path form source vertex $s \in V$ to every other vertex $v \in V$. If there exist a negative weight cycles in the input graph, then the algorithm can detect it, and hence "No solution".

- **Dijkstra's algorithm** allows only positive ***weight edges*** in the input graph and finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$.

To understand the basic concept of negative-weight cycle, consider the following 2 cases:

Case1: ***Shortest-path cannot contain a cycle***; ***it is just a simple path*** (i.e. no repeated vertex):

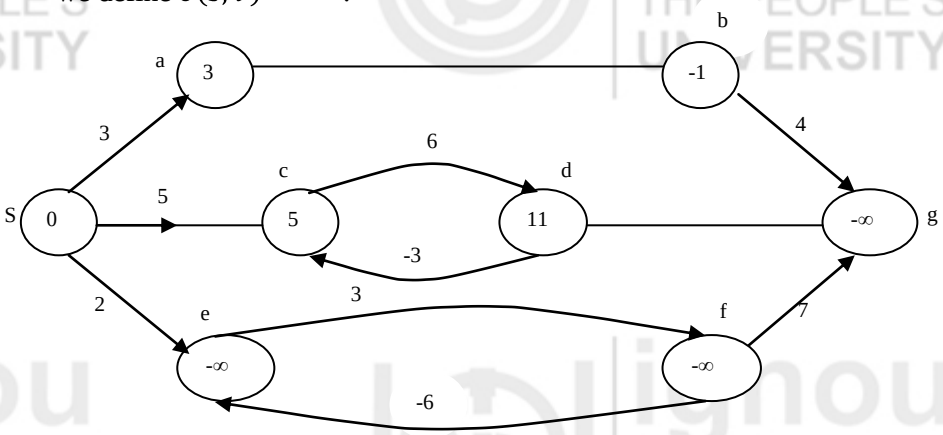
If some path from s to v contains a negative cost cycle, then there does not exist a shortest path. Otherwise there exists a shortest path (i.e. a simple path) from $u \rightsquigarrow v$.

Case 2: Graph containing a negative-weight cycle:

- No problem, if it is not reachable from the source vertex s.
- If it is reachable from source vertex s, then we just keep going around it and producing a path weight of $-\infty$. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty \; for \; all \; v \; on \; the \; cycle.$

For example: consider a graph with negative weight cycle:

If there is a negatives weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.



There are infinitely many paths from s to c: <s,c>,<a,c,d,c>,<s,c,d,c,d,c>, and so on.

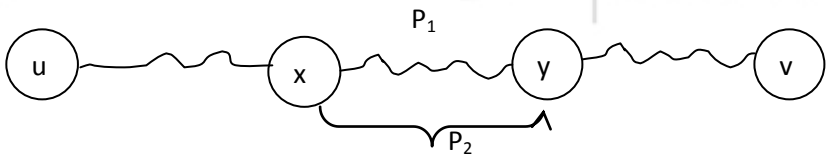
There are infinitely many paths from s to c: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$, and so on. Because the cycle $\langle c, d, c\rangle$ has weight 6 + (-3) = 3 > 0, the shortest path from s to c is $\langle s, c\rangle$, with weight $\delta(s, c) = 5$. Similarly, the shortest path from s to d is $\langle s, c, d\rangle$, with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from s to e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, and so on. Since the cycle $\langle e, f, e\rangle$ has weight 3 + (-6) = -3 < 0, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e, f, e\rangle$ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) = -\infty$. Because g is reachable from f , we can also find paths with arbitrarily large negative weights from s to g, and $\delta(s, g) = -\infty$. Vertices h, i, and j also form a negative-weight cycle. They are not reachable from s, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are non negative, as in the road-map example. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Shortest path : Properties

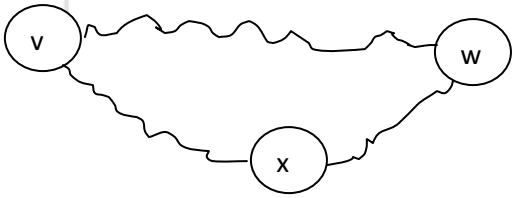
1. **Optimal sub-structure property**: Any sub-path of a shortest path is also a shortest path.

   Let P_1 is any sub-path (say x⇝y) of a shortest $s \rightsquigarrow v$, and P_2 is axy $x \rightsquigarrow y$ path; then the cost of $P_1 \leq$ cost of P_2; otherwise $s \rightsquigarrow v$ is not a shortest path.



2. **Triangle inequality**: Let $d(v, w)$ be the length of the shortest path from v to w, then

$$\boldsymbol{d(v, w) \leq d(v, x) + d(x, w}$$



3. Shortest path does not contains a cycle:

- Negative weight cycles are not allowed when it is reachable from source vertex s, since in this case there is no shortest path.
- If Positive –weight cycles are there then by removing the cycle, we can get a shorter path.

Generic Algorithm for solving single-source-shortest path (SSSP) problem:

Given a directed weighted graph $G(V, E)$, algorithm always maintain the following two fields for each vertex $v \in V$.

1. $\boldsymbol{d[v] = \delta(s, v)} =$
   $the\ length\ of\ the\ shortest\ path\ from\ starting\ vertex\ s\ to\ v,$
   (initially $d[v] = \infty$), and the value of $d[v]$ reduces as the algorithm progress. Thus we call $\boldsymbol{d[v]}$**,** a **shortest path estimate.**

2. $\boldsymbol{Pred}[v], which\ is\ a\ predecessor\ of\ v\ on\ a\ shortest\ path\ from\ s.$ The value of $\boldsymbol{Pred}[v]$ is either a another vertex or NIL

Initialization:

All the shortest-paths algorithms start with initializing $\boldsymbol{d[v]}$ and $\boldsymbol{Pred[v]}$ **by us**ing the following procedure INITIALIZE_SIGLE_SOURCE.

INITIALIZE_SIGLE_SOURCE(V,s)

1. $\boldsymbol{for}\ each\ v \in V$
2. **do** $d[v] \leftarrow \infty$
3. $pred[v] \leftarrow NIL$
4. $d[s] \leftarrow 0$

After this initialization procedure, $d[v] = 0$ for start vertex s, and $d[v] = \infty$ for $v \in V - \{s\}$ and $pred[v] = NIL$ for all $v \in V$.

Relaxing an edge (u, v):

The SSSP algorithms are based on the technique known as ***edge relaxation***. The process of relaxing an edge (u, v) consists of testing whether we can improve (or reduce) the shortest path to v found so far (i.e $d[v]$) by going through u and taking (u, v) and, if so, update $d[v]$ and $pred[v]$. This is accomplished by the following procedure:

RELAX(u,v,w)
1. ***if*** $(\boldsymbol{d}[\boldsymbol{v}] > d[\boldsymbol{u}] + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$
2. **then** $d[v] \leftarrow \boldsymbol{d}[\boldsymbol{u}] + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$
3. $pred[v] \leftarrow u$

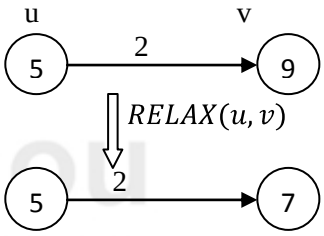


Figure (a): $\boldsymbol{d}[\boldsymbol{v}] > d[\boldsymbol{u}] + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$ **i.e.** $\boldsymbol{9} > 5 + 2$**, hence** $\boldsymbol{d}[\boldsymbol{v}] = \boldsymbol{7}$

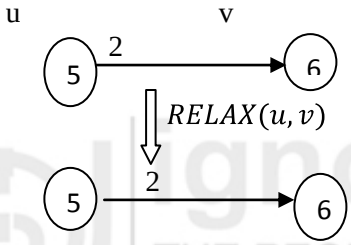


Figure (b): No change in $\boldsymbol{d}[\boldsymbol{v}]$ **since,** $\boldsymbol{d}[\boldsymbol{v}] < d[\boldsymbol{u}] + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$**.**

Note:

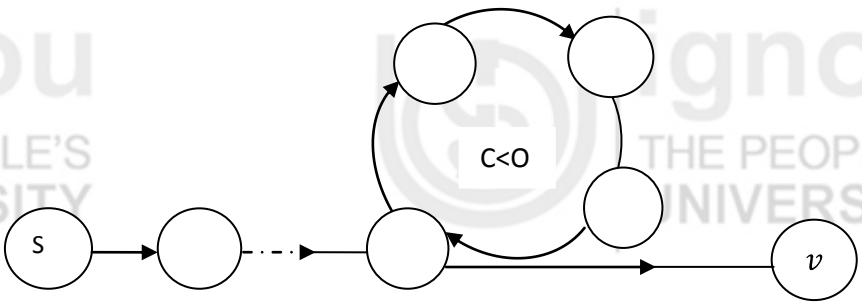
- For all the SSSP algorithm, we always start by calling **INITIALIZE_SIGLE_SOURCE(V,s)** and then relax edges.

- In **Dijkstra's algorithm** and the other shortest-path algorithm for directed acyclic graph, each edge is relaxed exactly once. In a **Bellman-Ford** algorithm, each edge is relaxed several times

1.6.1 Bellman-Ford Algorithm

- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from a source vertex $s \in V$ to every other vertex $v \in V - \{s\}$

or detect a negative weight cycles exist in G, hence ***no shortest path exist for some vertices***.



- Thus given a weighted, directed graph with $G = (V, E)$ with weight function $w: E \rightarrow R$, the Bellman-ford algorithm returns a **Boolean value** (either **TRUE** or **FALSE)** indicating whether or not there is a negative weight cycle is reachable from the source vertex. The algorithm returns a Boolean TRUE if the given graph G contains no negative weight cycle that are reachable from source vertex s, otherwise it returns Boolean FALSE.

- The algorithm uses a technique of ***relaxation*** and progressively decreases an estimate $d[v]$ on the weight of a shortest path from the source vertex s to each vertex $v \in V$ until it achieves the actual shortest path. We also maintain a predecessor value $\boldsymbol{pred}[\boldsymbol{v}]$ for all $v \in V$.

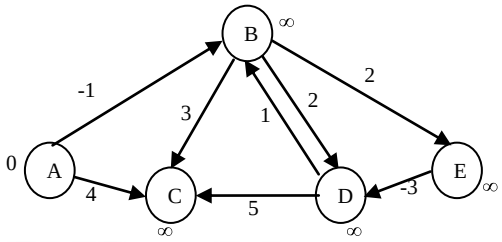
```
BELLMAN_FORD(G,w,s)

1. INITIALIZE_SIGLE_SOURCE(G,s)
2. for i ← 1 to |V| − 1
3.    do for each edge (u, v) ∈ E[G]
4.       do RELAX(u, v, w)
5.  for each edge (u, v) ∈ E[G]
6.       do  if (d[v] > d[u] + w(u, v)
7.        then return FALSE            // we detect a negative weight cycle exist
7.  return TRUE
```

Analysis of Bellman-ford algorithm

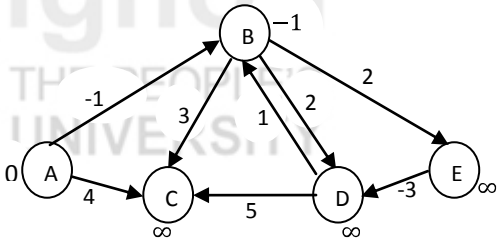
1. Line 1 for initializing $d[v]'s$, $Pred[v]'s$ and setting $v[s] = 0$ takes $\boldsymbol{O(V)}$ time.
2. For loop at line-2 executed $(|V| - 1)$ times which Relaxes all the E edges, so line 2-4 requires $(|V| - 1).O(E) = O(VE)$.
3. For loop at line 5 checks negative weight cycle for all the E edges, which requires O(E) time.

   Thus the run time of Bellman ford algorithm is $O(V + E + VE) = O(VE)$.

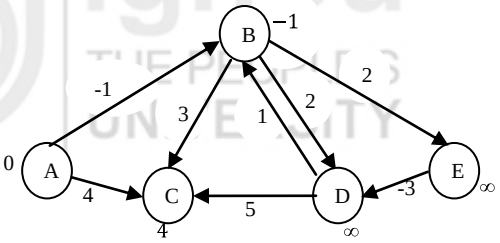
Order of edge: (B,E), (D,B), (B,D), (A,C), (D,C), (B,C), (E,D)



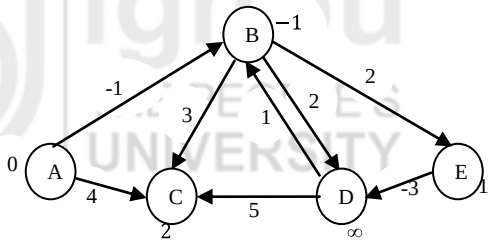
A	B	C	D	E
0	∞	∞	∞	∞



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞

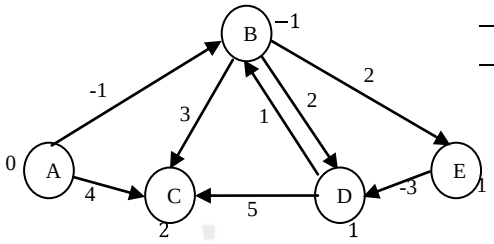


A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞

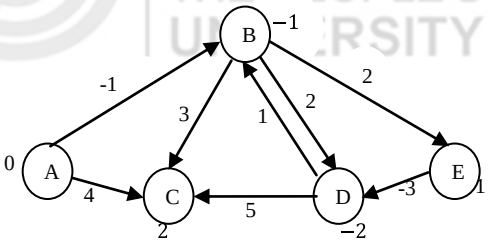


A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

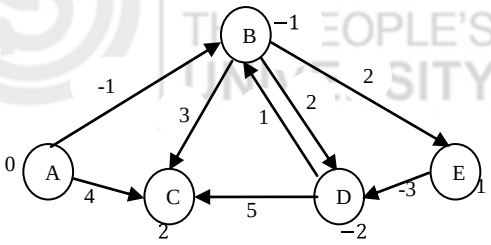
A	B	C	D	E
0	-1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

1.6.2 Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph G=(V,E) with ***non-negative edge*** weights i.e. we assume that w (u,v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. That is, all vertices $v \in S$, we have d[v] = $\delta(s, v)$. the algorithm repeatedly selects the vertex u ∈V-S with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in $V - s$ keyed by their d values. Graph G is represented by adjacency lists.

DIJKSTRA(G, w, s)

```
1       INITIALIZE-SINGLE-SOURCE(G, s)
2       S ← Ø
3       Q ← V[G]
4       while Q ≠ Ø
5             do u ← EXTRACT-MIN(Q)
6                S ← S ∪{u}
7                for each vertex v ∈ Adj[u]
8                do RELAX(u, v, w)
```

Because Dijkstra's algorithm always choose the "lightest" or "closest" vertex in V-S to insert into set S, we say that it uses a greedy strategy.

Dijkstra's algorithm bears some similarly to both breadth-first search and Prim's algorithm for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth- first distances.
Dijkstra's algorithm is like prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Analysis of Dijkstra's algorithm

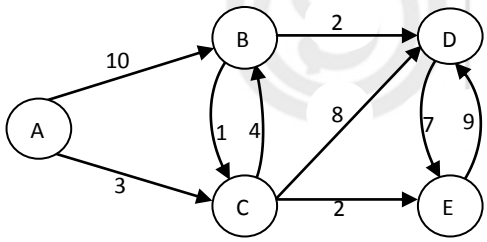
The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as function of |E| and |V| using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q.

in this case, the running time is $O(|V|^2 + |E|) = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently, storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires O((|E| + |V|) time (which is dominated by O|E|log|V|) assuming every vertex is connected, and the Fibonacci heap improves this to O|E| + |V|log|V|).

Example1:

Apply Dijkstra's algorithm to find shortest path from source vertex A to each of the other vertices of the following directed graph.

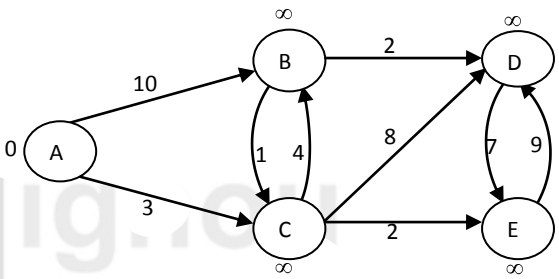


Solution:

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. The algorithm repeatedly selects the

vertex u ∈V-S with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in $V - s$ keyed by their d values.

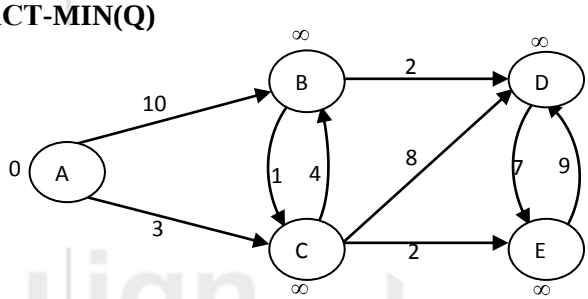
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S={}

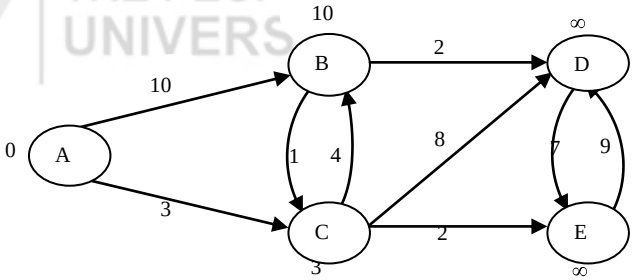
"A" ← EXTRACT-MIN(Q)



Q:	**A**	**B**	**C**	**D**	**E**
	0	∞	∞	∞	∞

S:{A}

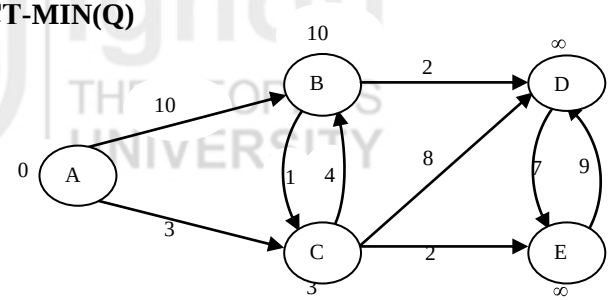
Relax all edges leaving A:



Q:	**A**	**B**	**C**	**D**	**E**
	0	∞	∞	∞	∞
		10	**3**	**-**	**-**

S:{A}

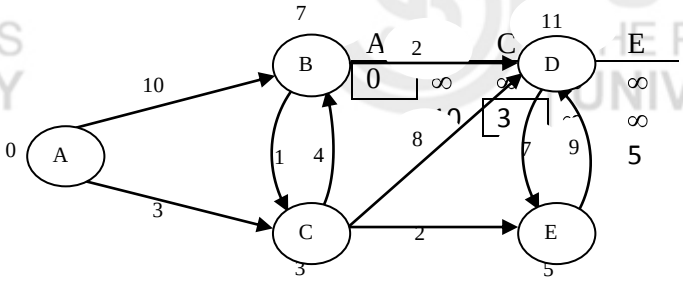
"C" ← EXTRACT-MIN(Q)



Q:	**A**	**B**	**C**	**D**	**E**
	0	∞	∞	∞	∞
		10	**3**	**-**	**-**

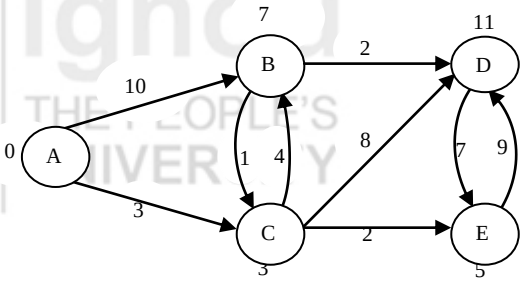
S:{A,C}

Relax all edges leaving C:



S:{A,C}

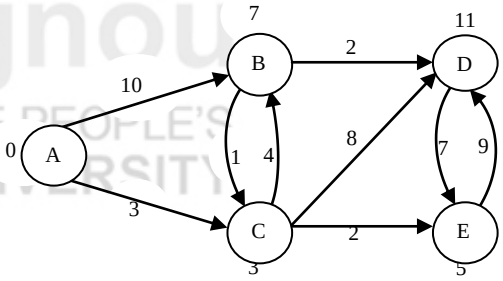
"E"← EXTRAXT-MIN(Q)



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	**3**	-	-
		7		**11**	**5**

S:{A,C,E}

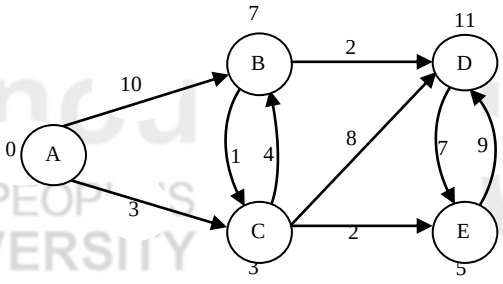
Relax all edges leaving E:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S:{A,C,E}

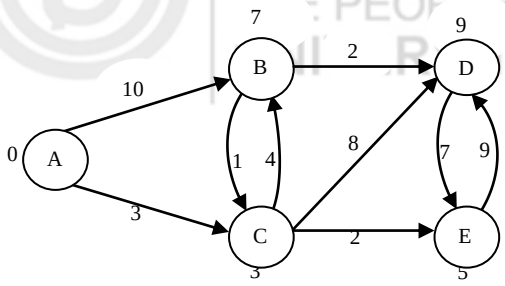
"B" ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S:{A,C,E,B}

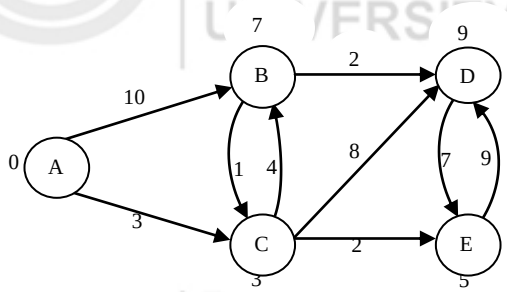
Relax all edges leaving B:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S:{A,C,E,B}

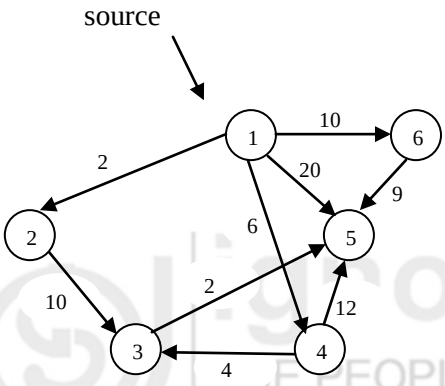
"D" ←EXTRACT-MIN(Q):



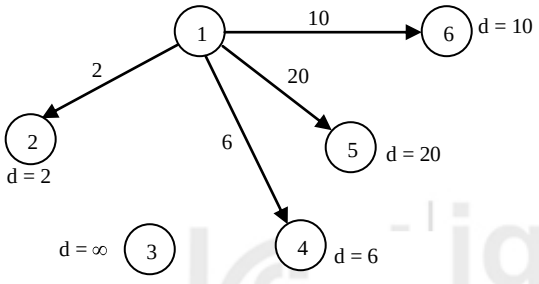
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S:{A,C,E,B,D}

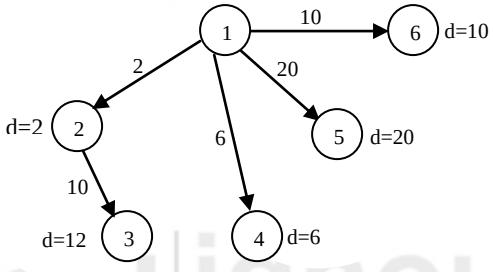
Example2: Apply dijkstra's algorithm on the following digraph (1 is starting vertex)



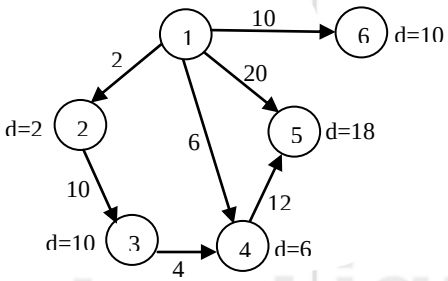
S={ }



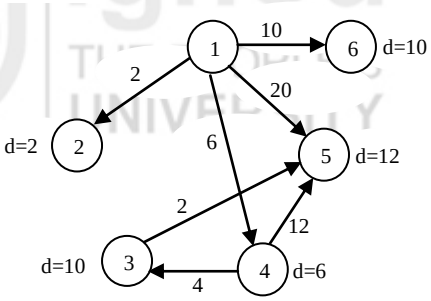
(initial) S={1}



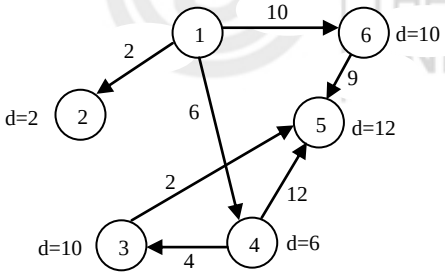
(1) S= {1, 2}



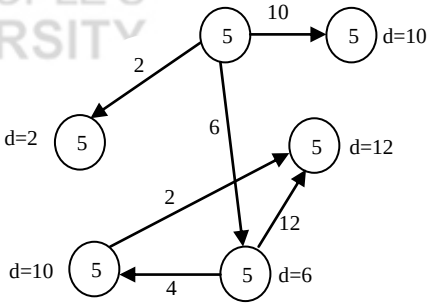
(2) S={1, 2, 4}



(3) S= {1, 2, 4, 3}



(4) S= {1, 2, 4, 3, 6}



(5) S={1, 2, 4, 3, 6, 5}

Figure 2: Stages of Dijkstra's algorithm

Iterations	S	Q (Priority Queue) d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	EXTRACT_ MIN(Q)
Initial	{ }	0	∞	∞	∞	∞	∞	1
1	(1}	**[0]**	2	∞	6	20	10	2
2	{1,2}		[2]	**12**	**6**	**20**	**10**	4
3	(1,2,4}			10	[6]	18	10	3
4	{1,2,4,3}			**[10]**		**18**	**10**	6
5	{1,2,4,3,6}					**12**	**[10]**	5
	{1,2,4,3,6,5}					**[12]**		

Table2: Computation of Dijkstra's algorithm on digraph of Figure 2

☞ Check Your Progress 2

Choose correct option for Q.1 to Q.5

Q.1: Dijkstra's algorithm running time, where n is the number of nodes in a graph is:

a) $O(n^2)$ b) $O(n^3)$ c) $O(n)$ d) $O(n \log n)$

Q2: This of the following algorithm allows negative edge weight in a graph to find shortest path?

a) Dijkstra's algorithm b) Bellman-ford algorithm c) Kruskal algo. d) Prim's Algo

Q3: The running time of Bellman-ford algorithm is

a) $O(|E|^2)$ b) $O(|V|^2)$ c) $O(|E|\log|V|)$ d) $O(|E||V|)$

Q.4: Consider a weighted undirected graph with positive edge weights and let (u,v) be an edge in the graph. It is known that the shortest path from source vertex s to u has weight 60 and the shortest path from source vertex s to v has weight 75. which statement is always true?

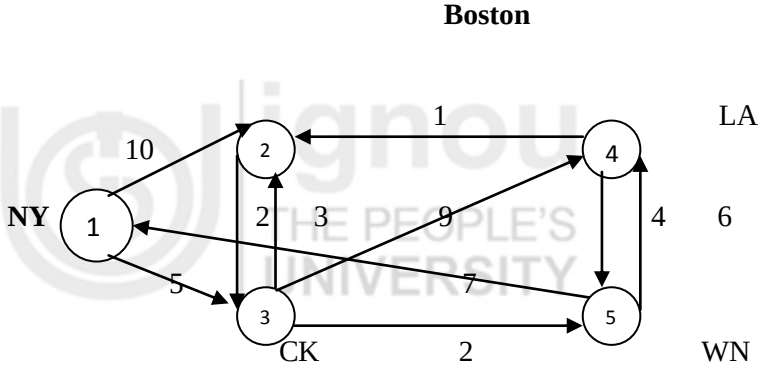
a) weight $(u,v) \le 15$ b) weight $(u,v) = 15$ c) weight $(u,v) \ge 15$ d) weight $(u,v) > 15$

Q.5: Which data structure is used to maintained the distance of each node in a Dijkstras's algorithm.

a) Stack b) Queue c) Priority Queue d) Tree

Q.6: Differentiate between Bellman-ford and Dijkstra's algorithm to find a shortest path in a graph?

Q.7: Find the minimum distance of each station from New York (**NY**) using Dijkstra's algorithm. Show all the steps.



Q.8: Analyze the running time of the Dijkstra's algorithm?

1.7 SUMMARY

- Greedy algorithms are typically 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 w. r. t. some constraints or conditions.
- Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it ***solution set***, S where S$\subseteq C$) that satisfies the given constraints or conditions. Any subset S $\subseteq C$, which satisfies the given constraints, is called a ***feasible*** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called a **optimal solution**.
- Greedy algorithm always makes the choice that looks best at the moment. That is, it makes a ***locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.***
- Greedy algorithm does not always yield an optimal solution; but for many problems they do.
- The (fractional) Knapsack problem is to fill a knapsack or bag (up to its maximum capacity M) with the given $n\ items$, which maximizes the total profit earned.
- Let G=(V,E) be an undirected connected graph. A ***subgraph*** T=(V,E') of G is a ***spanning tree*** of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.
- A *complete graph* (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if n=5 then total number of spanning tree is 125.
- A ***minimum cost spanning tree*** (MCST) of a weighted connected graph G is that spanning tree whose sum of length (or weight) of all its edges is minimum, among all the possible spanning tree of G.
- There are two algorithm to find a MCST of a given directed graph G, namely Kruskal's algorithm and Prim's algorithm.
- The basic difference between Kruskal's and Prim's algorithm is that in kruskal's algorithm it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus At intermediate step of algorithm, there are may be more than one connected components of trees are possible. But in case of Prim's algorithm it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus at intermediate step of algorithm, there will be only one connected components are possible.
    - Kruskal's algorithm runs in $O(|E|log|V|)$ time and Prim's algorithm runs in time (n^2) , where n is the number of nodes in the graph.

- ***Single-source-shortest path problem (SSSPP) problem*** is to find a shortest path from source vertex $s \in V$ to every other vertex $v \in V - s$ in a given graph $G = (V, E)$

- To find a SSSP for directed graphs $G(V, E)$, we have two different algorithms: ***Bellman-Ford algorithm*** and ***Dijkstra's algorithm***

- ***Bellman-ford algorithm***, allow ***negative weight edges*** also in the input graph where as **Dijkstra's algorithm** allows only ***positive weight edges*** in the input graph.

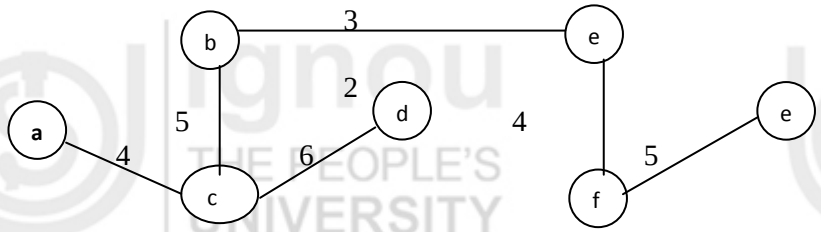
- ***Bellman-ford algorithm*** runs in time $O(|V||E|)$ **whereas Dijkstra's algorithm runs in time $O(n^2)$.**

1.8 SOLUTIONS/ANSWERS

Check your Progress 1:

1-d, 2-c, 3-b, 4-c, 5-c, 6-b, 7-d

Solution 8:



Total minimum cost of given graph G = $3 + 4 + 5 + 6 + 4 + 3 = 27$

Solution 9:

(a) FALSE, since edge with the smallest weight will be part of every minimum spanning tree.
(b) **TRUE:** edge with the smallest weight will be part of every minimum spanning tree.
(c) **TRUE:**
(d) **TRUE:** Since more than one edges in a Graph may have the same weight.
(e) **TRUE**: In a connected weighted graph in which edge weights are not all distinct, then the graph must have more than one spanning tree but the minimum cost of those spanning tree will be same.

Solution 10:

A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say ***select***). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. When a greedy algorithm works correctly, the first solution found in this way

is always optimal. In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say x, from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{x\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.

A general form for greedy technique can be illustrated as:

```
Algorithm Greedy(C, n)
   /* Input: A input domain (or Candidate set ) C of size n, from which solution is to be
            Obtained. */
    // function select (C: candidate_set)  return an element (or candidate).
   // function solution (S: candidate_set) return Boolean
   // function feasible (S: candidate_set) return Boolean
    /* Output: A solution set S, where S ⊆ C, which maximize or minimize the selection
              criteria w. r. t. given constraints */
   {

          S ← ϕ                          // Initially a solution set S is empty.
        While ( not solution(S) and C ≠ ϕ)
          {
            x ← select(C)              /* A "best" element x is selected from C which
                                          maximize or minimize the selection criteria. */
            C ← C − {x}                    /* once x is selected , it is removed from C
            if ( feasible(S ∪ {x}))  then  /* x is now checked for feasibility
                S ← S ∪ {x}
          }
        If (solution (S))
            return S;
         else
           return " No Solution"
     } // end of while
```

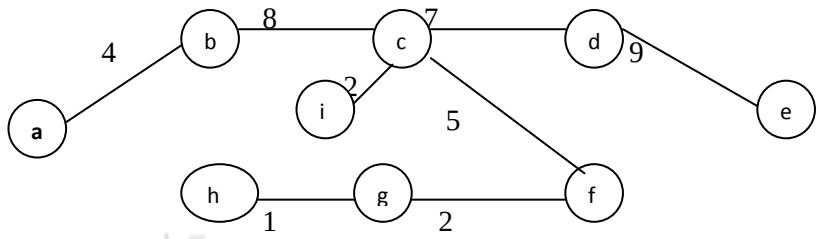
Solution11:

A main ***difference*** between ***kruskal's*** and ***Prim's*** algorithm to solve MCST problem is that the order in which the edges are selected.

Kruskal's Algorithm	**Prim's algorithm**
• Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST. • In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there are may be more than one connected components are possible. • Time complexity: $O(\|E\| log \|V\|)$	• Prim's algorithm always selects a vertex (say, v) to find MCST. • In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there will be only one connected components are possible • Time complexity: $O(\|V\|^2)$

Solution 12: No, spanning tree of a graph is not unique in general, because more than one edges of a graph may have the same weight.

For complete solution refer Q.2 , given in this booklet.



Total minimum cost of the spanning tree =$4 + 8 + 2 + 1 + 2 + 5 + 7 + 9 = 38$

Solution 13:

Given n=5, M=12,

$(P_1, P_2, ..., P_5) = (12,32,40,30,50)$

$(W_1, W_2, ..., W_5) = (4,8,2,6,1)$

$$\left(\frac{P_1}{w_1}, \frac{P_2}{w_2}, \dots, \frac{P_5}{w_5}\right) = (3,4,20,5,50)$$

Thus the item which has maximum P_i/w_i value will be placed into a knapsack first, That is 5th item first, then 3rd item then 4th item then 2nd and then 1st item (if capacity of knapsack is remaining). The following table shows a solution of this knapsack problem.

S.No	Solution Set $(x_1, x_2, x_3, , x_4, x_5)$	$\sum_{i=1}^{5} w_i x_i$	$\sum_{i=1}^{5} p_i x_i$
1	$(\frac{1}{2}, 1,1,1,1)$	$1 + 2 + 6 + 1 + 4 \times \frac{1}{2} = 12$	158

Solution 14:

Given n=5, M=18,

$(P_1, P_2, ..., P_7) = (12,10,8,11,14,7,9)$

$(W_1, W_2, ..., W_7) = (4,6,5,7,3,1,6)$.

$$\left(\frac{P_1}{w_1}, \frac{P_2}{w_2}, \dots, \frac{P_7}{w_7}\right) = (3.0, 1.67, 1.60, 1.57, 4.67, 7.0, 1.50)$$

The item which has maximum P_i/w_i value will be placed into a knapsack first. Thus the sequence of items placed into a knapsack is: 6th , 5th ,1st ,2nd, 3rd , 4th and then 7th item. The following table shows a solution of this knapsack problem.

S.No	Solution Set $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
1	$(1,1,\frac{4}{5}, 0,1,1,0)$	$1 + 3 + 4 + 6 + 5 \times \frac{4}{5} = 18$	49.4

Check Your Progress 2

1-a, 2-b, 3-d, 4-a, 5-c

Solution 6:

- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$ or detect a negative weight cycles in G, hence ***no solution***. If there is no negative weight cycles are reachable (or exist) from source vertex s, then we can find a shortest path form source vertex $s \in V$ to every other vertex $v \in V$. If there exist a negative weight cycles in the input graph, then the algorithm can detect it, and hence "No solution".

- **Dijkstra's algorithm** allows only positive ***weight edges*** in the input graph and finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$.

Solution 7:

Following Table summarizes the Computation of Dijkstra's algorithm for the given digraph of Question 7.

Iterations	**S**	**Q (Priority Queue)**					**EXTRACT _MIN(Q)**
		d[1]	**d[2]**	**d[3]**	**d[4]**	**d[5]**	
Initial	{ }	0	∞	∞	∞	∞	1
1	(1}	**[0]**	10	5	∞	∞	3
2	{1,3}		8	**[5]**	14	7	2
3	(1,3,2}		**[8]**		14	7	5
4	{1,3,2,5}				13	[7]	4
5	{1,3,2,5,4)				**[13]**		

Table1: Computation of Dijkstra's algorithm on digraph of question 7

Solution 8: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as function of |E| and |V| using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q. in this case, the running time is $O(|V|^2 + |E|) = O(V^2)$.

3.9 FURTHER READING

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).

UNIT 2 DIVIDE AND CONQUER APPROACH

Structure **Page Nos.**

2.0 INTRODUCTION

We have already mentioned in unit-1 of Block-1 that there are five fundamental techniques which are used to design the Algorithm efficiently. These are: Divide and Conquer, Greedy Method, Dynamic Programming, Backtracking and Branch and Bound. Out of these techniques Divide & Conquer is probably the most well-known.

Many useful algorithms are recursive in nature. To solve a given problem, they call themselves recursively one or more times. These algorithms typically follow a divide & Conquer approach. A divide & Conquer method works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough (i.e. smaller in size w.r.t. original problem) to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The following figure-1 show a typical Divide & Conquer Approach

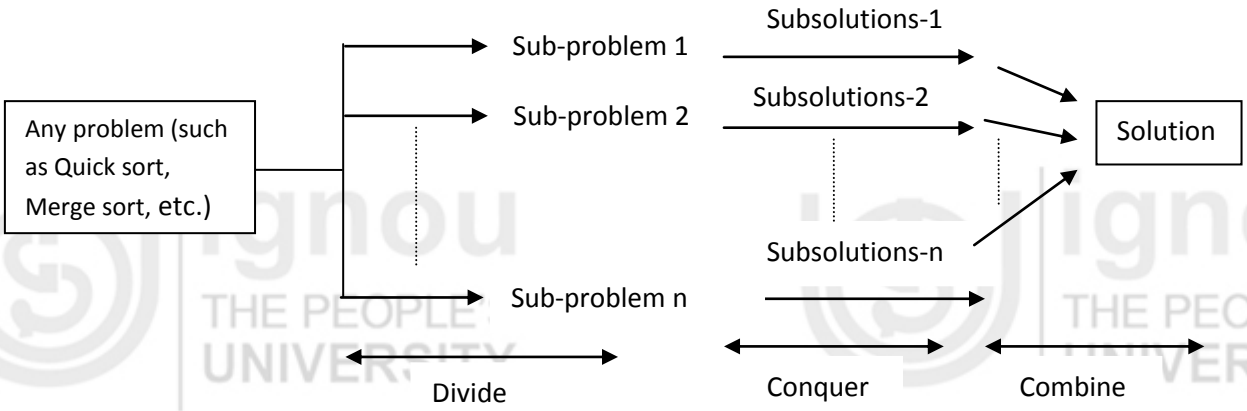


Figure1: Steps in a divide and computer technique

Thus, in general, a divide and Conquer technique involves 3 Steps at each level of recursion:

Step 1: Divide the given big problem into a number of sub-problems that are similar to the original problem but smaller in size. A sub-problem may be further divided into its sub-problems. A Boundary stage arrives when either a direct solution of a sub-problem at some stage is available or it is not further sub-divided. When no further sub-division is possible, we have a direct solution for the sub-problem.

Step 2: **Conquer** (Solve) each solutions of each sub-problem (independently) by recursive calls; and then

Step 3: **Combine** the solutions of each sub-problems to generate the solutions of original problem.

In this unit we will solve the problems such as Binary Search, Searching - QuickSort, MergeSort, integer multiplication etc by using Divide and Conquer method;

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concept of Divide-and-Conquer;
- Explain how Divide-and-Conquer method is applied to solve various problems such as Binary Search, Quick-Sort, Merge-Sort, Integer multiplication etc., and
- Write a general recurrence for problems that is solved by Divide-and-Conquer.

2.2 GENERAL ISSUES IN DIVIDE AND CONQUER

Many useful algorithms are recursive in structure, they make a recursive call to itself until a base (or boundary) condition of a problem is not reached. These algorithms closely follow the **Divide and Conquer** approach.

To analyzing the running time of divide-and-conquer algorithms, we use a **recurrence equation** (more commonly, a **recurrence**). A recurrence for the running time of a divide-and-conquer algorithm is based on the 3 steps of the basic paradigm.

1) **Divide**: The given problem is divided into a number of sub-problems.

2) **Conquer**: Solve each sub-problem be calling them recursively.
(**Base case:** If the sub-problem sizes are small enough, just solve the sub-problem in a straight forward or direct manner).

3) **Combine**: Finally, we combine the sub-solutions of each sub-problem (obtained in step-2) to get the solution to original problem.

Thus any algorithms which follow the divide-and-conquer strategy have the following recurrence form:

$$T(n) = \begin{cases} \theta(1) & if\ n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & Otherwise \end{cases}$$

Where

- $T(n) = running\ time\ of\ a\ problem\ of\ size\ n$
- If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case. The brute-force (or direct) solution takes constant time: $\Theta(1)$
- Otherwise, suppose that we divide into a sub-problems, each $1/b$ of the size of the original problem of size n.
- Suppose each sub-problem of size **n/b** takes $T(\frac{n}{b})$ time to solve and since there are a sub-problems so we spend $\boldsymbol{aT}(\frac{\boldsymbol{n}}{\boldsymbol{b}})$ total time to solve $\boldsymbol{a}$ sub-problems.
- $D(n)$ is the cost(or time) of dividing the problem of size n.
- $C(n)$ is the cost (or time) to combine the sub-solutions.

Thus in general, an algorithm which follow the divide and conquer strategy have the following recurrence:

$$T(n) = \boldsymbol{a}T\left(\frac{n}{b}\right) + f(n)$$

Where

- T(n) = running time of a problem of size n
- a means "In how many part the problem is divided"
- $T\left(\frac{n}{b}\right)$ means "Time required to solve a sub-problem each of size (n/b)"
- D(n) + C(n) = f(n) is the summation of the time requires to divide the problem and combine the sub-solutions.

  (Note: For some problem C(n)=0, such as Quick Sort)

Example: Merge Sort algorithm closely follows the Divide-and-Conquer approach. The following procedure MERGE_SORT (A, p, r) sorts the elements in the subarray A [p ,…, r]. If p ≥ r, the subarray has at most one element and is therefore already sorted. Otherwise, the Divide step is simply computer an index q that partitions A [p, … , r] into two sub-arrays: A [p, …, q] containing ceil (n/2) elements, and A[q+1, …, r] containing floor (n/2) elements.

```
MERGE_SORT (A, p, r)

1. if (p < r)
2.    then q ← [(p + r)/2]          /* Divide
3.       MERGE_SORT (A, r, q)       /* Conquer
4.       MERGE_SORT (A, q + 1, r)   /* Conquer
5.       MERGE (A, p, q, r)         /* Combine
```

Figure 2: Steps in merge sort algorithms

To set up a recurrence T(n) for MERGE SORT algorithm, we can note down the following points:

- **Base Case**: MERGE SORT on just one element (n=1) takes constant time i.e. $\Theta(1)$

- When we have n > 1 elements, we can find a running time as follows:

  (1) **Divide**: Just compute q as the middle of p and r, which takes constant time. Thus
  $D(n) = \Theta(1)$

  (2) **Conquer**: We recursively solve two sub-problems, each of size n/2, which contributes
  $2T\left(\frac{n}{2}\right)$ to the running time.

  (3) **Combine**: Merging two sorted subarrays (for which we use MERGE (A, p, r) of an n-element array) takes time $\Theta(n)$, so $C(n) = \Theta(n)$.
  Thus $f(n) = D(n) + c(n) = \Theta(1) + \Theta(n) = \Theta(n)$, which is a linear function of n.

Thus from all the above 3 steps, a recurrence relation for MERGE_SORT (A, 1, n) in the **worst case** can be written as:

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

Now after solving this recurrence by using any method such as Recursion-tree or Master Method (as given in UNIT-1), we have $T(n) = \Theta(nlogn)$.
This algorithms will be explained in detailed in section 2.4.2

2.3 BINARY SEARCH

Search is the process of finding the position (or location) of a given element (say x) in the linear array. The search is said to be successful if the given element is found in the array otherwise it is considered unsuccessful.

A Binary search algorithm is a 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. The Binary search algorithm proceeds as follow:

(1) Begin with the interval covering the whole array; binary search repeatedly divides the search interval by half.

(2) At each step, the algorithm compares the input key (or search) value x with the key value of the middle element of the array A.

(3) If it matches, then a searching element x has been found, so then 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 or the interval is empty, which indicates x is "not found".

```
BinarySearch_Iterative(A[1…n],n,x)

/* Input: A sorted (ascending) linear array of size n.

Output: This algorithm find the location of the search element x in linear array A. If
search ends in success, it returns the index of the searched element x, otherwise returns
-1 indicating x is "not found". Here variable low and high is used to keep track of the
first element and last element of the array to be searched, and variable mid is used as
index of the middle element of the array under consideration. */
{
    low=1
    high=n
    while(low<=high)
     {
       mid= (low+high)/2
       if(A[mid]==x]
          return mid;                    // x is found
       else if(x<A[mid])
          high=mid-1;
       else  low=mid+1;
     }
    return -1                           // x is not found
  }
```

Figure 3: Binary search algorithms

Analysis of Binary search:

Method1:

Let us assume for the moment that the size of the array is a power of 2, say 2^k. Each time in the while loop, when we examine the middle element, we cut the size of the sub-array into 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 after 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 $T(n) = O(logn)$

Method 2:

Binary search closely follow the Divide-and-conquer technique.

We know that any problem, which is solved by using Divide-and-Conquer having a recurrence of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Since at each iteration, the array is divided into two sub-arrays but we are solving only one sub-array in the next iteration. So value of a=1 and b=2 and f(n)=k where k is a constant less than n.
Thus a recurrence for a binary search can be written as

$T(n) = T\left(\frac{n}{2}\right) + k$; by solving this recurrence using substitution method, we have:

$k + k + k + \cdots \ldots \ldots \ldots up\ to\ (logn) terms = k.logn = O(logn)$.

Example1: consider the following sorted array DATA with 13 elements:

11	22	30	33	40	44	55	60	66	77	80	88	99

Illustrate the working of binary search technique, while searching an element (say ITEM)

(i) 40 (ii) 85

Solution

We apply the binary search to DATA[1,…13] for different values of ITEM.

(a) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in Fig.1, where the values of DATA[Low] and DATA[High] in each stage of the algorithm are indicated by circles and the value of DATA[MID] by a square. Specifically, Low, High and MID will have the following successive values:

1. Initially, Low = 1 and High = 13, Hence
   MID = $\lfloor (1 + 13)/2 \rfloor = 7$ and so DATA[MID] = 55

2. Since 40 < 55, High has its value changed by High = MID – 1 = 6.
   Hence MID = $\lfloor (1 + 6)/2 \rfloor = 3$ and so DATA[MID] = 30

3. Since 40 < 30, Low has its value changed by Low = MID – 1 = 4.
   Hence MID = $\lfloor (4 + 6)/2 \rfloor = 5$ and so DATA[MID] = 40

We have found ITEM in location LOC = MID = 5.

(1) (11,) 22, 30, 33, 40, 44, [55,] 60, 66, 77, 80, 88, (99,)

(2) (11,) 22, (30,) 33, 40, [44,] 55, 60, 66, 77, 80, 88, 99

(3) 11, 22, 30, (33,) [40,] [44,] 55, 60, 66, 77, 80, 88, 99 [Successful]

Figure 4: Binary search for ITEM = 40

(b) Suppose ITEM = 85. The binary search for ITEM is pictured in Figure 2. Here Low, High and MID will have the following successive values:

1. Again initially, Low = 1, High = 13, MID = 7 and DATA[MID] = 55.
2. Since 85 > 55, Low has its value changed by Low = MID + 1 = 8. Hence MID = $\lfloor (8 + 13)/2 \rfloor = 10$ and so DATA[MID] = 77
3. Since 85 > 77, Low has its value changed by Low = MID + 1 = 11. Hence MID = $\lfloor (11 + 13)/2 \rfloor = 12$ and so DATA[MID] = 88
4. Since 85 > 88, High has its value changed by High = MID – 1 = 11. Hence

MID = $\lfloor (11 + 11)/2 \rfloor = 11$ and so DATA[MID] = 80

(Observe that now Low = High = MID = 11.)

Since 85 > 80, Low has its value changed by Low = MID + 1 = 12. But now Low > High, Hence ITEM does not belong to DATA.

(1) (11,) 22, 30, 33, 40, 44 [55,] 60, 66, 77, 80, 88 (99,)

(2) 11, 22, 30, 33, 40, 44, 55, (60,) 66, [77,] 80, 88, (99)

(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80) 88, 99 [unsuccessful]

Figure 5: Binary search for ITEM = 85

Example 2: Suppose an array DATA contains 1000000 elements. How many comparisons are required (in worst case) to search an element (say ITEM) using Binary search algorithm.

Solution: Observe that

$2^{10} = 1024 > 1000$ and hence $2^{20} > 1000^2 = 1000000$

Using the binary search algorithm, one requires only about **20 comparisons** to find the location of an ITEM in an array DATA with 1000000 elements, since $log_2 1000000 \approx log_2 2^{20} = 20$

☞ **Check Your Progress 1**

(Objective questions)

1. What are the three sequential steps of divide-and-conquer algorithms?
   (a) Combine-Conquer-Divide
   (b) Divide-Combine-Conquer
   (c) Divide-Conquer-Combine
   (d) Conquer-Divide-Conquer

2. Binary search executes in _____ time.
   (a) $O(n)$ (b) $O(\log n)$ (c) $O(n \log n)$ (d) $O(n^2)$

3. The recurrence relation that arises in relation with the complexity of binary search is
   (where k is a constant)
   (a) $T(n) = T(n/2) + k$ (b) $T(n) = 2T(n/2) + k$
   $(c)\ T(n) = 2T(n/2) + \log(n)$ (d) $T(n) = T(n/2) + n$

4. Suppose an array A contains n=1000 elements. The number of comparisons required (in worst case) to search an element (say x) using binary search algorithm:
   a) 100 b) 9 c)10 d) 999

5. Consider the following sorted array A with 13 elements

7	14	17	25	30	48	56	75	87	94	98	115	200

   Illustrate the working of binary search algorithm, while searching for ITEM
   (i) 17 (ii) 118

6. Analyze the running time of binary search algorithm in best average and worst cases.

2.4 SORTING

Sorting is the process of arranging the given array of elements in either increasing or decreasing order.

Input: A sequence of n number $< a_1, a_2, \ldots \ldots, a_n >$

Output: A permutation (reordering) $< a'_1, a'_2, \ldots \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \ldots \leq a'_n$.

In this Unit we discuss the 2 sorting algorithm: Merge-Sort and Quick-Sort

2.4.1 MERGE-SORT

Merge Sort algorithm closely follows the Divide-and-conquer strategy. Merge sort on an input array A [1… n] with n-elements (n > 1) consists of (3) Steps:

Divide: Divide the n-element sequence into two sequences of length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ (say $A_1 = \langle A(1), (2), \ldots A[\lceil n/2 \rceil] \rangle$ and $A_2 \langle A[\lceil n/2 \rceil + 1], \ldots A[n])$

Conquer: Sort these two subsequences A_1 and A_2 recursively using MERGE SORT; and then

Combine: Merge the two sorted subsequences A_1 and A_2 to produce a single sorted subsequence.

The following figure shows the idea behind merge-sort:

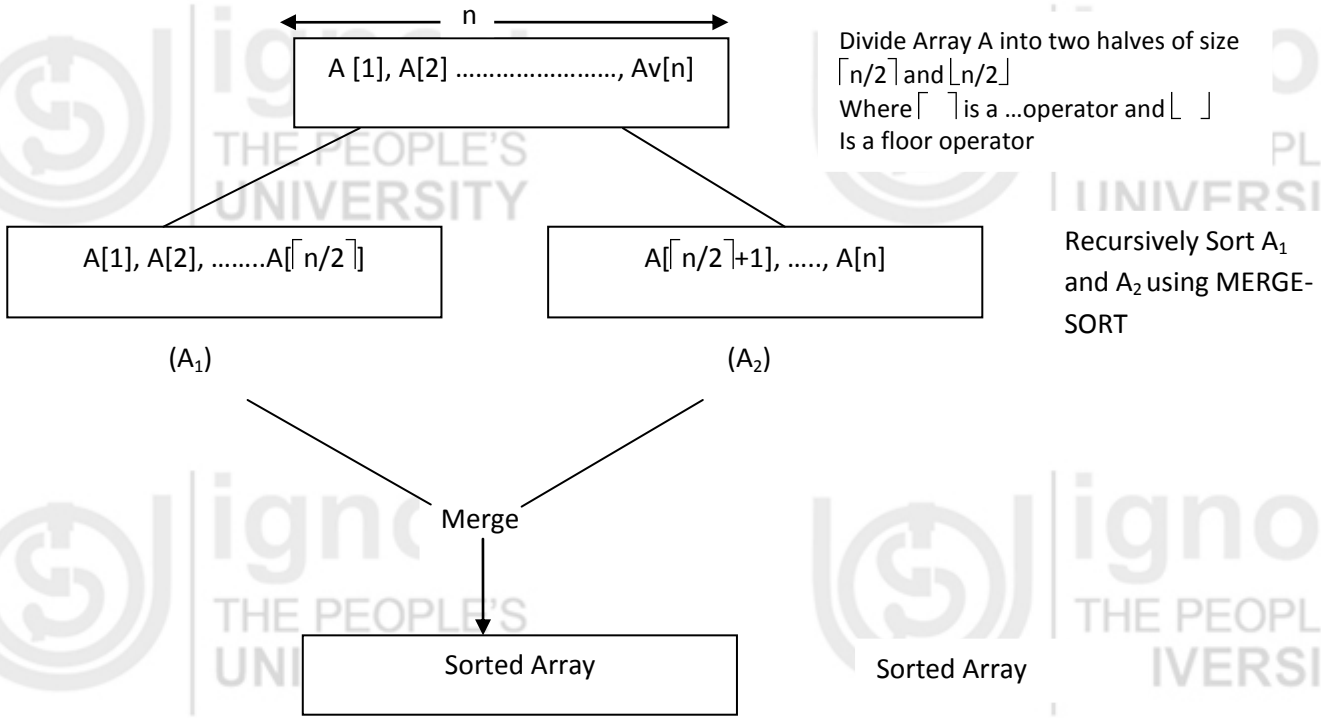


Figure 6: Merge sort

Figure 6: Illustrate the operation of two- way merge sort algorithm. We assume to sort the given array A [1…n] into ascending order. We divide the given array A [1..n] into 2 subarrays: A[1, … ⌈n/2⌉] and [⌈n/2⌉+1, …. n]. Each subarray is individually sorted, and the resulting sorted subarrays are merged to produce a single sorted array of n- elements.

For example, consider an array of 9 elements :{ 80, 45 15, 95, 55, 98, 60, 20, 70} The MERGE-SORT algorithm divides the array into subarrays and merges them into sorted subarrays by MERGE () algorithm as illustrated in arrows the (dashed line arrows indicate the process of splitting and regular arrows the merging process).

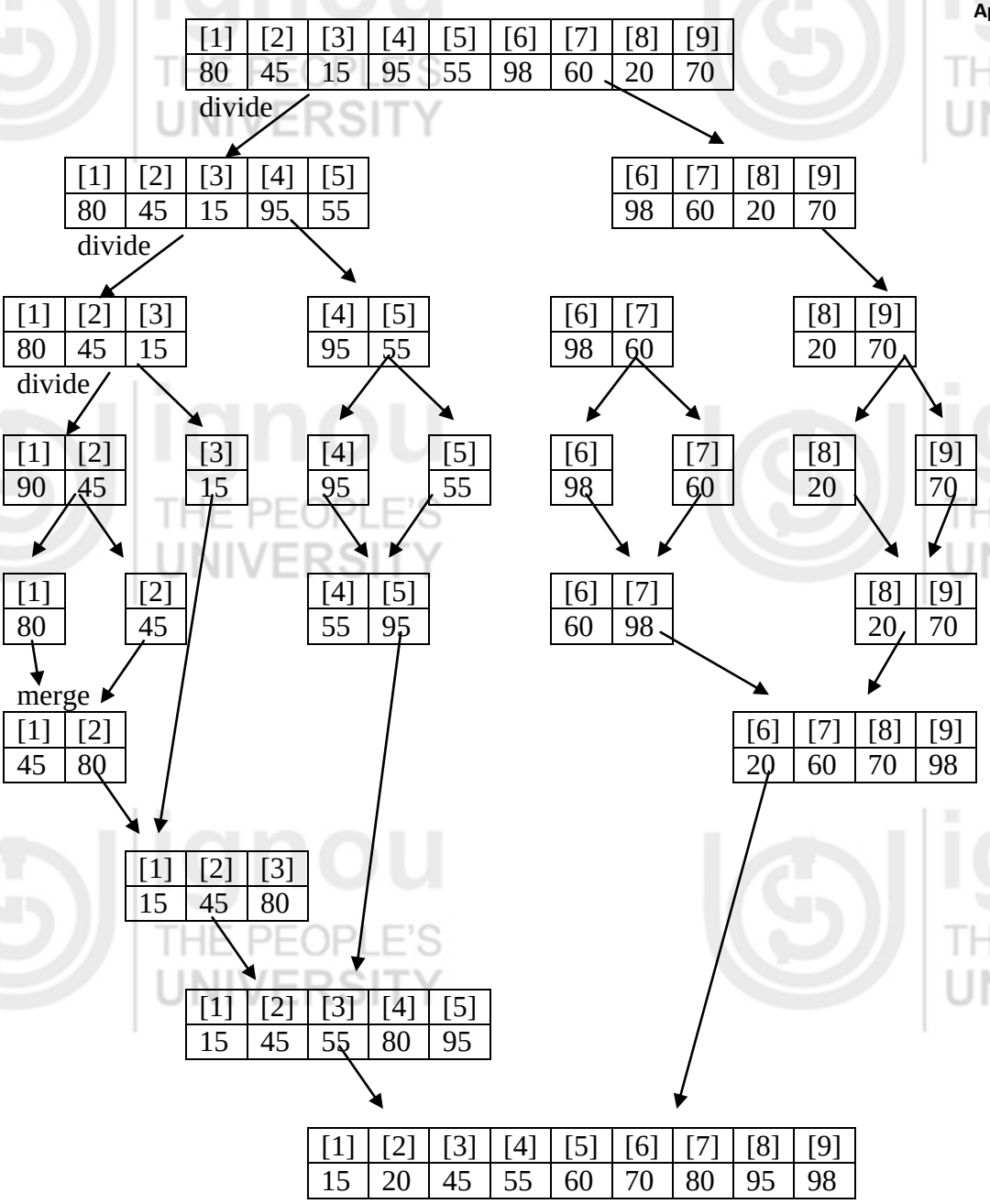


Figure 7: Two-way Merge Sort

From figure 7, we can note down the following points:

- 1st, left half of the array with 5-elements is being split and merge; and next second half of the array with 4-elements is processed.

- Note that splitting process continues until subarrays containing a single element are produced (because it is trivially sorted).

Since, here we are always dealing with sub-problems, we state each sub-problem as sorting a subarray A [p…r]. Initially p = 1 and r = n, but these values changes as we recurse through sub-problems.

Thus, to sort the subarray A[p…r]
1) **Divide**: Partition A [p…r] into two subarrays A [p…q] and A [q+1…r], where q is the half way point of A [p…r].

2) **Conquer**: Recursively Sort the two subarrays A [p…q] and A [q+1..r].

3) **Combine**: Merge the two sorted subarray A [p..q] and A[q+1..r] to produce a single sorted subarray A[p..r]. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion stops, when the subarray has just 1 element, so that it is trivially sorted.

Algorithm:

*This algorithm is for sorting the elements using Merge Sort.
Input: An array A [p..r] of unsorded elements, where p is a beginning element of an array and r as end element of array A.

Output: Sorted Array A [p..r].

Merge-Sort (A, P,r)

```
    {
1.      if (p <  r)                          /* Check for base case
                                          {
2.      q ← ⌊(p + r)/2⌋                      /* Divide step
3.      Merge-Sort (A,p,q)        /* Conquer step
4.      MERGE-SORT (A,q+1,r)      /* Conquer step
5.      MERGE (A,p,q,r)           /* Combine step
        }
    }
```

Intial Call is MERGE-SORT (A,1,n)

Next, we define Merge (A, p, q, r), which is called by the Algorithm MERGE-SORT (A, p, q).

Merging

Input: Array A and indices p,q, r s.t.

- $P \leq q < r$
- Subarray A [p..r] is sorted and subarray A[q+1..r] is sorted. By the restrictions on p, q,r, neither subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in A [p..r]

Idea behind linear-time merging:
Think of two piles of cards:

- Each pile is sorted and placed face-up on a table with the smallest cards on top.
- We will merge these into a single sorted pile, face down on the table.
- Basic step:
  → Choose the smaller of the two top cards.
  → Remove it from its pile, thereby exposing a new top card.
- Repeatedly perform basic steps until one input pile is empty.
- Once one input pile is empty, just take the remaining input pile and place it face down into the output pile.
- We put a special sentinel card and on the bottom of each input pile; when either of the input pile hits ∞ first, means all the non sentinel cards of that pile have already been placed into the output pile.
- Each basic step should take constant time, since we check just two top cards.
- There are $\leq$ n basic steps, since each basic step removes one card from the input piles and we started with n cards in the input piles.
- Therefore this procedure should take *O*(n) time.

Figure 8: Merging Steps

The following Algorithm merge the two sorted subarray A [p.. r] and A [q+1..r] into one sorted output subarray A [p..r].

Merge (A, p, q, r)

```
1. n1 ← q – p + 1                    // No. of elements in sorted subarray A [p..q]
2. n2 ← r – q                        // No. of elements in sorted subarray A[q+1 .. r]
   Create arrays L [1.. n1 +1] and R [1..n1 +1]
3. for i ← 1 to n1
4. do L[i] ← A [p + I -1]            // copy all the elements of A [p..r] into L [1 .. n1]
5. for j ← 1 to n1
6.       do R [j] ← A [q + j]        // copy all the elements of A [q + 1, .. r] into
                                        R[1..n2]
7. L [n1 +1] ← ∞

8. R [n2 + 1] ← ∞
9. i ← 1
10. j ← 1
11. for k ← p to r
12.       do if L [i] ≤ R [j]
13.                then A [k] ← L [i]
14.                     i ← i +1
15.                else A [k] ← R [j]
16.                     j ← j + 1
```

To understand both the algorithm Merge-Sort (A, p, r) and MERGE (A, p, q, r); consider a list of (7) elements:

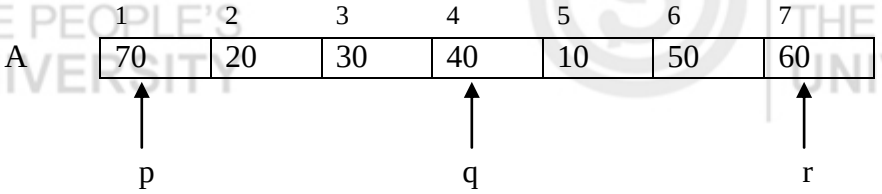


Figure 9: Merging algorithms

$$q = \left\lfloor \frac{1+7}{2} \right\rfloor = 4$$

Then we will first make two sub-lists as:
MERGER-SORT (A, p, r) → MERGE-SORT (A, 1, 4)
MERGE-SORT (A, q +1, r) → MERGE-SORT (A, 5, 7)

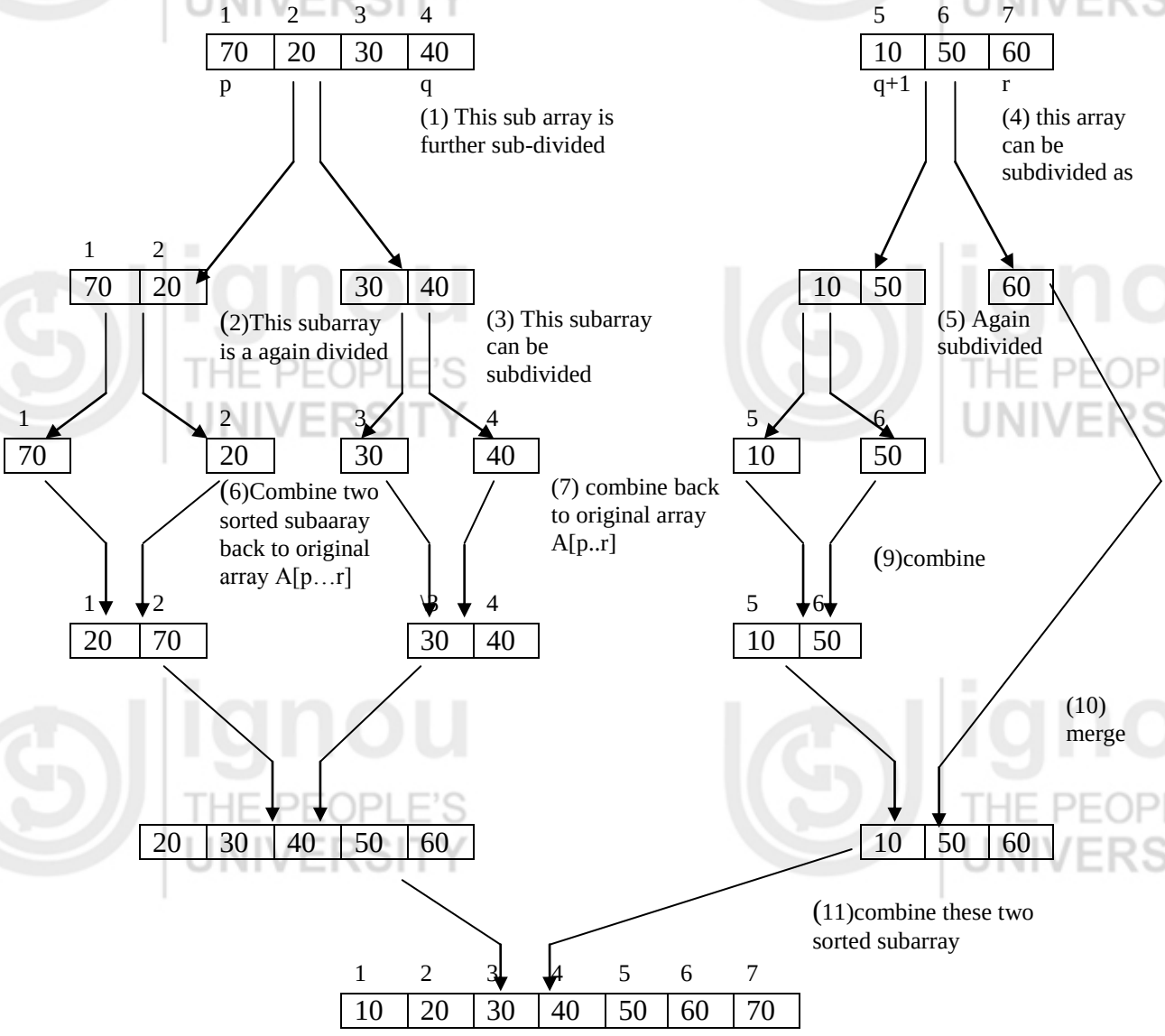


Figure 10: Illustration of merging process - 1

Lets us see the MERGE operation more closely with the help of some example. Consider that at some instance we have got two sorted subarray in A, which we have to merge in one sorted subarray.

Ex:

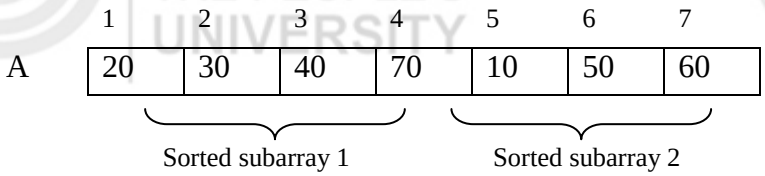
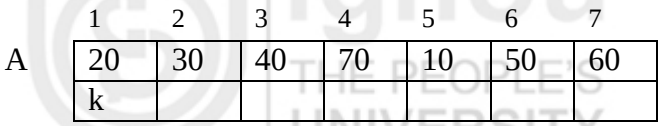


Figure 11: Example array for merging

Now we call MERGE (A,1,4,7), after line 1 to line 10, we have



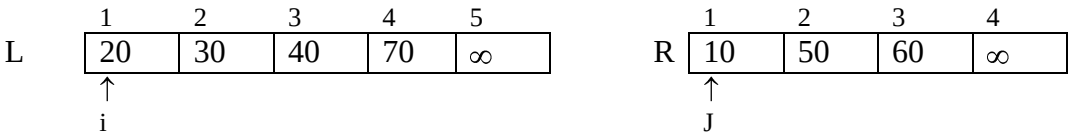
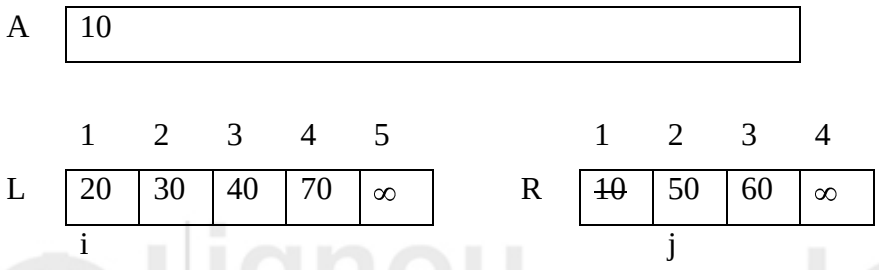


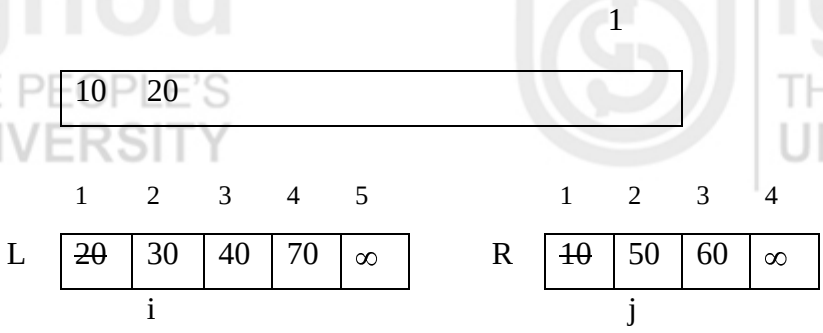
Figure 12 (a) : Illustration of merging – II using line 1 to 10

In figure (a), we just copy the A[p…q] into L[1..n_1] and A[q+1,…r] into R[1…n_2] Variable I and j both are pointing to 1st element of an array L & R, respectively.

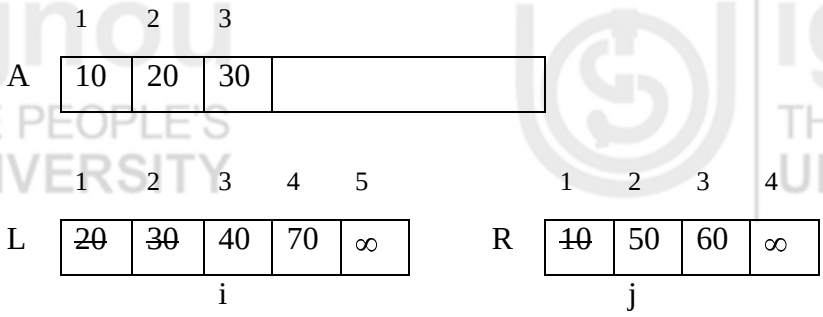
Now line 11-16 of MERGE (A,p,q,r) is used to merge the two sorted subarray L[1,…4] and R[1…4] into one sorted array [1…7]; (see figure b-h)



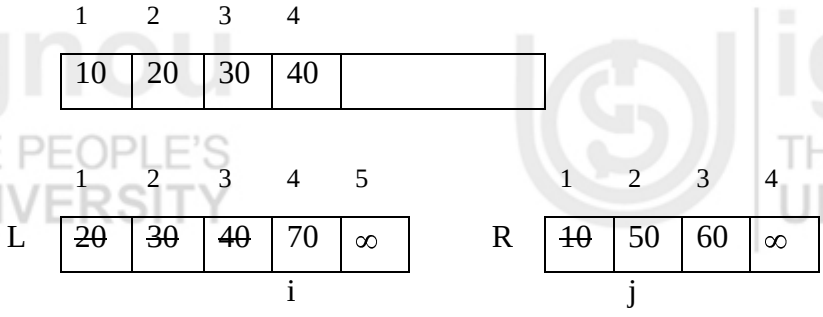
(b)


1
10 20
1 2 3 4 5
L
20 30 40 70 ∞
i
1 2 3 4
R
10 50 60 ∞
j

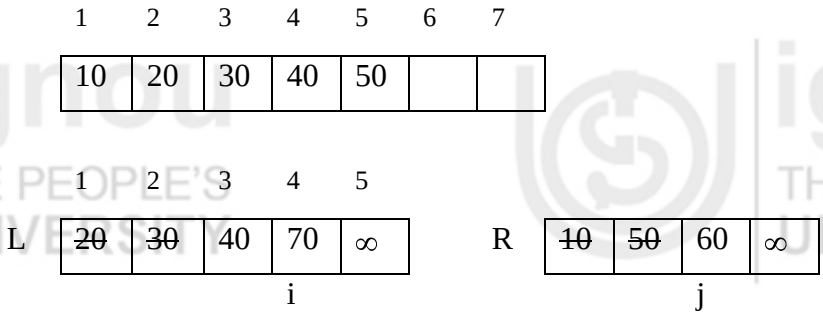
(c)


1 2 3
A
10 20 30
1 2 3 4 5
L
20 30 40 70 ∞
i
1 2 3 4
R
10 50 60 ∞
j

(d)


1 2 3 4
10 20 30 40
1 2 3 4 5
L
20 30 40 70 ∞
i
1 2 3 4
R
10 50 60 ∞
j

(e)


1 2 3 4 5 6 7
10 20 30 40 50
1 2 3 4 5
L
20 30 40 70 ∞
i
R
10 50 60 ∞
j

(f)

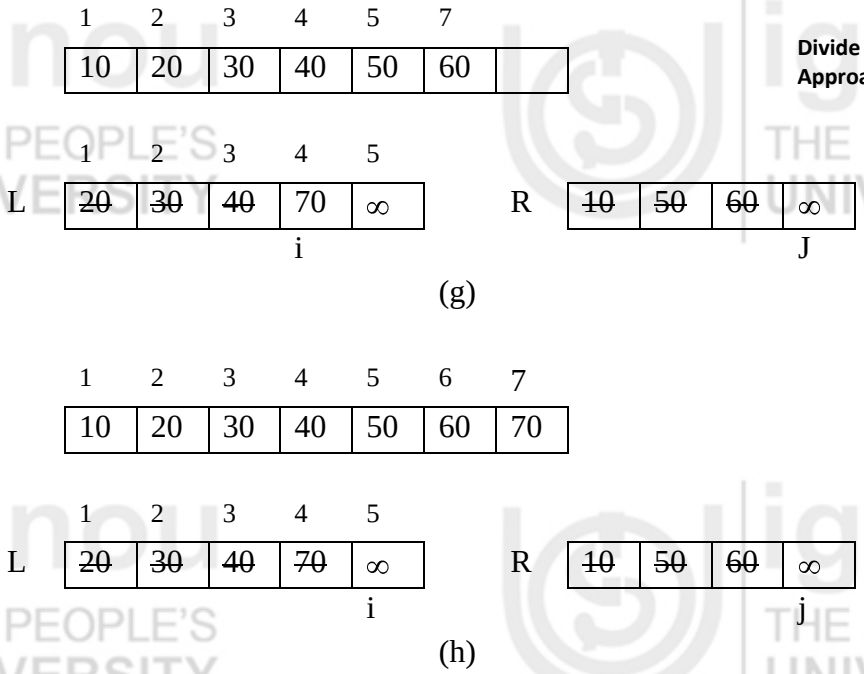


Figure 13 (a) : Illustration of merging – III using line 11 to 16

Analysis of MERGE-SORT Algorithm

- For simplicity, assume that n is a power of 2 ⇒ each divide step yields two sub-problem, both of size exactly n/2.
- The base case occurs when n = 1.
- When n ≥2, then

  Divide: Just compute q as the average of p and r ⇒ D (n) = O(1)

  Conquer: Recursively solve sub-problems, each of size $\frac{n}{2}$ ⇒2T($\frac{n}{2}$)

  Combine: MERGE an n-element subarray takes O(n) time ⇒ C (n) = O (n)

  - ❖ D(n) = O (1) and C (n) = O(n)
  - ❖ F (n) = D (n) + C (n) = O(n), which is a linear function in 'n'.

Hence Recurrence for Merge Sort algorithm can be written as:

$$T(n)=\begin{cases}\theta(1) & if\ n=1\\ 2T\left(\frac{n}{2}\right)+\theta(n) & if\ n\geq 2\end{cases} \quad \text{- (1)}$$

This Recurrence 1 can be solved by any of two methods:

(1) Master method or
(2) by Recursion tree method:

1) **Master Method:**

$$T(n)= 2T\left(\frac{n}{2}\right) + c.n \quad \text{-- (1)}$$

By comparing this recurrence with $T(n)= 2T\left(\frac{n}{2}\right)$ = + f (h)

We have: a = 2

b = 2

f (n) = n

$n^{\log_b a} = n^{\log_2 2} = n$; Now compare f(n) with $n^{\log_2 2}$ i-e ($n^{\log_2 2} = n$)

Since f(n) = n = O ($n^{\log_2 2}$) $\Rightarrow$ Case 2 of Master Method

$\Rightarrow$ T (n) = θ ($n^{\log_b a}$. logn)

= θ (n. logn)

2. Method 2: Recursion Tree Method

We rewrite the recurrence as:

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

Recursion tree:

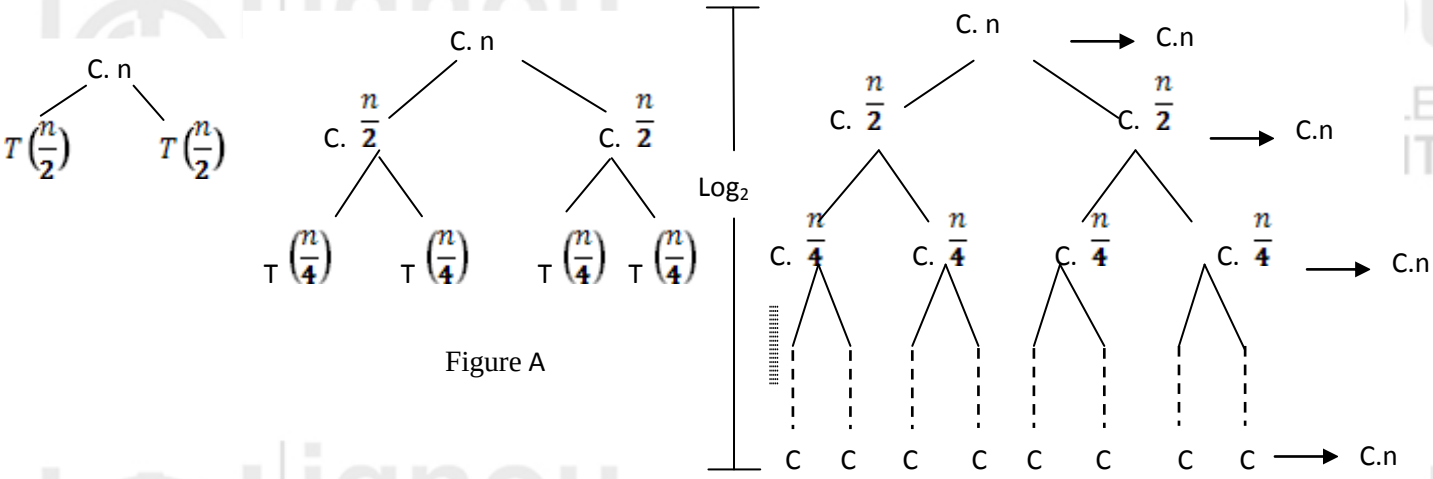


Figure A

Figure B

Total = C.n + C.n+ …… ($\log_2 n$ + 1) terms

= C.n (logn+1)

= $\theta(nlogn)$

2.4.2 QUICK-SORT

Quick-Sort, as its name implies, is the fastest known sorting algorithm in practice. The running time of Quick-Sort depends on the nature of its input data it receives for sorting. If the input data is already sorted, then this is the worst case for quick sort. In

this case, its running time is O (n^2). Inspite of this slow worst case running time, Quick sort is often the best practical of this choice for sorting because it is remarkably efficient on the average; its expected running time in θ (nlogn).

1) Worst Case (when input array is already sorted): $O(n^2)$

2) Best Case (when input data is not sorted): $\Theta(nlogn)$

3) Average Case (when input data is not sorted & Partition of array is not unbalance as worst case) : $\Theta(nlogn)$

Avanta: - Quick Sort algorithm has the advantage of "Sorts in place"

Quick Sort

The Quick sort algorithm (like merge sort) closely follows the Divide-and Conquer strategy. Here Divide-and-Conquer Strategy involves 3 steps to sort a given subarray A[p..r].

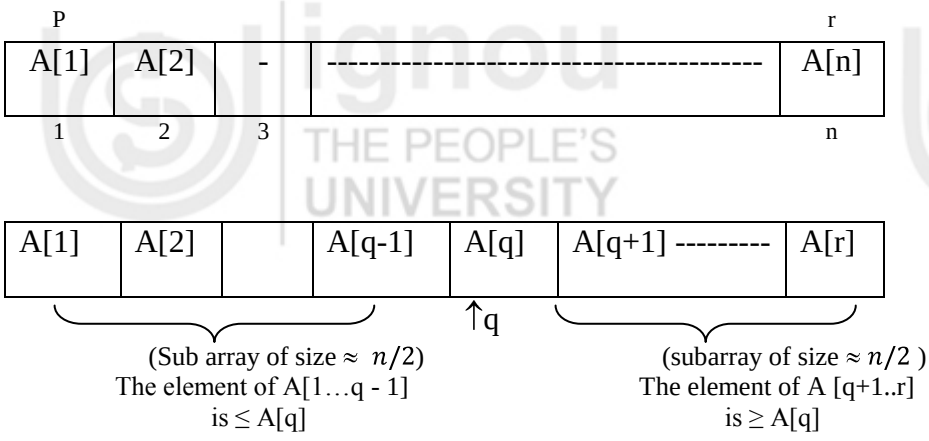
1) **Divide**: The array A [p. . r] is partitioned (rearranged) into two (possibility empty) sub-array A [p..q-1] and A [q+1,..r], such that each element in the left subarray A[p…q-1] is ≤ A [q] and A[q] is ≤ each element in the right subarray A[q+1…r]. to perform this Divide step, we use a PARTITION procedure; which returns the index q, where the array gets partitioned.

2) **Conquer**: These two subarray A [p…q-1] and A [q+1..r] are sorted by recursive calls to QUICKSORT.

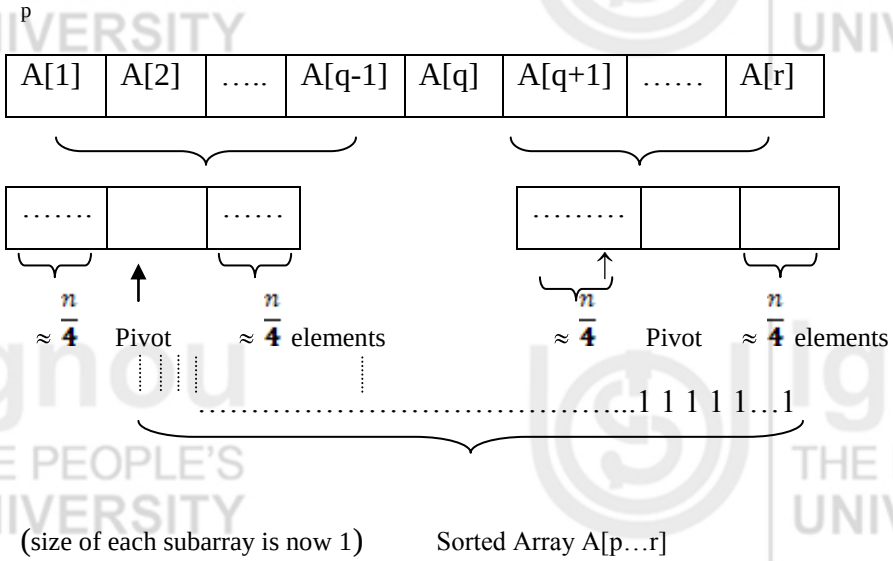
3) **Combine**: Since the subarrays are sorted in place, so there is no need to combine the subarrays.

Now, the entire array A{p..r} is sorted.

The basic concept behind Quick-Sort is as follows:
Suppose we have an unsorted input data A [p…r] to sort. Here PARTITION procedures always select a last element A[r] as a Pivot element and set the position of this A[r] as follows:



These two subarray A [p…q-1] and A[q+1..r] is further divided by recursive call to QUICK-SORT and the process is repeated till we are left with only one-element in each sub-array (or no further division is possible).



(size of each subarray is now 1) Sorted Array A[p…r]

Pseudo-Code for QUICKSORT:

```
QUICK SORT (A, p, r)
{
{ If (p < r)                    /* Base Condition
  {
  q← PARTITION (A, p, r)        /* Divide Step*/
  QUICKSORT (A, p, q-1)         /* Conquer
  QUICKSORT (A, q+1, r)         /* Conquer
    }
}
```

- To sort an array A with n-elements, a initial call to QuickSort in QUICKSORT (A, 1, n)
- QUICKSORT (A, p, r) uses a procedure Partition (), which always select a last element A[r], and set this A[r] as a Pivot element at some index (say q) in the array A[p..r].

  The PARTITION () always return some index (or value), say q, where the array A[p..r] partitioned into two subarray A[p..q-1] and A[q+1…r] such that A[p..q-1] ≤ A[q] and A[q] ≤ A[q+1…r].

Pseudo code for PARTITION:

```
PARTITION (A, p, r)
{
1:   x ← A[r]                    /* select last element
2:   i ← p – 1                   /* i is pointing one position
                                    before than p, initially
3:   for j ← p to r − 1 do
     {
4:     if A[j] ≤ r A [r]
       {
5:       i ← i + 1
6:       Exchange (A[i] ↔ A[j])
       }
     }/* end for
7:   Exchange (A [i + 1] and A[r])
8:   return ( i+ 1)
}
```

The running time of PARTITION procedure is θ (n), since for an array A[a…n], the loop at line 3 is running 0(n) time and other lines at code take constant time i.e. 0(1) so overall time is 0(n).

To illustrate the operational PARTITION procedure, consider the 8-element array:

A	2	8	7	1	3	5	6	4
	1	2	3	4	5	6	7	8

Step 1: The input array with initial value of I, j, p and r.

x ← A[r] = 4
I ← p-1 = 0

	p							r
	2	8	7	1	3	5	6	4
i	1	2	3	4	5	6	7	8

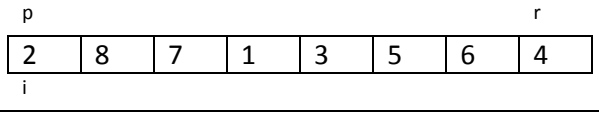
Step 2: The array A after executing the line 3 – 6
a)

J = 1 to 7
1) j = 1; if A[1] ≤ a i.e. 2 ≤ A[r] ⇒ YES
Therefore i ← i + 1 = 0 + 1 = 1
exchange (A[1] ↔ A[1])

	p							r
	2	8	7	1	3	5	6	4
	i							

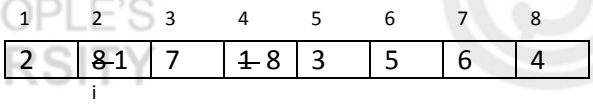
b)

2) j = 2; if A[2] ≤ 4 i.e 8 ≤ 4 ⇒ No
So line 5 – 6, will not be executed
Thus:



c)

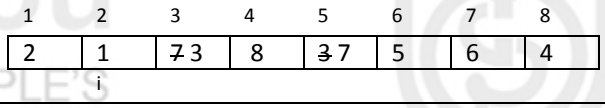
3) j = 3; if A[3] ≤ 4 i.e 7 ≤ 4 ⇒ No; so line 5-6 will not execute
4) j = 4; if A[4] ≤ 4 i.e 1 ≤ 4 ⇒ YES
So i ← i + 1 = 1 + 1 = 2
exchange (A[2] ↔ A[4])



d)

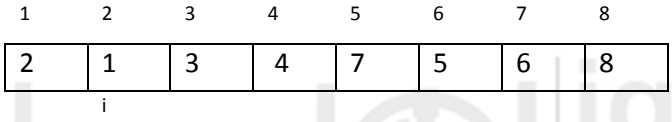
5) j = 5; A[5] ≤ 4 i.e 3 ≤ 4 ⇒ YES
- i ← i + 1 = 2 + 1 = 3

exchange (A [3] ↔ A [5])

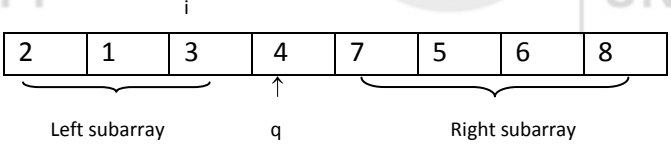


e)

6) j = 6; A[6] ≤ 4 i.e 5≤ 4 ⇒ NO
7) j = 7: A [7] ≤ 4 i.e 6 ≤ 4 ⇒ NO
Now for loop is now finished; so finally line 7 is execute i.e
exchange (A[4] ↔ A[8]), so finally we get:



Finally we return (i + 1) i.e (3 + 1) = 4; by this partition procedure;
Now we can easily see that all the elements of A[1, … 3] ≤ A[4]; and all the elements of A[5, ..8] ≥ A[4]. Thus



To sort the entire Array A{1..8}; there is a Recursive calls to QuickSort on both the subarray A[..3] and A[5…8].

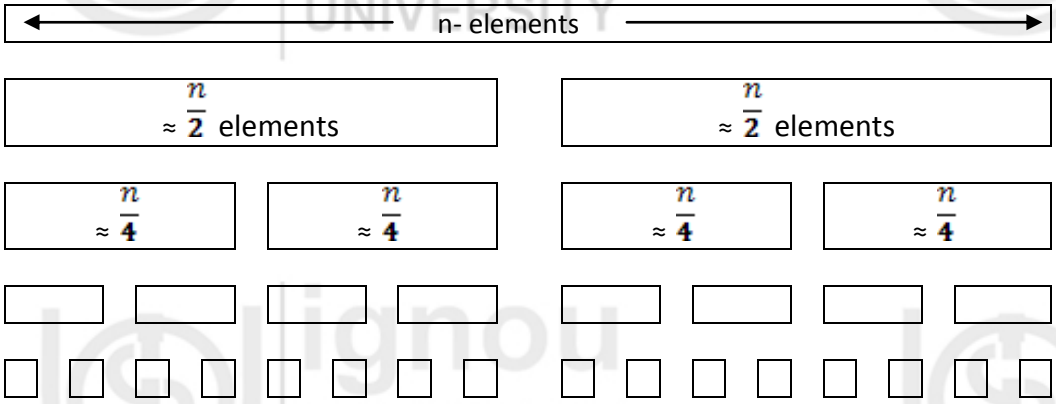
Performance of Quick Sort

The running time of QUICK SORT depends on whether the partitioning is balanced or unbalanced. Partitioning of the subarrays depends on the input data we receive for sorting.

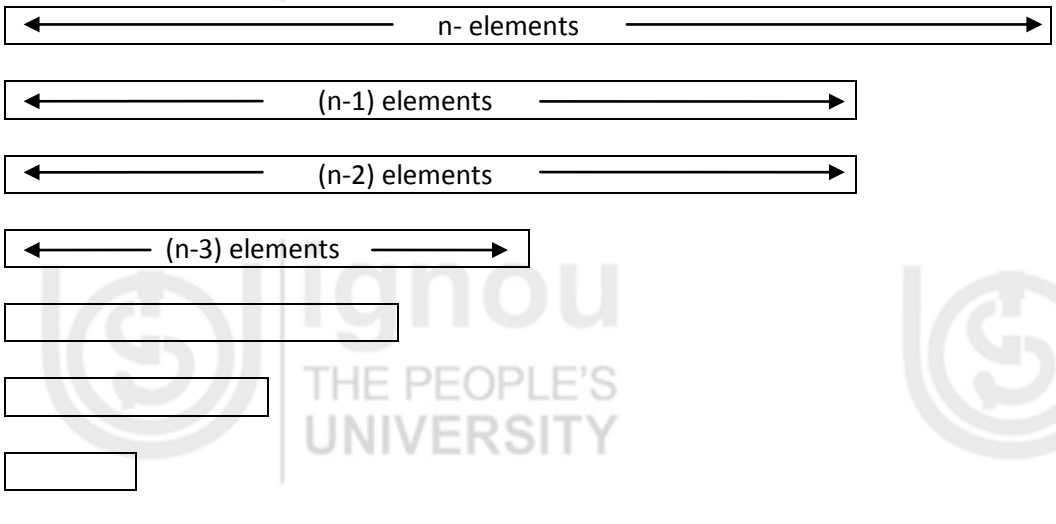
Best Case: If the input data is not sorted, then the partitioning of subarray is balanced; in this case the algorithm runs asymptotically as fast as merge-sort (i.e. 0(nlogn)).

Worst Case: If the given input array is already sorted or almost sorted, then the partitioning of the subarray is unbalancing in this case the algorithm runs asymptotically as slow as Insertion sort (i.e. θ (n^2)).

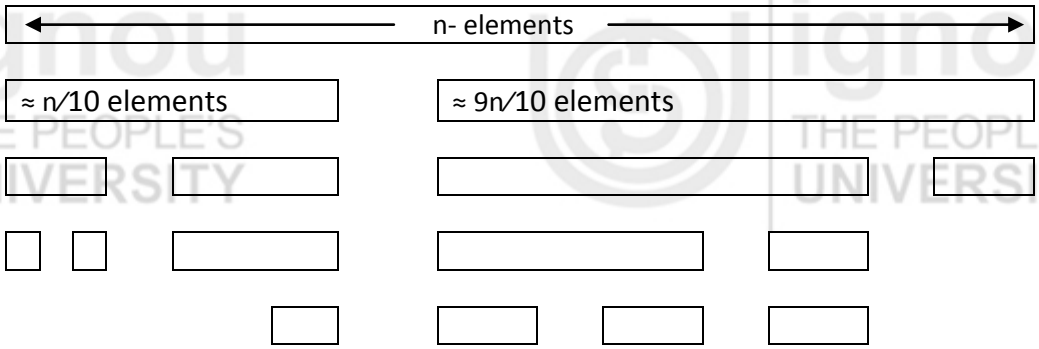
Average Case: Except best case or worst case.
The figure (a to c) shows the recursion depth of Quick-sort for Best, worst and average cases:



(a) Best Case



(b) Worst Case



(c) Average Case

Best Case (Input array is not sorted)

The best case behaviour of Quicksort algorithm occurs when the partitioning procedure produces two regions of size $\approx \frac{n}{2}$ elements.

In this case, Recurrence can be written as:

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

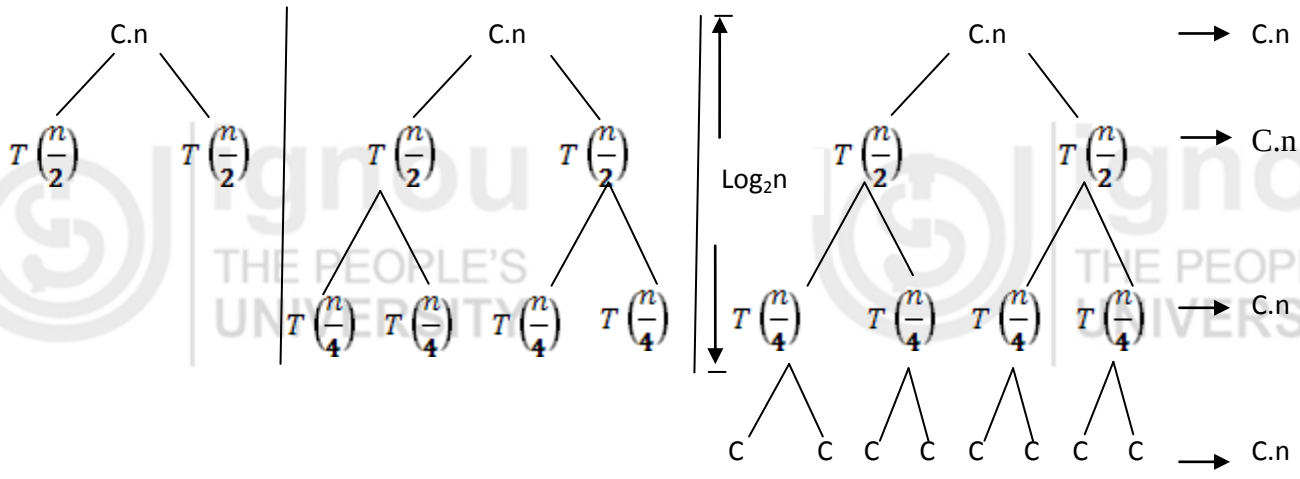
Method 1: Using Master Method; we have a=2; b=2, f(n)=n and $n^{\log_b a} = n^{\log_2 2} = n$

- F(n) = n= 0 ($n^{\log_2 2}$) →Case 2 of master method

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

Method 2: Using Recursion Tree:

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



Total = C.n + C.n+ - - - - - -+ $\log_n n$ terms (c)

= C.n logn

= θ (nlogn)

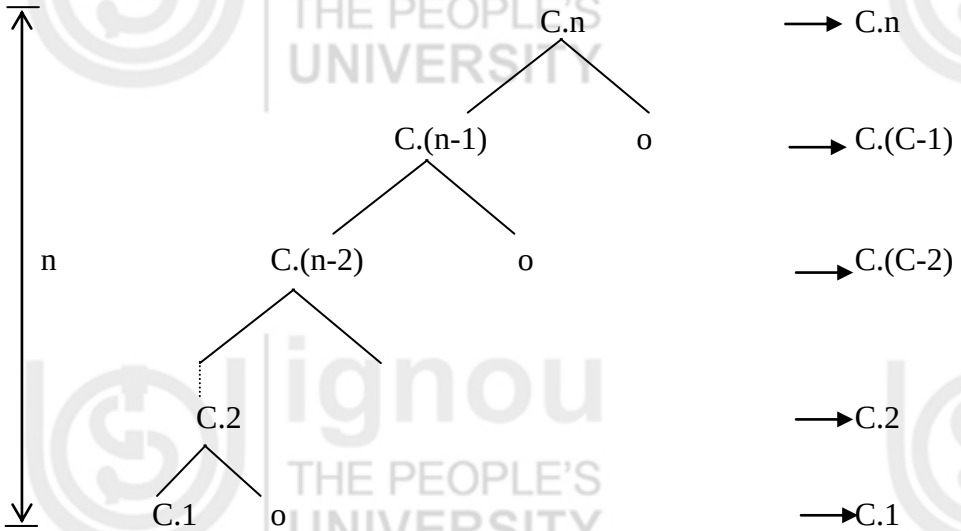
Worst Case:
[When input array is already sorted]

The worst case behaviour for QuickSort occurs, when the partitioning procedures one region with (n-1) elements and one with 0-elements → completely unbalanced partition.

In this case:

$T(n) = T(n-1) + T(0) + \theta(n)$
$= T(n-1) + 0 + C.n$

Recursion Tree:



Total = C (n + (n-1) + (n-2) + -------+2+1)

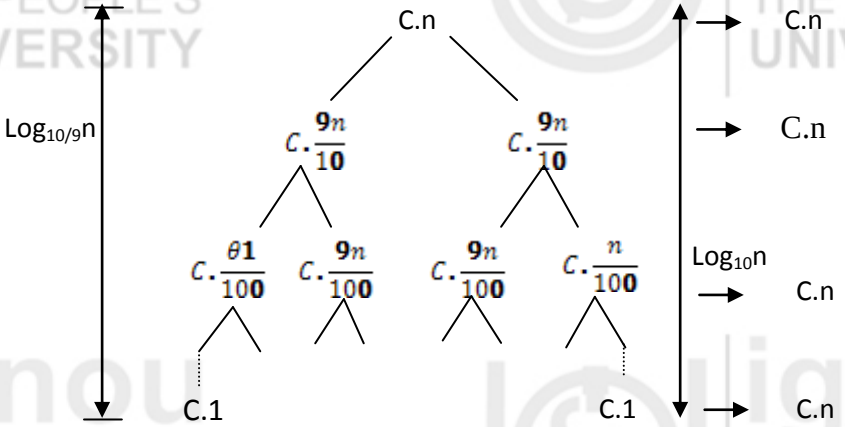
$= C.\left(\frac{n(n+1)}{2}\right) = 0(n^2)$

Average Case

Quick sort average running time is much closer to the best case.
Suppose the PARTITION procedure always produces a 9-to-1 split so recurrence can be:

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \theta(n)$$

Recursion Tree:



For Smaller Height:

Total = C.n+C.n+----$\log_{10}$n times

= C.n$\log_{10}$n

= Ω (nlogn)

For Bigger height

Total = C.n.+C.n +---$\log_{10/9}$n

= C.n$\log_{10/9}$n

= 0(nlogn)

T(n) = Ω (nlogn) — (1)
& T (n) = 0(nlogn) — (2) } ⇒ T(n) = θ (nlogn)

☞ **Check Your Progress 2**

(Objective questions)

1) Which of the following algorithm have same time complexity in Best, average and worst case:
   a) Quick sort b) Merge sort c) Binary search d) all of these

2) The recurrence relation of MERGESORT algorithm in worst case is:
   a) $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$ b) $T(n) = T\left(\frac{n}{2}\right) + O(n^2)$
   c) $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ d) $T(n) = 2T\left(\frac{n}{2}\right) + O(nlogn)$

3) The recurrence relation of QUICKSORT algorithm in worst case is:
   a) $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ b) $T(n) = T(n-1) + O(n)$
   c) $T(n) = T\left(\frac{n}{2}\right) + O(n)$ d) $T(n) = 2T(n-1) + O(n)$

4) The running time of PARTITION procedure of QUICKSORT algorithm is
   a) $\Theta(n^2)$ b) $\Theta(nlogn)$ c) $\Theta(n)$ d) $\Theta(logn)$

5) Suppose the input array A[1…n] is already in sorted order (increasing or decreasing) then it is _____ case situation for QUICKSORT algorithm
   a) Best b) worst c) average d) may be best or worst

6) Illustrate the operation of MERGESORT algorithm to sorts the array: $A[1 \ldots 9] =$

70	35	5	85	45	88	50	10	60

7) Show that the running time of MERGESORT algorithm is $\Theta(nlogn)$.

8) Illustrate the operation of PARTITION Procedure on the array
$A =< 35,10,40,5,60,25,55,30,50,25 >$

9) Show that the running time of PARTITION procedure of QUICKSORT algorithm on a Sub-array on size n is $\Theta(n)$.

10) Show that the running time of QUICKSORT algorithm in the best case is $\Theta(nlogn)$

11) Show that the running time of QUICKSORT algorithm is $\Theta(n^2)$ when all elements of array A have the same value.

12) Find the running time of QUICKSORT algorithm when the array A is sorted in non increasing order.

2.5 INTEGER MULTIPLICATION

Input: Two n-bit decimal numbers x and y are represented as:
$X = < x_{n-1}\ x_{n-2}$ --------$x_1 x_0 >$ and
$Y = < y_{n-1}\ y_{n-2}$ --------$y_1 y_0 >$, where each x_i and y_i E$\{0, 1 \ldots.9\}$.

Output: The 2n-digit decimal is representative of the product x.y;
$x.\ y = z = z_{2n-2}\ z_{2n-3}$ …….$z_1\ z_0$

Note: The algorithm, which we are discussing here, works for any number base, e.g., binary, decimal, hexadecimal etc. For simplicity matter, we use decimal number.

The straight forward method (Broute force method) requires $0(n^2)$ time to multiply two n-bit numbers. But by using divide and conquer, it requires only $0(n^{\log_2 3})$ i.e. $0(n^{1.59})$ time.

In 1962, A.A. Karatsuba discovered an asymptotically faster algorithm $0(n^{1.59})$ for multiplying two n-digit numbers using divide & conquer approach.

A Divide & Conquer based algorithm splits the number X and Y into ②equal parts as:

X= | a | b | = | x_{n-1} ⌈n/2⌉ | x_{n-2}----------x | x ⌈n/2⌉ ---------- x_1 x_0 | = a × $10^{n/2}$ + b

n/2 n/2

Y= | c | d | = | y_{n-1} ⌈n/2⌉ | y_{n-2}----------x | x ⌈n/2⌉– 1---------- $y_1\ y_0$ | = c × $10^{n/2}$ + d

n/2 n/2

Note: Both number X and Y should have same number of digits; if any number has less number of digits then add zero's at most-significant bit position. So that we can easily get a, b, c and d of $\frac{n}{2}$ - digits. Now X and Y can be written as:

$X = a * 10^{n/2} + b$ -----①

$Y = c * 10^{n/2} + d$ ------②

For example : $\lfloor n/2 \rfloor$ = largest integer less than or equal to $n/2$

If X = 1026732

Y = 743914

Then X = 1026732 = 1026 x 10^3 + 732

Y = 0743914 = 0743 x 10^3 + 914

Now we can compute the product as:

Z = X.Y $\left(a.10^{\lfloor n/2 \rfloor} + b\right).\left(c.10^{\lfloor n/2 \rfloor} + d\right)$

X.Y = $a.c.10^{2.\lfloor n/2 \rfloor} + (bc + ad)10^{\lfloor n/2 \rfloor} + b.d$ --------- (1)

Where a, b, c, d is $\frac{n}{2}$ - digits. This equation ⊛ requires 4 multiplication of size $\frac{n}{2}$ – digits and 0(n) additions; Hence

$$T(n) = 4T\left(\frac{n}{2}\right) + 0(n)$$

After solving this recurrence using master method, we have: $T(n) = \theta(n^2)$; So direct (or Brute force) method requires $O(n^2)$ time.

Karatsuba method (using Divide and Conquer)

In 1962, A.A. Karatsuba discovered a method to compute X.Y (as in Equation(1)) in only 3 multiplications, at the cost of few extra additions; as follows:

Let U = (a+b).(c+d)

V = a.c

W = b.d

Now X.Y = $V.10^{2.\lfloor n/2 \rfloor} + (U - V - W).10^{\lfloor n/2 \rfloor} + W$ ---------- (2)

Now, here, X.Y (as computed in equation (2)) requires only 3 multiplications of size n/2, which satisfy the following recurrence:

$$T(n) = \begin{cases} 0(1) & \text{if } n=1 \\ 3T(n/2) + 0(n) & \text{Otherwise} \end{cases}$$

Where 0(n) is the cost of addition, subtraction and digit shift (multiplications by power of 10's), all these takes time proportional to 'n'.

Method 1: - (Master Method)

$$T(n) = 3T\left(\frac{n}{2}\right) + 0(n)$$

a = 3
b = 2
f(n) = n
$n^{\log_b a} = n^{\log_2 3}$
$f(n)=n=0(n^{\log_2 3-\epsilon}) \Rightarrow$ case 1 of Master Method

$\Rightarrow T(n) = \theta(n^{\log_2 3})$
$\Rightarrow \theta(n^{1.59})$

Method 2 (Substitution Method)

$$T(n) = 3T\left(\frac{n}{2}\right) + 0(n)$$
$$= 3T\left(\frac{n}{2}\right) + c.n$$

Now

$$T(n) = c.n + 3T\left(\frac{n}{2}\right)$$
$$= c.n + 3\left\{c.\frac{n}{2} + 3T\left(\frac{n}{4}\right)\right\}$$
$$= c.n + \frac{3}{2}\,c\,n + 3^2\,.T\left(\frac{n}{4}\right)$$
$$= c.n + \frac{3}{2}\,c\,n + 3^2\left\{c\,\frac{n}{4} + 3T\left(\frac{n}{8}\right)\right\}$$
$$= c.n + \frac{3}{2}\,c\,n + \left(\frac{3}{2}\right)^2 c\,n + 3^3 T\left(\frac{n}{8}\right)$$
$$= c.n + \frac{3}{2}\,c\,n + \left(\frac{3}{2}\right)^2 c\,n + 3^3\left\{c.\frac{n}{8} + 3T\left(\frac{n}{16}\right)\right\}$$
$$= c.n + \frac{3}{2}\,c\,n + \left(\frac{3}{2}\right)^2 c\,n + \left(\frac{3}{2}\right)^3 c\,n + \cdots + \left(\frac{3}{2}\right)^k .T\left(\frac{n}{2k}\right)$$
$$= c.n + \frac{3}{2}\,c\,n + \left(\frac{3}{2}\right)^2 c\,n + \left(\frac{3}{2}\right)^3 c\,n + \cdots + \left(\frac{3}{2}\right)^{logn} .c$$
$$= c.n\left(1 + \frac{3}{2}\,c\,n + \left(\frac{3}{2}\right)^2 c\,n + \left(\frac{3}{2}\right)^3 c\,n + \cdots + \left(\frac{3}{2}\right)^{logn}\right)$$
$$= c.n\left[\frac{1\left[\left(\frac{3}{2}\right)^{logn+1} - 1\right]}{\frac{3}{2} - 1}\right]$$
$$= 2\,c.n\left[\left(\frac{3}{2}\right)^{logn+1} - 1\right]$$
$$= 2\,c.n\left[\left(\frac{3}{2}\right)^1\left(\frac{3}{2}\right)^{logn} - 1\right]$$
$$= 2\,c.n\left[\frac{3}{2}\,n^{log_{2^3}/_2} - 1\right]$$
$$= 2\,c.n\left[\frac{3}{2}\,n^{log_{2^3-log_{2^2}}} - 1\right]$$

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

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

$$= 3\,c.n^{log_2 3} - 2\,c.n$$

$$= \mathbf{0}(n^{log_2 3})$$

2.6 MATRIX MULTIPLICATION

Let A and B be two (n x n) – matrices.

$A = (a_{ij})$ I,j = 1…n $[a_{ij}]$ wher i,j = 1…….n
$B = (b_{ij})$ i.j = 1…n c_{ij} B= $[b_{ij}]$ where i, j = n

The product matrix C = A.B. = $(Cij)i_{,j\ =1\ldots n}$ is also an (n x n) matrix, whose $(i,j)^{th}$ elements is defined as:

$$\mathbf{Cij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Straight forward method:

To compute Cij using this formula, we need multiplications.
As the matrix C has (n^2) elements, the time for the resulting matrix multiplication is

1) Divide & Conquer Approach:

The divide and conquer strategy is yet another way to compute the product of two (n x n) matrices. Assuming that n is an exact power of 2 (i.e. $n=2^k$). We divide each of A, B and C into four

$\left(\frac{n}{2}\times\frac{n}{2}\right)$ matrices. i.e.

$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and B = $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ where each A_{ij} and B_{ij} are sub matrices of size $\left(\frac{n}{2}\times\frac{n}{2}\right)$,

$$A = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

i.e. $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
$C_{12} = A_{11}B_{12} + A_{12}B_{22}$
$C_{21} = A_{21}B_{11} + A_{22}B_{21}$
$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

(2) Here all Aij, Bij are sub matrices of size $\left(\frac{n}{2}\times\frac{n}{2}\right)$,

Algorithm Divide and Conquer Multiplication (A,B)

1. n ← no. of rows of A
2. if n = 1 then return (a_{11} b_{11})
3. else
4. Let Aij, Bij (for i,j = 1,2, be $\left(\frac{n}{2}\times\frac{n}{2}\right)$ submatrices)

S.t $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

5. Recursively compute $A_{11}B_{11}$, $A_{12}B_{21}$, $A_{11}B_{12}$,……. $A_{22}B_{22}$

6. Compute $C_{11} = A_{11.}B_{11} + A_{12.}B_{21}$
   $C_{12} = A_{11.}B_{12} + A_{12.}B_{22}$
   $C_{21} = A_{21.}B_{11} + A_{22.}B_{21}$
   $C_{22} = A_{21.}B_{21} + A_{22.}B_{22}$

7. Return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Analysis of Divide and Conquer based Matrix Multiplication

Let T(n) be the no. of arithmetic operations performed by D&C-MATMUL.

- Line 1,2,3,4,7 require $\theta($ 1) arithmetic operations.
- Line 5, requires 8T $\left(\frac{n}{2}\right)$ arithmetic operations.
  (i.e. in order to compute AB using e.g. (2), we need 8- multiplications of

  $\left(\frac{n}{2}\ \left(\frac{n}{2}\right)\right.$ matrices).

- Line 6 requires 4 $\left(\frac{n}{2}\right)$ = $\theta(n^{\ 2})$

  (i.e 4 additions of $\left(\frac{n}{2}\ \left(\frac{n}{2}\right)\right.$ matrices)

So the overall computing time, T(n), for the resulting Divide and conquer Matrix Multiplication

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

Using Master method, a = 8, b = 2 and f(n) = n^2 ; since
f(n) = n^2 = O ($n^{log2-€)}$ => case 1of master method
⇨ T (n) = Q ($n^{\log_2 8)}$ = Q (n^3)

Now we can see by using V. Stressen's Method, we improve the time complexity of matrix multiplication from O(n^3) to O ($n^{2.81}$).

3) Strassen's Method

Volker Strassen had discovered a way to compute the Cij of Eg. (2) using only (7) multiplication and (18) additions / subtractions.

This method involves (2) steps:

1) Let

$$P_1 = (A_{11}+A_{22})\,(B_{11}+B_{22})$$
$$P_2 = (A_{21}+A_{22}).B_{11}$$
$$P_3 = A_{11}\,(B_{12}-B_{22})$$
$$P_4 = A_{22}\,(B_{21}-B_{11})$$
$$P_5 = (A_{11}+A_{12}).B_{22}$$
$$P_6 = (A_{21}-A_{11})\,(B_{11}+B_{12})$$
$$P7 = (A_{12}-A_{22})\,(B_{21}+B_{22}) \quad \text{——— (I)}$$

Recursively compute the (7) $\left(\frac{n}{2}\,(\frac{n}{2})\right)$ matrices $P_1, P_2 \ldots\ldots P_7$ as in eg. (I)

2) Then, the Cij are computed using the formulas in eg. (II).

$$C_1 = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6 \quad \text{——— (II)}$$

Here the overall computing time

$$T(n) = \begin{cases} \theta(1) & n < 2 \\ 7T(n/2) + \theta(n^2) & \text{Otherwise} \end{cases} \Rightarrow$$

Using master method: a = 7, b = 2 and $n^{\log_b a} = n^{\log_2 7} = n^{2.81}$ $f(n) = n^2$;

$f(n) = n^2 = O(n^{\log 2-\epsilon})$ => case 1of master method

$\Rightarrow$ $T(n) = Q(n^{\log_b a}) = Q(n^{\log_2 7})$

$= Q(n^{2.81})$.

Ex:- To perform the multiplication of A and B

$$AB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

We define the following eight n/2 by n/2 matrices:

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} \quad A_{12} = \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix} \qquad B_{12} = \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \quad B_{12} = \begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix}$$

$A_{21} = \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix}$ $A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix}$ $B_{21} = \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix}$ $B_{22} = \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$

Strassen showed how the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions (12 additions and 6 subtractions):

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22})B_{11}$$
$$P_3 = A_{11}(B_{12} - B_{22})$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12})B_{22}$$
$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

The correctness of the above equations is easily verified by substitution.

$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$

$= \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$

$P_2 = (A_{21} + A_{22}) \times B_{11} = \begin{bmatrix} 14 & 23 \\ 14 & 23 \end{bmatrix}$

$P_3 = A_{11} \times (B_{12} - B_{22}) = \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$

$P_4 = A_{22} \times (B_{21} - B_{11}) = \begin{bmatrix} -3 & 2 \\ 5 & -20 \end{bmatrix}$

$P_5 = (A_{11} \times A_{12}) \times B_{22} = \begin{bmatrix} 34 & 18 \\ 45 & 9 \end{bmatrix}$

$P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = \begin{bmatrix} 3 & 27 \\ -18 & -18 \end{bmatrix}$

$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = \begin{bmatrix} 18 & 16 \\ 3 & 0 \end{bmatrix}$

$C_{11} = P_{21} + P_4 - P_5 + P_7 = \begin{bmatrix} 17 & 22 \\ 21 & 18 \end{bmatrix}$

$C_{12} = P_3 + P_5 = \begin{bmatrix} 31 & 30 \\ 33 & 33 \end{bmatrix}$

$C_{21} = P_2 + P_4 = \begin{bmatrix} 11 & 25 \\ 19 & 3 \end{bmatrix}$

$C_{22} = P_1 + P_3 - P_2 + P_6 = \begin{bmatrix} 22 & 38 \\ 14 & 30 \end{bmatrix}$

$$C = \begin{bmatrix} 17 & 22 & 31 & 30 \\ 21 & 18 & 33 & 33 \\ 11 & 25 & 22 & 38 \\ 19 & 3 & 14 & 30 \end{bmatrix}$$

The overall time complexity of stressen's Method can be written as:

$$T(n) = \begin{cases} 0(1) & if\ n = 1 \\ 7T(n/2) + 0(n^2) & \end{cases}$$

Otherwise

a = 7; b = 2; f(n) = n^2

$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

f (n) $n^2 = 0(n^{\log_2 7 - \epsilon})$ => case 1of master method

⇨ T (n) = Q ($n^{\log_2 7}$)

= Q ($n^{2.81}$).

The solution of this recurrence is T(n) = $0(n^{\log_2 7}) = 0(n^{2.81})$

☞ Check Your Progress 3

(Objective questions)

1) The recurrence relation of INTEGER Multiplication algorithm using Divide & conquer is:

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

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

2) Which one of the following algorithm design techniques is used in Strassen's matrix multiplication algorithm?

(a) Dynamic programming
(b) Backtracking approach
(c) Divide and conquer strategy
(d) Greedy method

3) Strassen's algorithm is able to perform matrix multiplication in time_____.

(a)$O(n^{2.61})$ (b) $O(n^{2.71})$ (c) $O(n^{2.81})$ (d) $O(n^3)$

4) Strassen's matrix multiplication algorithm (*C = AB),*if the matrices *A* and *B* are not of type $2^n \times 2^n$, the missing rows and columns are filled with _____.

(a) 0′*s* (b) 1's (c) -1's (d) 2's

5) Strassen's matrix multiplication algorithm (*C = AB),* the matrix *C* can be computed using only 7 block multiplications and 18 block additions or subtractions. How many additions and how many subtractions are there out of 18?

(a) 9 and 9 (b) 6 and 12 (c)12 and 6 (d) none of theses

6) Multiply 1026732× 0732912 using divide and conquer technique (use karatsuba method).

7) Use Strassen's matrix multiplication algorithm to multiply the following two matrices:

$$A=\begin{bmatrix}5 & 3\\ 6 & 7\end{bmatrix} \quad B=\begin{bmatrix}1 & 4\\ 5 & 9\end{bmatrix}$$

2.7 SUMMARY

- Many useful algorithms are recursive in structure as they make a recursive call to itself until a base (or boundary) condition of a problem is not reached. These algorithms closely follow the **Divide and Conquer** approach.

- **Divide and Conquer** is a top-down approach, which directly attack the complete instance of a given problem and break down into smaller parts.

- Any divide-and-conquer algorithms consists of 3 steps:
  1) **Divide**: The given problem is divided into a number of sub-problems.
  2) **Conquer**: Solve each sub-problem be calling them recursively.
  (**Base case:** If the sub-problem sizes are small enough, just solve the sub-problem in a straight forward or direct manner).
  3) **Combine**: Finally, we combine the sub-solutions of each sub-problem (obtained in step-2) to get the solution to original problem.

- To analyzing the running time of divide-and-conquer algorithms, we use a **recurrence equation** (more commonly, a **recurrence**). Any algorithms which follow the divide-and-conquer strategy have the following recurrence form:

$$T(n)=\begin{cases}\theta(1) & if\ n \le c\\ aT\left(\frac{n}{b}\right)+D(n)+C(n) & Otherwise\end{cases}$$

Where

T(n) = running time of a problem of size n

a means "In how many part the problem is divided"

$T\left(\frac{n}{b}\right)$ means "Time required to solve a sub-problem each of size (n/b)"

D(n) + C(n) = f(n) is the summation of the time requires to divide the problem and combine the sub-solutions.

- Applications of divide-and conquer strategy are Binary search, Quick sort, Merge sort, multiplication of two n-bit numbers and V. Strassen's matrix multiplications.

- The following table summarizes the recurrence relations and time complexity of the various problems solved using Divide-and-conquer.

Problems that follows	Recurrence relation	Time complexity		
		Best	Worst	Average
Binary search	Worst case: $T(n) = T\left(\frac{n}{2}\right) + k$	O(1)	$O(logn)$	$O(logn)$
Quick Sort	Best case: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(nlogn)$	$O(n^2)$	$O(nlogn)$
Merge sort	Worst Case: $T(n) = T(n-1) + O(n)$ Average: $T(n) = T\left(\frac{3n}{10}\right) + T\left(\frac{7n}{10}\right) + O(n)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
Multiplication of two n-bits numbers	Best case or worst: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$		$O\left(n^{log_2 3}\right) \approx O(n^{1.59})$	
Strassen's matrix multiplication	Worst case: $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$ Worst case: $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$		$O(n^{log_2 7}) \approx O(n^{2.81})$	

2.8 SOLUTIONS/ANSWERS

Check Your Progress 1

(Objective Questions): 1-c, 2-b, 3-a, 4-c,

Solution 5: Refer page number……., example1 of binary search.

Solution 6: Best case: $\Theta(1)$.

Worst case:

Binary search closely follow the Divide-and-conquer technique.
We know that any problem, which is solved by using Divide-and-Conquer having a recurrence of the form : $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Since at each iteration, the array is divided into two sub-arrays but we are solving only one sub-array in the next iteration. So value of a=1 and b=2 and f(n)=k where k is a constant less than n.

Thus a recurrence for a binary search can be written as

$T(n) = T\left(\frac{n}{2}\right) + k$; by solving this recurrence using substitution method, we have:

$k + k + k + \cdots \ldots \ldots \ldots up\ to\ (logn) terms = k.logn = O(logn)$.

Average case: Same as worst case: $O(logn)$

Check Your Progress 2

(Objective Questions): 1-b, 2-c, 3-b, 4-c,5-b

Solution 6: Refer numerical question of merge sort on page number- ,

Solution 7: A recurrence relation for MERGE_SORT algorithm can be written as:

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

Using any method such as Recursion-tree or Master Method (as given in UNIT-1), we have $T(n) = \Theta(nlogn)$.

Solution 8:

Let

A[1…10] =

1	2	3	4	5	6	7	8	9	10
35	10	40	5	60	25	55	30	50	25
									x

Here $p = 1$ $r = 10$
$x = A[10] = 25$
$i = p - 1 = 0$
$j = 1\ to\ 9$

(1)

$j = 1\ and\ i = 0$
$A[j] = A[1] = 35$ and $35 \not\leq 25$
(2)
$j = 2$ and $i = 0$
$A[2] = 10 \not\leq 25$(True)
then $i = 0 + 1$ and A[1]A ↔ [2]
i.e.,

p									*r*
1	2	3	4	5	6	7	8	9	10
10	35	40	5	60	25	55	30	50	25
i									

(3)
Now $j = 3$ *and* $i = 1$
$A[3] = 40 \nleq 25$
(4)
$j = 4$ and $i = 1$
$A[4] = 5 \leq 25$(True)
then $i = 1 + 1 = 2$ and A[2] ↔ A[4]
i.e.,

1	2	3	4	5	6	7	8	9	10
10	5	40	35	60	25	55	30	50	25
	i								

(5) $j = 5, i = 2$
$A[5] = 60 \nleq 25$

(6) j= 6 and $i = 2$
$A[j] = 25 \nleq 25$(True)
then. $i = 2 + 1 = 3$ and A[3] ↔ A[6]
i.e.,

1	2	3	4	5	6	7	8	9	10
10	5	25	35	60	40	55	30	50	25
		i							

(7)
$j = 7 \quad i = 3$
$A[7] = 55 \nleq 25$

(8)
$j = 8 \quad i = 3$
$A[8] = 25 \nleq 25$

(9)
$j = 9 \quad i = 3$
$A[9] = 50 \nleq 25$
Now $A[i+1]$ $i.e.$ $A[4] \leftrightarrow A[10]$
i.e.,

1	2	3	4	5	6	7	8	9	10
10	5	25	20	60	40	50	30	50	35
		i							

Here PARTITION procedure finally return index (i+1)=4, where the array gets partitioned. Now the two sub-arrays are

15	1	25

60	40	50	25	50	36

Solution 9: By analyzing PARTITION procedure , we can see at line 3 to 6 **for loop** is running $maximum\ O(n)\ time$. Hence PARTITION procedure takes $O(n)\ time$.

Solution 10: A recurrence relation for QUICKSORT algorithm for best case can be written as:

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

Using any method such as Recursion-tree or Master Method (as given in UNIT-1), we have $T(n) = \Theta(nlogn)$.

Solution 11: when all elements of array A have the same value, then we have a worst case for QUICKSORT and in worst case QUICKSORT algorithm requires $\Theta(n^2)$ time.

Solution 12: when the array A is sorted in non increasing order, then we have a worst case for QUICKSORT and in worst case QUICKSORT algorithm requires $\Theta(n^2)$ time.

Check Your Progress 3

(Objective Questions): 1-d, 2-c, 3-c, 4-a,5-c

Solution 6

1026732 × 732912

In order to apply Karatsuba's method, first we make number of digits in the two numbers equal, by putting zeroes on the left of the number having lesser number of digits. Thus, the two numbers to be multiplied are written as

x = 1026732 and y = 0732912.
As n = 7, therefore [n/2] = 3, we write
x = $1026 \times 10^3 + 732 = a \times 10^3 + b$
y = $0732 \times 10^3 + 912 = c\ 10^3 + d$

where a = 1026, b = 732
c = 0732, d = 912

Then

$$x \times y = (1026 \times 0732)\,10^{2\times3} + 732 \times 912 + [(1026 + 732) \times (732 + 912) - (1026 \times 0732) - (732 \times 912)]10^{3}$$

$$= (1026 \times 0732)\,10^{6} + 732 \times 912 + [(1758 \times 1644) - (1026 \times 0732) - (732 \times 912)]10^{3} \quad \ldots (A)$$

Though, the above may be simplified in another simpler way, yet we want to explain Karatsuba's method, therefore, next, we compute the products.

U = 1026 × 732
V = 732 × 912
P = 1758 × 1644

Let us consider only the product 1026 × 732 and other involved products may be computed similarly and substituted in (A).

Let us write

$$U = 1026 \times 732 = (10 \times 10^{2} + 26)\,(07 \times 10^{2} + 32)$$
$$= (10 \times 7)\,10^{4} + 26 \times 32 + [(10 + 7)\,(26 + 32) - 10 \times 7 - 26 \times 32)]\,10^{2}$$
$$= 17 \times 10^{4} + 26 \times 32 + (17 \times 58 - 70 - 26 \times 32)\,10^{2}$$

At this stage, we do not apply Karatsuba's algorithm and compute the products of 2-digit numbers by conventional method.

Solution7:

Strassen's matrix multiplication:

$$A = \begin{bmatrix} 5 & 3 \\ 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 5 \\ 6 & 9 \end{bmatrix}$$

We set $C = A \times B$ and partition each matrix into four sub-matrices.

Accordingly, $A_{11} = [5], A_{12} = [3], A_{21} = [6], A_{22} = [7],$

$B_{11} = [1],\ B_{12} = [5], B_{21} = [6]\ B_{22} = [9]$

$where$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},\ A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

Applying Strassen's algorithm, we compute the following products:

$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = ([5] + [7]) \times ([1] + [9]) = [120]$
$P_2 = (A_{21} + A_{22}) \times B_{11} = ([6] + [7]) \times [1] = [13]$
$P_3 = A_{11} \times (B_{12} - B_{22}) = [5] \times ([5] - [9]) = [-20]$
$P_4 = A_{22} \times (B_{21} - B_{11}) = [7] \times ([6] - [1]) = [35]$
$P_5 = (A_{11} + A_{12}) \times B_{22} = ([5] + [3]) \times [9] = [72]$
$P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = ([6] - [5]) \times ([1] + [5]) = [6]$
$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = ([3] - [7]) \times ([6] + [9]) = [-60]$

From the above products, we can compute C as follows:

$C_1 = P_1 + P_4 - P_5 + P_7 = [120]+[35]-[72]-[60]=[23]$
$C_{12} = P_3 + P_5 = [-20]+[72]=[52]$
$C_{21} = P_2 + P_4 = [13]+[35]=[48]$
$C_{22} = P_1 + P_3 - P_2 + P_6 = [120]+[-20]-[13]+[6]=[93]$

$$C = \begin{bmatrix} 23 & 52 \\ 48 & 93 \end{bmatrix}$$

2.9 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).

UNIT 3 GRAPH ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

The vast majority of computer algorithm operate on data. Organsing these data in a certain way (i.e. data structure) has a significant role is design and analysis of algorithm. Graph is one such fundamental data structure. Array, linked list, stack, queue, tree, sets are other important data structures. A graph is generally used to represent connectivity information i.e. connectivity between cities for example. Graphs have been used and considered very interesting data structures with a large number of applications for example the shortest path problem. While several representations of a graph are possible, we discuss in the unit the two most common representations of a graph: adjacency matrix and adjacency list. Many graph algorithms requires visiting nodes and vertices of a graph. This kind of operation is also called traversal. You must have read various traversal methods for tree such as preorder, postorder and inorder In this unit we present two graph traversal algorithms which are called as Depth first search and Breadth first search algorithm.

3.1 OBJECTIVES

After going through this unit you will be able to

- define a graph,
- differentiate between an undirected and a directed graph,
- represent a graph though a adjacency matrix and an adjacency list and;
- traverse a graph using DFS and BFS.

3.2 BASIC DEFINITION AND TERMINOLOGIES

A graph G = (V. E) is a set of vertices V, with edges connecting some of the vertices (edge set E). An edge between vertex u and v is denoted as (u, v). There are two types of a graph: (1) undirected a graph and directed graph (digraph). In a undirected graph the edges have no direction whereas in a digraph all edges have direction.

You can notice that edges have no direction. Let us have an example of an undirected graph (figure 1) and a directed graph (figure 2)

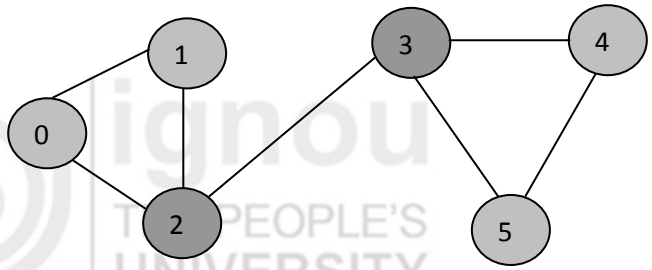


Figure.1 Undirected graph

V = {0, 1, 2, 3, 4, 5}

E = {(0, 1) , (0, 2),

(1,2), or (2, 2) both are same

(2, 3),

(3, 4), (3, 5)

(4, 5)

}

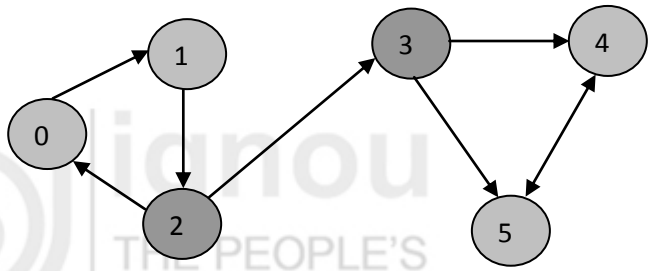


Figure 2: Diagraph

V = {0, 1, 2, 3, 4, 5, }

E = { (0, 1),

(1, 2)

(2, 0), (2, 3),

(3, 4), (3, 5)

(4, 5) and (5, 4) are not the same. These are two different edges.

(5, 4)

You can notice in Figure 2 that edges have direction

You should also consider the following graph preparations.

The geometry of drawing has no particular meaning: edges of a graph can be drawn "straight" or "curved".

A vertex v is adjacent to vertex u, if there is an edge (u, v). In an undirected graph, existence of edge (u, v) means both u and v are adjacent to each other. In a digraph, existence of edge (u, v) does not mean u is adjacent to v.

PATH

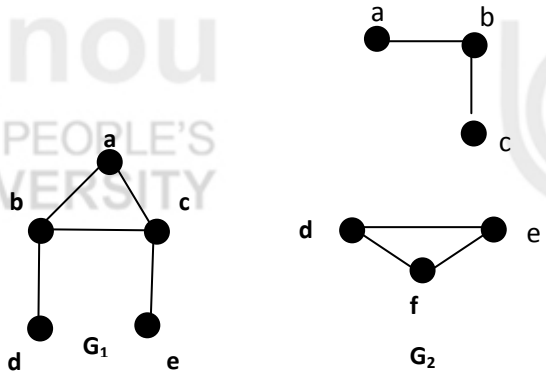
An edge may not have a weight. A path in a graph is sequence of vertices $V_1 V_2 \ldots V_n$ such that consecutive vertices $V_i V_{i+1}$ have an edge between them, i.e., V_{i+1} is adjacent to V_i

A path in a graph is simple if all vertices are distinct i.e. no repetition of a path of any vertices (and therefore edges) in the sequence, except possibly the first and the last one. Length of a path is the number of edges in the path. A cycle is a path of length at least 1 such that the first and the last vertices are equal. A cycle is a simple path with the same vertex as the first and the last vertex in the sequence if the path is simple. For undirected graph, we require a cycle to have distinct edges. Length of a cycle is the number of edges in the cycle.

There are many problems in computer science such as of route with minimum time and diagnostic: minimum shortcut path routing, traveling sales problem etc. can be designed using paths obtained by marking traversal along the edges of a graph.

CONNECTED GRAPHS

Connectivity: A graph is connected if there is a path from every vertex to every other vertex. In an undirected graph, if there is a path between every pair of distinct vertices of the graph, then the undirected graph is connected. The following example illustrates this:



(a) Connected **(b) Unconnected**

Figure 3: The connected and unconnected undirected graph

In the above example G, there is a path between every pair of distinct vertices of the graph, therefore G_1 is connected. However the graph G_2 is not connected.

In directed graph, two vertices are strongly connected if there is a (directed) path from one to the other.

Undirected: Two vertices are connected if there is a path that includes them.

Directed: Two vertices are strongly-connected if there is a (directed) path from any vertex to any other.

3.3 GRAPH REPRESENTATION

In this section, we will study the two more important data structure for graph representation: Adjacency matrix and Adjacency list.

3.3.1 ADJACENCY MATRIX

The adjacency matrix of a graph G = {V, E} with n vertices is a *n* x *n* boolean/matrix. In this matrix the entry in the *i*th row and *j*th column is 1 if there is an edge from the *i*th vertex to the *j*th vertex in the graph G. If there is no such edge, then the entry will be zero. It is to be noted that
(i) the adjacency matrix of a undirected graph is always symmetric, i.e., M [*i*,*j*] = M [*j*, *i*]

(ii) The adjacency matrix for a directed graph need not be symmetric.

(iii) The memory requirement of an adjacency matrix is n^2 bits

For example for the graph in the following figure (a) is adjacency matrix is given in (b)

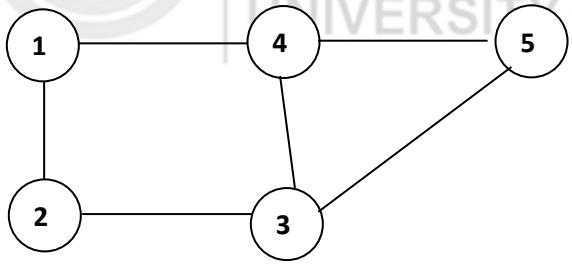


Figure. 4 (a)

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	1	1	0	1	1
	0	0	1	0	1
	0	0	1	1	0

Figure.4 (b) Adjacency Matrix

Let us answer the following questions:

(i) Suppose if we want to know how much time will take in finding number of edges a graph with *n* vertices?

Since the space needed to represent a graph is n^2 bits where *n* is a number of vertices. All algorithm will require at least 0 (n^2) time because $n^2 - n$ entries of the matrix have to be examined. Diagonal entries are zero.

(ii) Suppose the most of the entries in the adjacency matrix are zeros, i.e., when a graph is a sparse... How much time is needed to the find *m* number of edges in a graph? It will take much less time if say 0 ($e + n$), where *e* is the number of edges is a graph and $e << n^2/2$. But this can be achieved if a graph is represented through an adjacency list where only the edges will be represented.

3.3.2 ADJACENCY LIST

The adjacency list of a graph or a diagraph is a set of linked lists, one linked list for each vertex. The nodes in the linked list *i* contain all the vertices that are adjacent to vertex *i* of the list (i.e. all the vertices connected to it by an edge). The following figure.5 represents adjacency list of the graph in figure 4 (a).

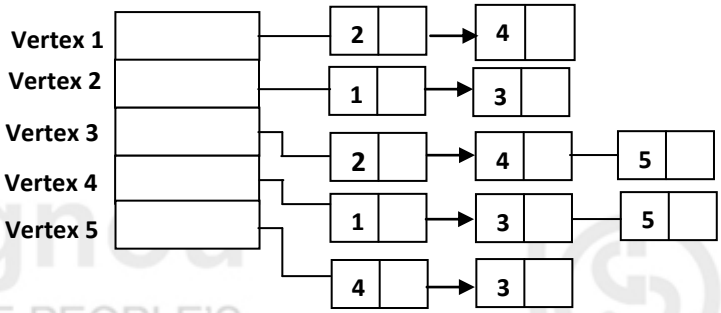


Figure. 5 Adjacency List

Putting it in another way of an adjacency list represents only columns of the adjacency matrix for a given vertex that contains entries as 1's. It is to be observed that adjacency list compared to adjacency matrix consumes less memory space if a graph is sparse. A graph with few edges is called sparse graph. If the graph is dense, the situation is reverse. A dense graph, is a graph will relatively few missing edges. In case of an undirected graph with *n* vertices and e edge adjacency list requires *n* head and 2 *e* list nodes (i.e. each edges is represented twice).

What is the storage requirement (in terms of bits) for a adjacency list of any graph!

(i) For storing *n* (*n* vertices) head nodes – we require – $\log_2 n$ bits –

(ii) For storing list nodes for each head *n* nodes – we require $\log n + \log e$

Therefore total storage requirement in item of bits for adjacency matrix is $^2log_2 n$ ($^2log_2{}^n + log_2{}^e$)

Question. What is time complexity in determining number of edges in an undirected graph.

It may be done in just 0 ($n + e$) because in degree of any vertex (i.e. number of edges incident to that vertex) in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

Use of adjacency matrix or adjacency list for representing your graph – depends upon the type of a problem; type of algorithm to be used for solving a problem and types of a input graph (dense or sparse)

3.4 GRAPH TRAVERSAL ALGORITHMS

3.4.1 DEPTH-FIRST SEARCH

You are aware of tree traversal mechanism. Give a tree, you can traverse it using preorder, inorder and postorder. Similarly given an undirected graph you can traverse it or visit its nodes using breadth first-search and depth-first search.

Searching in breadth-first search or depth first search means exploring a given graph. Through searching a graph one can find out whether a graph is connected or not? There are many more applications of graph searching algorithms. In this section we will illustrate Depth First Search algorithm followed by Breadth first Search algorithm in the next section.

The logic behind this algorithm is to go as far as possible from the given starting node searching for the target. In case, we get a node that has no adjacent/successor node, we get back (recursively) and continue with the last vertex that is still not visited.

Broadly it is divided into 3 steps:

- Take a vertex that is not visited yet and mark it visited
- Go to its first adjacent non-visited (successor) vertex and mark it visited
- If all the adjacent vertices (successors) of the considered vertex are already visited or it doesn't have any more adjacent vertex (successor) – go back to its parent vertex

Before starting with an algorithm, let us discuss the terminology and structure used in the algorithm. The following algorithm works for undirected graph and directed graph both.

The following color scheme is to maintain the status of vertex i.e mark a vertex is visited or unvisited or target vertex:

white- for an undiscovered/unvisited vertex

gray - for a discovered/visited vertex

black - for a finished/target vertex

The structure given below is used in the algorithm.

p[u]- Predecessor or parent node.

Two (2) timestamps referred as

t[u] – First time discovering/visiting a vertex, store a counter or number of times

f[u]= finish off / target vertex

Let us write the algorithm DFS for any given graph G. In graph G, V is the vertex set and E is the set of edges written as G(V,E). Adjacency list for the given graph G is stored in Adj array as described in the previous section.

color[] - An array color will have status of vertex as white or gray or black as defined earlier in this section.

```
DFS(G)
 {
    for each v in V,                //for loop  V+1 times
    {
        color[v]=white; // V times
         p[v]=NULL;                 // V times
    }
    time=0;                         // constant time O(1)
   for each u in V,                 //for loop V+1 times
     if (color[u]==white)           // V times
                DFSVISIT(u) // call to DFSVISIT(v) , at most V times O(V)

 }
```

```
DFSVISIT(u)
  {
     color[u]=gray;                 // constant time
     t[u] = ++time;
     for each v in Adj(u)           // for loop
        if (color[v] == white)
           {
            p[v] = u;
             DFSVISIT(v);           // call to DFSVISIT(v)
           }
     color[u] = black;              // constant time
     f[u]=++time;                   // constant time
}
```

Complexity analysis

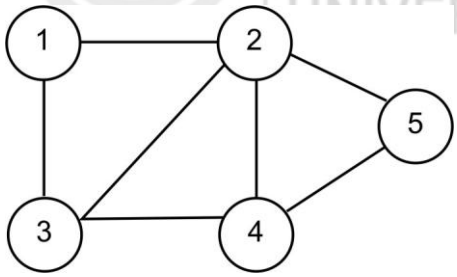
In the above algorithm, there is only one DFSVISIT(u) call for each vertex u in the vertex set V. Initialization complexity in DFS(G) for loop is O(V). In second for loop of DFS(G) , complexity is O(V) if we leave the call of DFSVISIT(u).

Now, Let us find the complexity of function DFSVISIT(u)

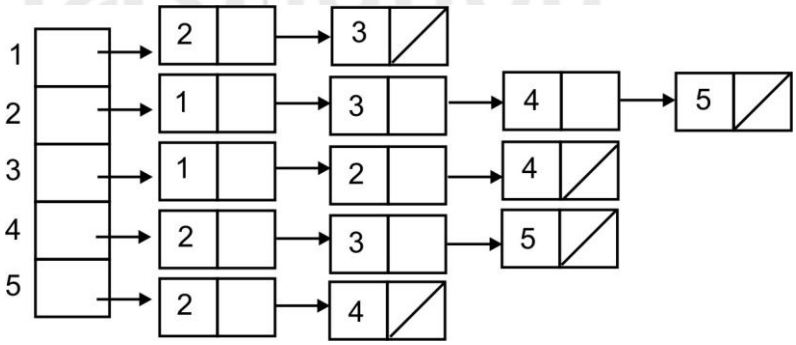
The complexity of for loop will be O(deg(u)+1) if we do not consider the recursive call to DFSVISIT(v). For recursive call to DFSVISIT(v), (complexity will be O(E) as Recursive call to DFSVISIT(v) will be at most the sum of degree of adjacency for all vertex v in the vertex set V. It can be written as $\sum_{v \in V} |Adj(v)| = O(E)$

Hence, overall complexity for DFS algorithm is O(V + E)

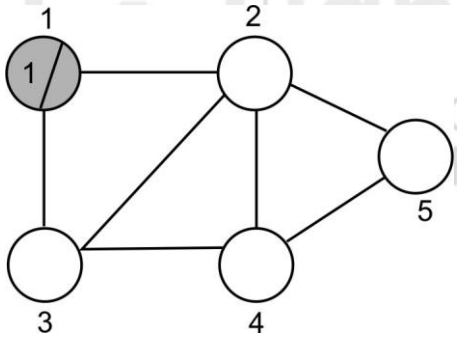
The strategy of the DFS is to search "deeper" in the graph whenever possible. Exploration of vertex is in the fashion that first it goes deeper then widened. Let us take up an example to see how exploration of vertex takes place by Depth First Search algorithm.

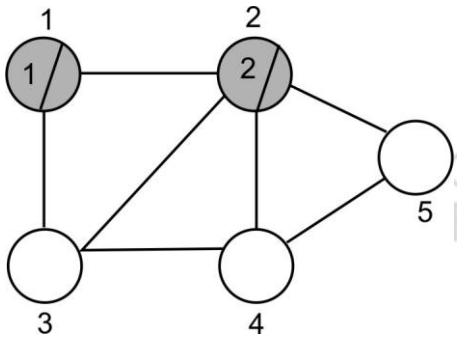


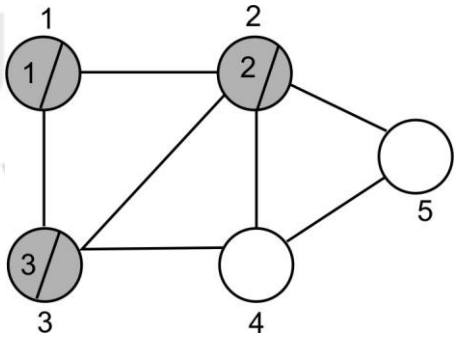
Adjacency list of the above graph is as below:

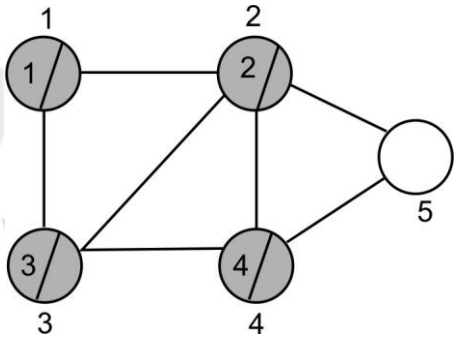


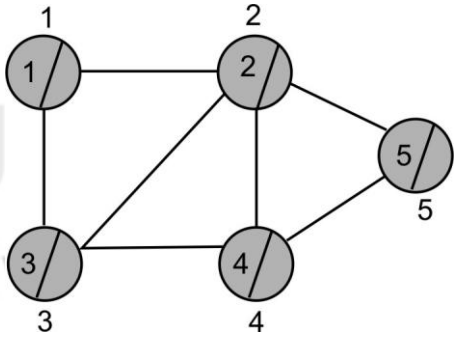
Let us explore the vertices of the graph using DFS algorithm.

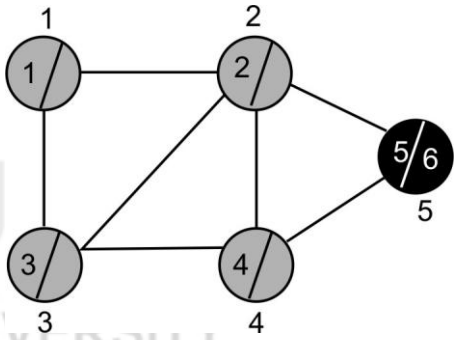


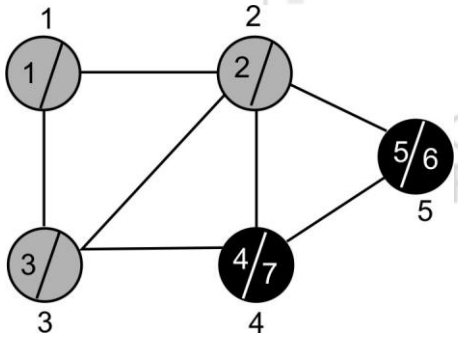


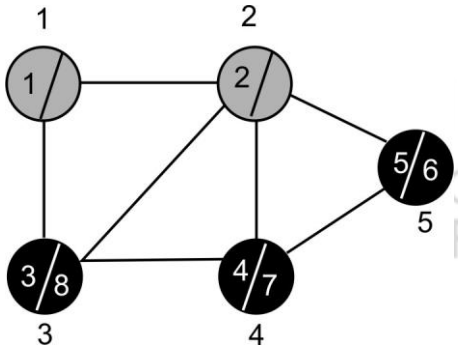

1
2
1
2
5
3
4
3
4

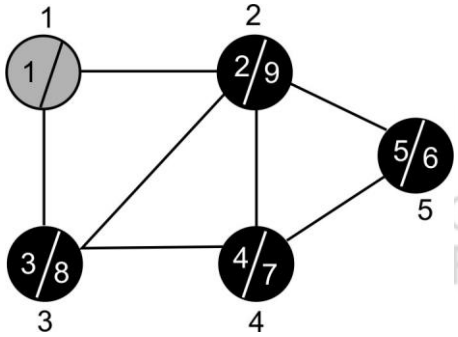

1
2
1
2
5
3
4
3
4

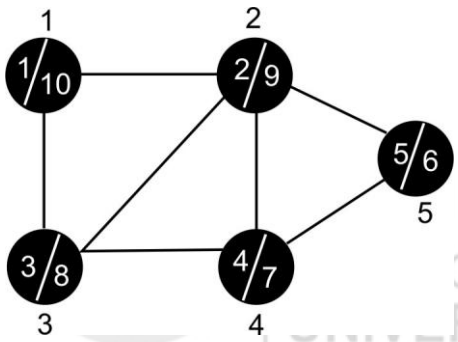

1
2
1
2
5
5
3
4
3
4


1
2
1
2
5/6
5
3
4
3
4

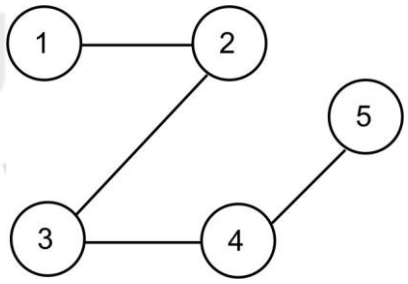








Now each vertex of the given graph is visited/explored by DFS algorithm and DFS tree is as follows:



Data structure used for implementing DFS algorithm is stack. In the diagram along with each vertex start and finish time is written in the format a/b here a represent start time and b represent finish time. This will result in to tree or forest. The order of vertices explored by DFS algorithm according to adjacency list considered for given graph is 1,2,3,4,5.

3.4.2 BREADTH-FIRST SEARCH

In this section, we will discuss breadth first search algorithm for graph. This is very well known searching algorithm. A traversal depends both on the starting vertex, and on the order of traversing the adjacent vertices of each node. The analogy behind breadth first search is that it explores the graph wider then deeper. The method starts with a vertex v then visit all its adjacent nodes v1,v2,v3…then move to the next node which is adjacent to v1, v2, v3 …. This also referred as level by level search.

Basic steps towards exploring a graph using breadth-first search:

- Mark all vertices as "unvisited".
- Start with start vertex v
- Find an unvisited vertex that are adjacent to v , mark them visited
- Next consider all recently visited vertices and visit unvisited vertices adjacent to them
- Continue this process till all vertices in the graph are explored /visited

Now, let us see the structure used in this algorithm and color scheme for status of vertex.

Color scheme is same as used in DFS algorithm i.e to maintain the status of vertex i.e mark a vertex is visited or unvisited or target vertex:

white- for an undiscovered/unvisited vertex

gray - for a discovered/visited vertex

black - for a finished/target vertex

The structure given below is used in the algorithm. G will be the graph as G(V,E) with set of vertex V and set of edges E.

p[v]-The parent or predecessor of vertex

d[v]-the number of edges on the path from *s* to *v*.

Data structure used for breadth-first search is queue, Q (FIFO), to store gray vertices.

color[v]- This array will keep the status of vertex as white, grey or black

The following algorithm for BFS takes input graph G(V,E) where V is set of vertex and E is the set of edges. Graph is represented by adjacency list i.e Adj[]. Start vertex is s in V.

```
Line    BFS(G,s)
No.     {
1.              for each v in V  - {s}                  // for loop
                {
2.                      color[v]=white;
3.                      d[v]= INFINITY;
4.                      p[v]=NULL;
                }
5.              color[s] = gray;
6.              d[s]=0;
7.              p[s]=NULL;
8.              Q=∅;            // Initialize queue is empty
9.              Enqueue(Q,s); /* Insert start vertex s in Queue Q */
10.             while Q is nonempty             // while loop
               {
11.                     u = Dequeue[Q];  /* Remove an element from Queue Q*/

12.                     for each v in Adj[u]            // for loop
                        {
13.                       if (color[v]  == white) /*if v is unvisted*/
                         {
14.                         color[v] = gray;            /* v is visted */
15.                         d[v]  = d[u] + 1;     /*Set distance of v to no. of edges
        from s to u*/
16.                         p[v]  = u;                  /*Set parent of v*/
17.                         Enqueue(Q,v);       /*Insert  v in Queue Q*/
                          }
                        }
18.                     color[u]  = black;          /*finally visted or explored vertex
u*/
                }
        }
```

Complexity Analysis

In this algorithm first for loop executes at most O(V) times.

While loop executes at most O(V) times as every vertex v in V is enqueued only once in the Queue Q. Every vertex is enqueued once and dequeued once so queuing will take at most O(V) time.

Inside while loop, there is for loop which will execute at most O(E) times as it will be at most the sum of degree of adjacency for all vertex v in the vertex set V.

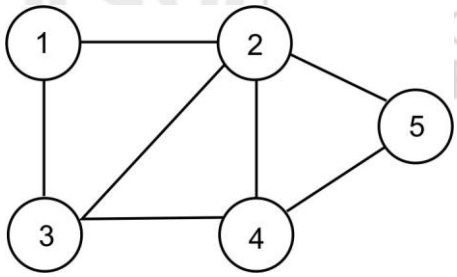
Which can be written as $\sum_{v \varepsilon V} |Adj(v)| = O(E)$

Let us summarize the number of times a statement will execute in the algorithm for BFS.

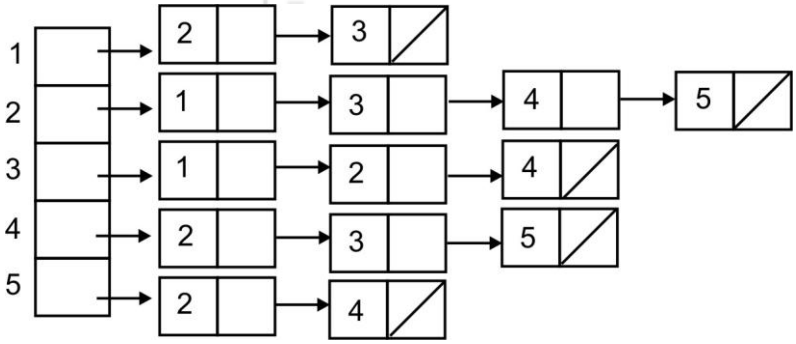
Line no.	No. of times statement will execute	Cost
1	V	O(V)
2	V-1	
3	V-1	
4	V-1	O(1)
5	1	
6	1	
7	1	
8	1	
9	1	
10	V+1	O(V)
11	V	
12	V+E+1	O(V+E)
13	V+E	
14	V	
15	V	
16	V	
17	V	
18	V	

Thus overall complexity of BFS will be V + V + E, i.e. O(V+E)

Let us take up an example to see how exploration of vertex takes place by Breadth First Search algorithm.



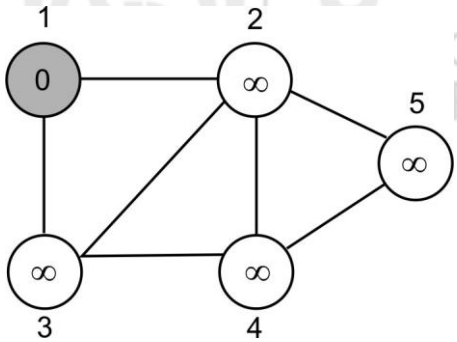
The adjacency list of the above graph is as below:

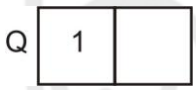


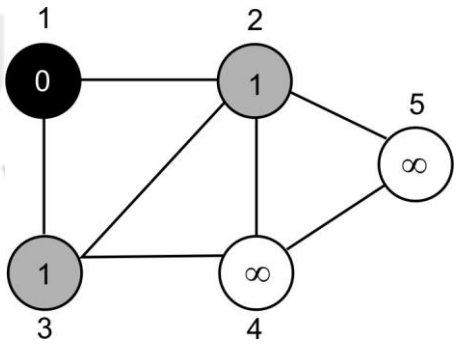
Let us explore vertices of the graph by BFS algorithm.

Consider initial vertex as vertex 1.

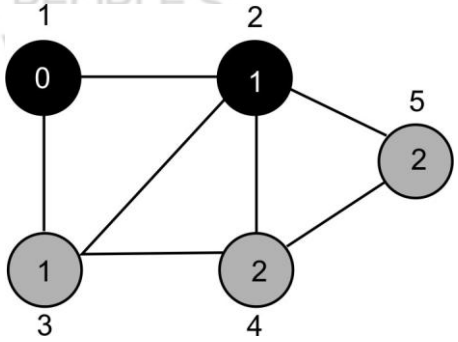
Initial status of the Queue is Q = Ø

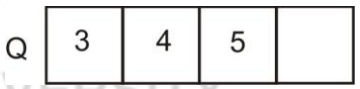


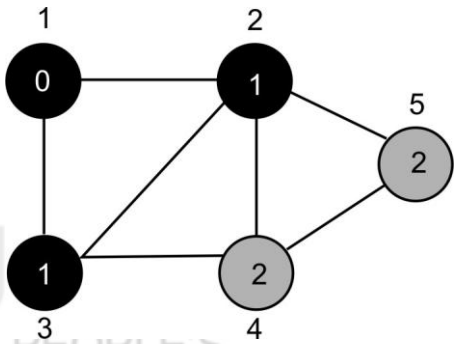



1
2
0
1
5
∞
1
∞
3
4

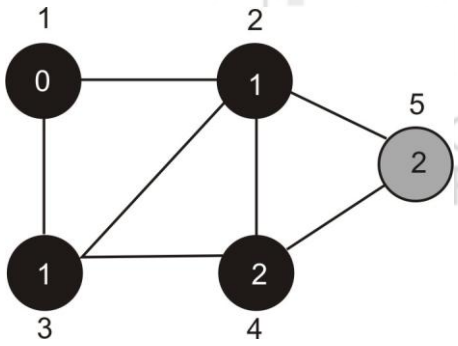

Q
2
3

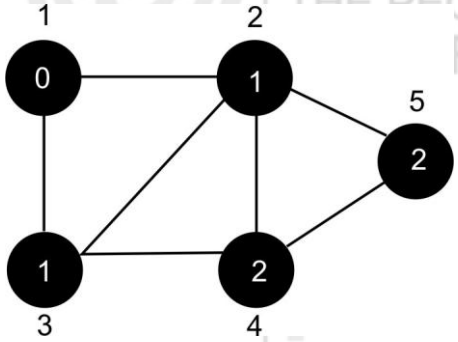

1
2
0
1
5
2
1
2
3
4


Q
3
4
5


1
2
0
1
5
2
1
2
3
4


Q
4
5





Q = Ø

After exploring the vertex of given graph by BFS algorithm, BFS traversal sequence is

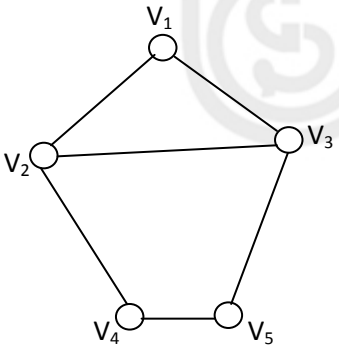
1, 2, 3, 4, 5

In this algorithm sequence of vertex visited or explored may vary. The final sequence of vertex visited is dependent on adjacency list. But the array d[] will have same number irrespective of order of vertices in adjacency list. In the above diagram distance is shown along the vertex. According to adjacency list drawn in the diagram, exploration sequence of vertex by BFS algorithm is 1,2,3,4,5.

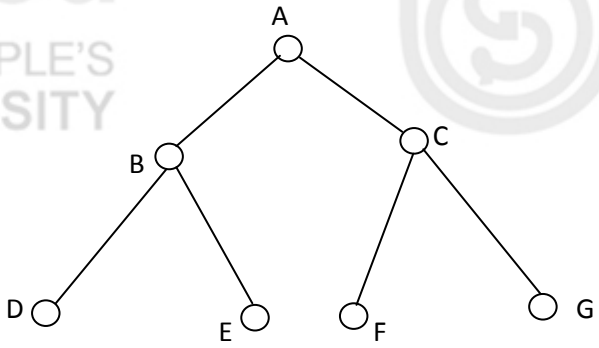
☞ Check Your Progress 1

1. What is the complexity of graph search algorithms if graph is represented by adjacency matrix and adjacency list?
2. Enlist few applications where DFS and BFS can be used?

3. Consider a graph with 5 vertices and 6 edges. Write its adjacency matrix and adjacency list.



4. For the following graph write DFS and BFS traversal sequence.



3.5 SUMMARY

A graph G(V,E) where V is the finite set of vertices i.e { v1,v2,v3….} and E is the finite set of edges {(u,v),(w,x)….}. Graph is known as directed graph if the each edge in the graph has ordered pair of vertices i.e (u,v) means an edge from u to v. In Undirected graph each edge is unordered pair of vertices i.e (u,v) and (v,u) refers to the same edge. A graph can be represented by adjacency matrix and adjacency list. In adjacency, list memory requirement is more as compared to adjacency list representation. Graph searching problem has wide range of applications. Breadth First search and Depth first search are very well known searching algorithms. In breadth first search, exploration of vertex is wider first then deeper. In depth first search it is deeper first and then it is widened. By exploration of vertex in any search algorithm, implies visiting or traversing each vertex in the graph. Data structure used for Breadth first search is queue and depth first search is stack. By using these search algorithms, connected components of graph can be found. Breadth first search method, gives shortest path between two vertices u and v. Depth first search is used in topological sorting. There are many more applications where these searching algorithms are used.

3.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1. For BFS algorithm complexity will be as follows:
   Adjacency Matrix – $O(V^2)$
   Adjacency List – $O(V+E)$

For DFS algorithm complexity is as follows:
Adjacency Matrix – $O(V^2)$
Adjacency List – O(V+E)

2. Application where DFS can be used:
   - Finding connected component of the graph
   - Finding shortest path between two vertices

Application where BFS can be used:

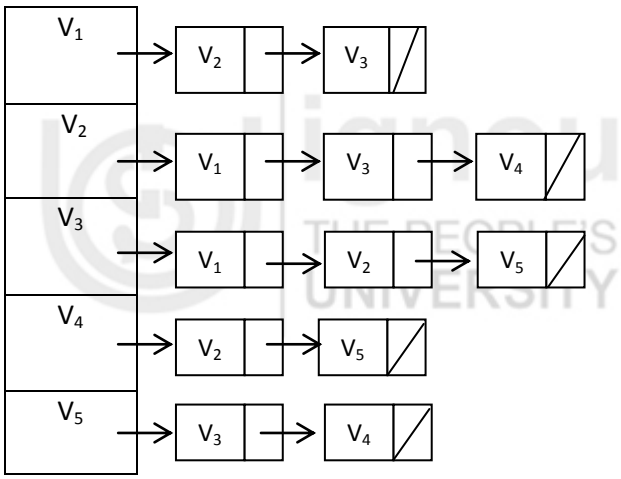
   - Finding connected component of the graph
   - Topological sorting
   - For finding cycle existence in the graph or not

3.

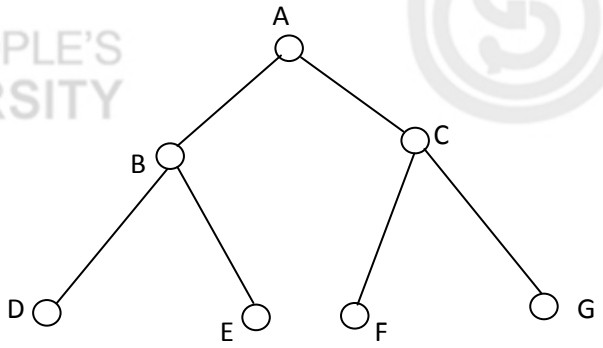
Adjacency Matrix

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	0	0
V_2	1	0	1	1	0
V_3	1	1	0	0	1
V_4	0	1	0	0	1
V_5	0	0	1	1	0

Adjacency List



4. For the given graph



BFS traversal sequence is A B C D E F G

DFS traversal sequence is A B D E C F G

3.7 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, "Introduction to Algorithms", 2 nd Ed., PHI, 2004.
2. Robert Sedgewick, "Algorithms in C", , Pearson Education, 3rd Edition 2004
3. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajasekaran, "Fundamentals of Computer algorithms", 2nd Edition, Universities Press, 2008
4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education, 2003.