UNIT 1 ELEMEMTARY ALGORITHMICS

Structure **Page Nos.**

1.0 INTRODUCTION

We are constantly involved in solving problem. The problems may concern our survival in a competitive and hostile environment, may concern our curiosity to know more and more of various facets of nature or may be about any other issues of interest to us. **Problem** may be a *state of mind* of a living being, of not being satisfied with some situation. However, for our purpose, we may take the unsatisfactory/ unacceptable/ undesirable *situation itself*, as a problem.

One way of looking at a possible **solution** of a problem, is as a **sequence of activities** (*if such a sequence exists at all*), that if carried out using allowed/available **tools**, leads us from the unsatisfactory (**initial) position** to an acceptable, satisfactory or **desired position**. For example, the solution of the problem of baking **delicious pudding** may be thought of as a **sequence of activities**, that when carried out, gives us the pudding (***the desired state***) **from the raw materials** that may include sugar, flour and water (***constituting the initial position***)**using** cooking gas, oven and some utensils etc. (***the tools***). The sequence of activities when carried out gives rise to a **process**.

Technically, the **statement or description** in some notation, of the process is called an **algorithm,** the raw materials are called the **inputs** and the **resulting entity** (in the above case, the pudding) is called the **output**. In view of the importance of the concept of algorithm, we repeat:

An **algorithm** is a ***description or statement*** of a sequence of activities that constitute a process of getting the desired outputs from the given inputs.

Later we consider in detail the characteristics features of an algorithm. Next, we define a closely related concept of computer program.

Two ideas lie gleaming on the jeweller's velvet. The first is the calculus; the second, the algorithm. **The calculus** and the rich body of mathematical analysis to which it gave rise **made modern science possible**; but it has been the **algorithm** that has **made possible** the **modern world**.

David Berlinski in The Advent of the Algorithm, 2000.

Computer Program: *An **algorithm**, when expressed in a notation that can be **understood** and **executed by a computer system** is called a computer program or simply a program.*We should be clear about the **distinction between the terms viz., a process, a program and an algorithm**.

A **process** is a sequence of activities **actually being carried out or executed,** to solve a problem. But **algorithm** and **programs** are just ***descriptions*** of a **process** in some notation. Further, a **program** is an **algorithm** in a notation that can be understood and be executed by a computer system.

It may be noted that for some problems and the available tools, there **may not exist any algorithm** that should give the desired output. For example, the problem of baking delicious pudding may not be solvable, if no cooking gas or any other heating substance is available. Similarly, the problem of reaching the moon is **unsolvable**, if no spaceship is available for the purpose.

These examples also highlight **the significance of available tools** in solving a problem. Later, we discuss some of mathematical problems which are not solvable. But, again these problems are said to be *unsolvable*, because of the fact that the operations (i.e., the tools) that are allowed to be used in solving the problems, are from a restricted pre-assigned set.

Notation for expressing algorithms

This issue of notation for representations of algorithms will be discussed in some detail, later. However, mainly, some combinations of mathematical symbols, English phrases and sentences, and some sort of pseudo-high-level language notations, shall be used for the purpose.

Particularly, **the symbol '←' is used for *assignment***. For example, x←y + 3, means that 3 is added to the value of the variable y and the resultant value becomes the new value of the variable x. However, the value of y remains unchanged.

If in an algorithm, more than one variables are required to store values of the same type, notation of the form A[1..n] is used to denote n variables A[1], A[2], … A[n].

In general, for the integers m, n with m ≤ n, A [m..n] is used to denote the variables A[m], A[m+1], …, A[n]. However, we must note that another similar notation A[m, n] is used to indicate the element of the matrix (or two-dimensional array) A, which is in m^{th} row and n^{th} column.

Role and Notation for Comments

The *comments* do not form that part of an algorithm, corresponding to which there is an (executable) action in the process. However, the comments help the human reader of the algorithm to better understand the algorithm. In different programming languages, there are different notations for incorporating comments in algorithms. We use the convention of *putting comments between pair of braces, i.e., **{ }***. The comments may be inserted at any place within an algorithm. For example, if an algorithm finds roots of a quadratic equation, then we may add the following comments, somewhere in the beginning of the algorithm, to tell what the algorithm does:

{*this algorithm finds the roots of a quadratic equation in which the coefficient of x^2 is assumed to be non-zero*}.

Section 1.2 explains some of the involved ideas through an example.

Mathematical Notations shall be introduced in Section 2.2.

1.1 OBJECTIVES

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

- explain the concepts: problem, solution, instance of a problem, algorithm, computer program;
- tell characteristics of an algorithm;
- tell the role of available tools in solving a problem;
- tell the basic instructions and control structures that are used in building up programs, and
- explain how a problem may be analyzed in order to find out its characteristics so as to help us in designing its solution/algorithm.

1.2 EXAMPLE OF AN ALGORITHM

Before going into the details of problem-solving with algorithms, just to have an idea of what an algorithm is, we consider a well-known algorithm for finding Greatest Common Divisor (G.C.D) of two natural numbers and also mention some related historical facts. First, the algorithm is expressed in English. Then, we express the algorithm in a notation resembling a programming language.

Euclid's Algorithm for Finding G.C.D. of two Natural Numbers m & n:

E1. {*Find Remainder*}. Divide m by n and let r be the (new) remainder {e have $0 \leq r < n$}

E2. {*Is r zero?*} If r = 0, the algorithm terminates and n is the answer. Otherwise,

E3. {*Interchange*}. Let the new value of m be the current value of n and the new value of n be the current value of r. Go back to Step E1.

The termination of the above method is guaranteed, as m and n must reduce in each iteration and r must become zero in finite number of repetitions of steps E1, E2 and E3.

The great Greek mathematician *Euclid* sometimes between fourth and third century BC, at least knew and may be the first to suggest, the above algorithm. The algorithm is considered as among the first non-trivial algorithms. However, the word '*algorithm*' itself came into usage quite late. The word is derived from the name of the Persian mathematician *Mohammed al-Khwarizmi* who lived during the ninth century A.D. The word '*al-khowarizmi*' when written in Latin became '*Algorismus*', from which '*algorithm*' is a small step away.

In order to familiarise ourselves with the notation usually used to express algorithms, next, we express the Euclid's Algorithm in a pseudo-code notation which is closer to a programming language.

Algorithm GCD-Euclid (m, n)

```
{This algorithm computes the greatest common divisor of two given positive
    integers}
    begin {of algorithm}
      while n ≠ 0 do
       begin {of while loop}
            r ← m mod n;
    {a new variable is used to store the remainder which is obtained by dividing
    m by n, with 0≤ r < m}
```

```
        m ←n;
{the value of n is assigned as new value of m; but at this stage value of n
remains unchanged}
        m← r;
{the value of r becomes the new value of n and the value of r remains
unchanged}
    end {of while loop}
        return (n).
end;   {of algorithm}
```

1.3 PROBLEMS AND INSTANCES

The difference between the two concepts viz., '*problem*' and '*instance*', can be understood in terms of the following example. An instance of a problem is also called a *question*. We know that the roots of a *general* quadratic equation

$$ax^2 + bx + c = 0 \qquad a \neq 0 \qquad (1.3.1)$$

are given by the equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \qquad (1.3.2)$$

where a, b, c may be *any* real numbers except the restriction that $a \neq 0$.

Now, if we take a = 3, b = 4 and c = 1,

we get the *particular* equation

$$3x^2 + 4x + 1 = 0 \qquad (1.3.3)$$

Using (1.2.2), the roots of (1.2.3) are given by

$$\frac{-4 \pm \sqrt{4^2 - 4 \times 3 \times 1}}{2 \times 3} = \frac{-4 \pm 2}{6} \quad \text{, i.e.,}$$

$$x = \frac{-1}{3} \text{ or } -1.$$

With reference to the above discussion, the **issue** of finding roots of the **general** quadratic equation $ax^2 + bx + c = 0$, with $a \neq 0$ is called a *problem*, whereas the **issue** of finding the roots of the **particular** equation

$$3x^2 + 4x+1 = 0$$

is called *a question or an instance of the (general) problem*.

In general, a problem may have a large, possibly infinite, number of instances. The above-mentioned *problem* of finding the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

with $a \neq 0$, b and c as real numbers, has *infinitely* many *instances*, each obtained by giving some specific real values to a, b and c, taking care that the value assigned to *a* is not zero. However, all problems may not be of generic nature. For *some problems*, *there may be only one instance/question* corresponding to each of the problems. For

example, the problem of finding out the largest integer that can be stored or can be arithmetically operated on, in a given computer, is a single-instance problem. Many of the interesting problems like the ones given below, are just **single-instance problems**.

Problem (i): Crossing the river in a boat which can carry at one time, alongwith the boatman only one of a wolf, a horse and a bundle of grass, in such a way that neither wolf harms horse nor horse eats grass. In the presence of the boatman, neither wolf attacks horse, nor horse attempts to eat grass.

Problem (ii): The Four-Colour Problem* which requires us to find out whether a political map of the world, can be drawn using only four colours, so that no two adjacent countries get the same colour.

> The problem may be further understood through the following explanation. Suppose we are preparing a coloured map of the world and we use *green* colour for the terrestrial part of India. Another country is a neighbour of a given country if it has some boundary in common with it. For example, according to this definition, Pakistan, Bangladesh and Myanmar (or Burma) are some of the countries which are India's neighbours. Then, in the map, for all the neighbour's of India, including Pakistan, Bangladesh and Myanmar, we *can not use green colour*. The problem is to show that the minimum number of colours required is four, so that we are able to colour the map of the world under the restrictions of the problem.

Problem (iii): The Fermat's Last Theorem: which requires us to show that there do not exist positive integers a, b, c and n such that

$$a^n + b^n = c^n \quad \text{with } n \geq 3.$$

The problem also has a very fascinating history. Its origin lies in the simple observation that the equation

$$x^2 + y^2 = z^2$$

has a number of solutions in which x, y and z all are integers. For example, for x = 3, y = 4, z = 5, the equation is satisfied. The fact was also noticed by the great mathematician Pierre De *Fermat* (*1601 – 1665*). But, like all great intellectuals, he looked at the problem from a different perspective. Fermat felt and claimed that for all integers $n \geq 3$, the equation

$$x^n + y^n = z^n$$

has no non-trivial[!] solution in which x, y and z are all positive integers. And he jotted down the above claim in a corner of a book without any details of the proof.

However, for more than next 300 years, mathematicians could not produce any convincing proof of the Fermat's the-then conjecture, and now a theorem. Ultimately, the proof was given by Andrew Wiles in 1994. Again the proof is based not only on a very long computer program but also on sophisticated modern mathematics.

Problem (iv): On the basis of another generalisation of the problem of finding integral solutions of $x^2+y^2 = z^2$, great Swiss mathematician Leonhard Euler conjectured that for $n \geq 3$, the sum of $(n - 1)$

---

* The origin of the Four-colour conjecture, may be traced to the observation by Francis Guthrie, a student of Augustus De Morgan (*of De Morgan's Theorem fame*), who noticed that all the counties (sort of parliamentary constituencies in our country) of England could be coloured using four colours so that no adjacent counties were assigned the same colour. De Morgan publicised the problem throughout the mathematical community. Leaving aside the problem of *parallel postulate* and the problem in respect of *Fermat's Last Theorem*, perhaps, this problem has been the most fascinating and tantalising one for the mathematicians, remaining unsolved for more than one hundred years. Ultimately, the problem was solved in 1976 by two American mathematician, Kenneth Appel and Wolfgang Haken.

However, the proof is based on a computer program written for the purpose, that took 1000 hours of computer time (in 1976). Hence, the solution generated, among mathematicians, a controversy in the sense that many mathematicians feel such a long program requiring 1000 hours of computer time in execution, may have logical and other bugs and hence can not be a reliable basis for the *proof* of a conjecture.

[!] one solution, of course, is given by x = 0 = y = z, though x, y and z, being zero, are not positive.

number of nth powers of positive integers can not be an nth power of an integer. For a long time the conjecture neither could be refuted nor proved. However, in 1966, L.J. Lander and T.R. Parkin found a counter example for n = 5, by showing that $27^5 + 84^5 + 110^5 + 133^5 = 144^5$.

Coming back to the problem of finding the roots of a quadratic equation, it can be easily seen that in finding the roots of a quadratic equation, the only *operations* that have been used are *plus, minus, multiplication and division* of numbers alongwith the *operation of finding out the square root of a number*. Using only these operations, it is also possible, ***through step-by-step method***, to find the roots of a **cubic equation** over the real numbers, which, in general, is of the form

$$ax^3 + bx^2 + cx + d = 0,$$

where $a \neq 0$, b, c and d are real numbers.

Further, using only the set of operations mentioned above, it is also possible, ***through a step-by-step method***, to solve a **biquadratic equation** over real numbers, which, in general, is of the form

$$ax^4 + bx^3 + cx^2 + dx + e = 0,$$

where $a \neq 0$, b, c, d and e are real numbers.

However, the problem of finding the **roots** of a general **equation of degree five or more**, **can not** be solved, using only the operations mentioned above, ***through a step-by-step method***, i.e., can not be solved **algorithmically**.

In such cases, we may attempt some *non-algorithmic methods* including solutions based on numerical methods which may not give exact but some good approximate solutions to such problems. Or we may just use some **hit and trial method, e.g., consisting** of guessing a possible root and then verifying the guess as a possible solution, by actually substituting the guessed root in the given equation. A **hit and trial method is not an algorithm**, because we **cannot guarantee the termination** of the method, where as discussed later, termination is one of the characteristic properties of an algorithm.

It may be noted that a (*general*) *problem*, like finding the roots of an equation of degree 5 or more, may not be solvable algorithmically, i.e., through some step-by-step method, still it is possible for some (particular) *instances* of the problem to have algorithmic solutions. For example, the roots of the equation

$$x^5 - 32 = 0$$

are easily available through a step-by-step method. Also, the roots of the equation $2x^6 - 3x^3 + 1 = 0$ can be easily found through a method, in which, to begin with, we may take $y = x^3$

Ex. 1) Give at least three examples of problems, each one of which has only *finitely* many instances.

Hint: Structures over *Boolean set {0, 1} may be good sources for such examples*.

1.4 CHARACTERISTICS OF AN ALGORITHM

Next, we consider the concept of algorithm in more detail. While designing an algorithm as a solution to a given problem, we must take care of the following ***five important characteristics of an algorithm***:

1. **Finiteness:** An algorithm must terminate after a **finite number** of steps and further each step must be executable in **finite amount of time**. In order to establish a sequence of steps as an algorithm, it should be established that it **terminates** (in finite number of steps) on *all allowed* inputs.

2. **Definiteness**[*](*no ambiguity*)**:** Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case. Through the next example, we show how an instruction may *not* be definite.

Example 1.4.1: Method which is effective (*to be explained later)* but not definite.
The following is a program fragment for the example method:

x ← *1*
Toss a coin,
If *the result is Head* ***then*** *x* ←*3* ***else*** *x* ← *4*

{*in the above, the symbol '←' denotes that the value on its R.H.S is assigned to the variable on its L.H.S. Detailed discussion under (i) of Section 1.6.1}*

All the steps, like tossing the coin etc., can be (effectively) carried out. However, the method is *not definite*, as two different executions *may yield different outputs*.

3. **Inputs:** An algorithm **has zero or more, but only finite,** number of inputs. Examples of algorithms requiring *zero* inputs:

   (i) Print the largest integer, say MAX, representable in the computer system being used.

   (ii) Print the ASCII code of each of the letter in the alphabet of the computer system being used.

   (iii) Find the sum *S* of the form *1+2+3+…*, where *S* is the largest integer less than or equal to MAX defined in Example (i) above.

4. **Output:** An algorithm has **one or more outputs**. The requirement of at least one output is obviously essential, because, otherwise we can not know the answer/solution provided by the algorithm.

   The outputs have specific relation to the inputs, where the relation is defined by the algorithm.

5. **Effectiveness:** An algorithm should be effective. This means that **each of the operation** to be performed in an algorithm **must be sufficiently basic** that it can, in principle, **be done exactly** and in a **finite length of time,** by a person using pencil and paper. *It may be noted that the 'FINITENESS' condition is a special case of 'EFFECTIVENESS'. If a sequence of steps is not finite, then it can not be effective also.*

***A method may be designed which is a definite sequence of actions but is <u>not</u> finite* (*and hence not effective*)**

Example 1.4.2: If the following instruction is a part of an algorithm:
Find exact value of e using the following formula

[*] There are some methods, which are not definite, but still called algorithms viz., *Monte Carlo algorithms* in particular and *probabilistic algorithms* in general. However, we restrict our algorithms to those methods which are *definite* alongwith other four characteristics. In other cases, the full name of the method viz., *probabilistic algorithm*, is used.

$e = 1 + 1/(1!) + 1/(2!) + 1/(3!) + \ldots..$

and add it to x.

Then, the algorithm is *not effective*, because as per instruction, computation of e requires computation of *infinitely many* terms of the form *1/n*! for n = 1, 2, 3, ….., which is not possible/effective.

However, the instruction is *definite* as it is easily seen that computation of each of the term *1/n*! is definite (at least for a given machine).

Ex. 2) For each of the following, give one example of a method, which is not an algorithm, because

(i) the method is not finite
(ii) the method is not definite
(iii) the method is not effective but finite.

1.5 PROBLEMS, AVAILABLE TOOLS & ALGORITHMS

In order to explain an important point in respect of the *available tools*, of which one must take care while designing an algorithm for a given problem, we consider some **alternative algorithms for finding the product m*n** of two natural numbers m and n.

First Algorithm:

The **usual method** of multiplication, in which table of products of pair of digits x, y (i.e.; $0 \leq x, y \leq 9$) are presumed to be available to the system that is required to compute the product m*n.

For example, the product of two numbers 426 and 37 can be obtained as shown below, using multiplication tables for numbers from 0 to 9.

```
   4 2 6
     3 7
  ------
   2 9 8 2
 1 2 7 8 0
 ---------
 1 5 7 6 2
```

Second Algorithm:

For this algorithm, we assume that the **only arithmetic capabilities** the system is endowed with, are

(i) *that of counting and*
(ii) *that of comparing two integers w.r.t. 'less than or equal to' relation.*

With only these two capabilities, the First Algorithm is meaningless.

For such a system having only these two capabilities, one possible algorithm to calculate m*n, as given below, uses two separate portions of a paper (or any other storage devices). One of the portions is used to accommodate marks upto **n, the multiplier,** and the other to accommodate marks upto m*n, **the resultant product**.

The algorithm constitutes the following steps:

***Step 1*:** Initially make a mark on First Portion of the paper.

Step 2*:** For each new mark on the First Portion, ***make m new marks on the Second Portion.

Step 3*:** Count the number of marks in First Portion. ***If the count equals n, then count the number of all marks in the Second Portion and return the last count as the result. ***However, if*** the count in the First Portion is less than n, then make one more mark in the First Portion and go to Step 2.

Third Algorithm:

The algorithm to be discussed, is known *a'**la russe method***. In this method, it is presumed that the system has **the** *only* **capability of multiplying and dividing any integer by 2, in** addition to the capabilities of Second Algorithm. The division must result in an integer as quotient, with remainder as either a 0 or 1.

The algorithm using only these capabilities for multiplying two positive integers m and n, is based on the observations that

(i) If m is even then if we divide m by 2 to get (m/2) and multiply n by 2 to get (2n) then (m/2) . (2 n) = m . n.

Hence, by halving successive values of m (or (m − 1) when m is odd as explained below), we expect to reduce m to zero ultimately and stop, without affecting at any stage, the required product by doubling successive values of n alongwith some other modifications, if required.

(ii) However, if *m is odd* then (m/2) is not an integer. In this case, we write m = (m − 1) + 1, so that *(m − 1) is even* and (m − 1)/2 is an integer.

Then

$$m \cdot n = ((m-1)+1) \cdot n = (m-1)n + n$$
$$= ((m-1)/2) \cdot (2n) + n.$$

where (m − 1)/2 is an integer as m is an odd integer.

For example, m = 7 and n = 12

Then

$$m * n = 7 * 11 = ((7-1)+1) * 11 = (7-1) * 11 + 11$$
$$= \frac{(7-1)}{2} (2 * 11) + 11$$

Therefore, if at some stage, m is even, we halve m and double n and multiply the two numbers so obtained and repeat the process. But, if m is odd at some stage, then we halve (m – 1), double n and multiply the two numbers so obtained and then add to the product so obtained the odd value of m which we had before halving (m −1).

Next, we describe the *a'**la russe method***/algorithm.

The algorithm that **uses four variables**, viz., **First, Second, Remainder and Partial-Result,** may be described as follows:

***Step 1*:** Initialize the variables First, Second and Partial-Result respectively with m (the first given number), n (the second given number) and 0.

***Step 2*: If** First or Second* is zero, return Partial-result as the final result and then **stop**.

* If, initially, Second ≠ 0, then Second ≠ 0 in the subsequent calculations also.

Else, set the value of the **Remainder as 1** if First is odd, else set Remainder as 0. If Remainder is 1 then add Second to Partial-Result to get the new value of Partial Result.

***Step 3*:** New value of First is the quotient obtained on (integer) division of the current value of First by 2. New value of Second is obtained by multiplying Second by 2. Go to Step 2.

Example 1.5.1: The logic behind the a'la russe method, consisting of Step 1, Step 2 and Step 3 given above, may be better understood, in addition to the argument given the box above, through the following explanation:

Let First = 9 and Second = 16

Then First * Second = 9 * 16 = (4 * 2 + 1) * 16
= 4 * (2 * 16) + 1 * 16
where 4 = [9/2] = [first/2], 1 = Remainder.
Substituting the values back, we
first * second = [first/2] * (2 * Second) + Second.

Let us take $First_1$ = [First/2] = 4
$Second_1$ = 2 * Second = 32 and
$Partial\text{-}Result = First_1 * Second_1$.
Then from the above argument, we get
$First * Second = First_1 * Second_1 + Second$
$= Partial\text{-}Result_1 + Second$.

Here, we may note that as First = 9 is odd and hence Second is added to Partial-Result. Also
$Partial\text{-}Result_1 = 4*32 = (2 * 2 + 0) * 32 = (2 * 2) * 32 + 0 * 32$
$= 2* (2 * 32) = First_2 * Second_2$.

Again we may note that $First_1 = 4$ is even and we *do not* add $Second_2$ to $Partial\text{-}Result_2$, where $Partial\text{-}Result_2 = First_2 * Second_2$.

Next, we execute the a'la russe algorithm to compute 45 * 19.

	First	**Second**	**Remainder**	**Partial Result**
Initially:	45	19		0
Step 2	As value of First ≠ 0, hence continue		1	19
Step 3	22	38		
Step 2	Value of first ≠ 0, continue		0	
Step 3	11	76		
Step 2	Value of First ≠ 0, continue		1	76+19=95
Step 3	5	152		
Step 2	Value of first ≠ 0, continue		1	152+95=247
Step 3	2	304		
Step 2	Value of First ≠ 0, continue		0	
Step 3	0	608	1	608+247=855
Step 2				

As the value of the First is 0, the value 855 of Partial Result is returned as the result and stop.

Ex. 3) A system has ONLY the following arithmetic capabilities:

(i) that of counting,
(ii) that of comparing two integers w.r.t. 'less than or equal to' relation and
(iii) those of both multiplying and dividing by 2 as well as 3.

Design an algorithm that multiplies two integers, and fully exploits the capabilities of the system. Using the algorithm, find the product.

1.6 BUILDING BLOCKS OF ALGORITHMS

Next, we enumerate the **basic actions** and corresponding **instructions** used in a computer system based on a Von Neuman architecture. We may recall that an **instruction** is a **notation** for an action and a sequence of instructions defines a **program** whereas a sequence of **actions** constitutes a **process**. **An instruction is also called a statement.**

The following *three basic actions and corresponding instructions* form the basis of any imperative language. **For the purpose of explanations, the notation similar to that of a high-level programming language is used**.

1.6.1 Basic Actions & Instructions

(i) Assignment of a value to a variable is denoted by

variable ← *expression*;

where the expression is composed from variable and constant operands using familiar operators like +, -, * etc.

Assignment action includes evaluation of the expression on the R.H.S. An example of assignment instruction/statement is

$$j \leftarrow 2 * i + j - r ;$$

It is assumed that each of the variables occurring on R.H.S. of the above statement, has a value *associated* with it before the execution of the above statement. The association of a value to a variable, whether occurring on L.H.S or on R.H.S, is made according to the following rule:

For each variable name, say i, there is a unique location, say loc 1 (i), in the main memory. Each location loc(i), at any point of time contains a unique value say v(i). Thus the value v(i) is associated to variable i.

Using these values, the expression on R.H.S. is evaluated. The value so obtained is the new value of the variable on L.H.S. This value is then stored as a new value of the variable (*in this case, j*) on L.H.S. It may be noted that the variable on L.H.S (*in this case, j*) may also occur on R.H.S of the assignment symbol.

In such cases, the value corresponding to the occurrence on R.H.S (*of j, in this case*) is finally replaced by a new value obtained by evaluating the expression on R.H.S (*in this case, 2 * i + j − r*).

The values of the other variables, viz., i and r remain unchanged due to assignment statement.

(ii) The next basic action is to read values of variables i, j, etc. **from some secondary storage device, the identity of which is (implicitly) assumed here,** by a statement of the form

read (i,j, ,…);

The values corresponding to variables i, j,… in the read statement, are, due to read statement, stored in the corresponding locations loc(i) , loc(j),…, in the main memory. The values are supplied either, by default, through the keyboard by the user or from some secondary or external storage. In the latter case, the identity of the secondary or external storage is also specified in the read statement.

(iii) The last of the three basic actions, is to deliver/write values of some variables say i, j, etc. to the monitor or to an external secondary storage by a statement of the form

write (i, j ,….);

The *values* in the locations loc(i), loc(j),…, corresponding to the variables i, j …, in the write statement are copied to the monitor or a secondary storage. Generally, values are written to the monitor by default. In case, the values are to be written to a secondary storage, then identity of the secondary storage is also specified in the write statement. Further, if the argument in a write statement is some sequence of characters enclosed within quotes then the sequence of characters as such, but without quotes, is given as output. For example, corresponding to the write statement.

Write (*'This equation has no real roots'*)

the algorithm gives the following output:

This equation has no real roots.

In addition to the types of instructions corresponding to the above mentioned actions, there are other non-executable instructions which include the ones that are used to define the structure of the data to be used in an algorithm. These issues shall be discussed latter.

1.6.2 Control Mechanisms and Control Structures

In order to understand and to express an algorithm for solving a problem, it is not enough to know just the *basic actions viz., assignments, reads and writes*. In addition, we must know and understand the control mechanisms. These are the mechanisms by which the human beings and the executing system become aware of the next instruction to be executed after finishing the one currently in execution. *The sequence of execution of instructions need not be the same as the sequence in which the instructions occur in program text*. First, we consider three basic control mechanisms or structuring rules, before considering more complicated ones.

(i) Direct Sequencing: When the sequence of execution of instructions is to be the same as the sequence in which the instruction are written in program text, the control mechanism is called **direct sequencing**. Control structure, (i.e., the *notation* for the control mechanism), for direct sequencing is obtained by **writing of the instructions,**

- **one after the other on successive lines, or even on the some line if there is enough space on a line,** and
- separated by some statement separator, say **semi-colons**, and
- in the order of intended execution.

For example, the sequence of the three next lines

A; B;
C;
D;

denotes that the execution of A is to be followed by execution of B, to be followed by execution of C and finally by that of D.

When the composite action consisting of actions denoted by A, B, C and D, in this order, is to be treated as a single component of some larger structure, brackets such as '*begin….end*' may be introduced, i.e., in this case we may use the structure

Begin A;B;C;D end.

Then the above is also called a (composite/compound) statement consisting of four (component) statement viz A, B, C and D.

(ii) Selection: In many situations, we intend to carry out some action *A* if condition *Q* is satisfied and some other action *B* if condition *Q* is not satisfied. This intention can be denoted by:

If Q then do A else do B,

Where A and B are instructions, which may be even composite instructions obtained by applying these structuring rules recursively to the other instructions.

Further, in some situations the action *B* is null, i.e., if *Q* is false, then no action is stated.

This new situation may be denoted by

If Q then do A

In this case, if *Q* is true, A is executed. If *Q* is not true, then the remaining part of the instruction is ignored, and the next instruction, if any, in the program is considered for execution.

Also, there are situations when Q is not just a Boolean variable i.e., a variable which can assume either a *true* or a *false* value *only*. Rather *Q* is some variable capable of assuming some finite number of values say *a, b, c, d, e, f.* Further, suppose depending upon the value of *Q*, the corresponding intended action is as given by the following table:

Value	**Action**
a	*A*
b	*A*
c	*B*
d	*NO ACTION*
e	*D*
f	*NO ACTION*

The above intention can be expressed through the following notation:

```
Case Q of
        a, b :  A;
          c  :  B;
         e : D;
end;
```

Example 1.6.2.1: We are to write a program segment that converts % of marks to grades as follows:

% of marks (M)	**grade (G)**
$80 \leq M$	A
$60 \leq M < 80$	B

$50 \le M < 60$	C
$40 \le M < 50$	D
$M < 40$	F

Then the corresponding notation may be:

```
Case M of
80 . . 100 :  'A'
60 . . 79   :  'B'
50 . . 59  :  'C'
40 . . 49  :  'D'
0   . . 39  :  'F'
```

where M is an integer variable

(iii) Repetition: Iterative or repetitive execution of a sequence of actions, is the basis of expressing *long processes* by comparatively *small number of instructions*. As we deal with only finite processes, therefore, the repeated execution of the sequence of actions, has to be terminated. The termination may be achieved either through some condition Q or by stating in advance the number of times the sequence is intended to be executed.

(a) When we intend to execute a sequence S of actions repeatedly, while condition Q holds, the following notation may be used for the purpose:

While (Q) do begin S end;

Example 1.6.2.2: We are required to find out the sum (*SUM*) of first n natural numbers. Let a variable x be used to store an integer less than or equal to n, then the algorithm for the purpose may be of the form:

```
algorithm Sum_First_N_1
begin
read (n); {assuming value of n is an integer ≥ 1}
        x←1  ;  SUM← 1;
        while (x < n) do ……(α1)
        begin
                x ← x + 1;
                SUM ← SUM + x
        end; {of while loop}……  (β1)
        write ('The sum of the first', n, 'natural numbers is' SUM)
        end. {of algorithm}
```

Explanation of the algorithm Sum_First_N_1:

Initially, an integer value of the variable n is read. Just to simplify the argument, we assume that the integer $n \ge 1$. The next two statements assign value 1 to each of the variables x and SUM. Next, we come the execution of the while-loop. The while-loop extends from the statement $(\alpha 1)$ to $(\beta 1)$ both inclusive. Whenever we enter the loop, the condition x<n is (always) tested. If the condition x<n is true then the whole of the remaining portion upto β (inclusive) is executed. However, if the condition is false then all the remaining statement of the while-loop, i.e., all statements upto and including $(\beta 1)$ are skipped.

Suppose we read 3 as the value of n, and (initially) x equals 1, because of $x \leftarrow 1$. Therefore, as 1<3, therefore the condition x<n is true. Hence the following portion of the while loop is executed:

begin

$x \leftarrow x+1;$

$SUM \leftarrow SUM + x;$

end

and as a consequence of execution of this composite statement

the value of x becomes 1 and
and the value of SUM becomes 3

As soon as the word end is encountered by the meaning of the *while-loop*, the whole of the while-loop between $(\alpha 1)$ and $(\beta 1)$, (*including* $(\alpha 1)$ *and* (β1)) is again executed.

By our assumption n = 3 and it did not change since the initial assumption about n; however, x has become 2. Therefore, x<n is again satisfied. Again the rest of the while loop is executed. Because of the execution of the rest of the loop, x becomes 3 and SUM becomes the algorithm comes to the execution of first statement of while-loop, i.e., *while (x<n) do*, which tests whether x<n. At this stage x=3 and n=3. Therefore, x<n is false. Therefore, all statement upto and including (β_1) are x<n skipped.

Then the algorithm executes the next statement, viz., write ('The sum of the first', n, 'natural numbers is', sum). As, at this stage, the value of SUM is 6, the following statement is prompted, on the monitor:

The sum of the first 3 natural numbers is 6.

It may be noticed that in the statement *write* (' ', *n*, ' ', *SUM*) the variables n and SUM are not within the quotes and hence, the values of n and SUM viz 3 and 6 just before the write statement are given as output,

Some variations of the 'while…do' notation, are used in different programming languages. For example, if S is to be executed at least once, then the programming language C uses the following statement:

Do S while (Q)

Here S is called the body of the 'do. while' loop. It may be noted that here S is not surrounded by the brackets, viz., begin and end. It is because of the fact do and while enclose S.

Again consider the example given above, of finding out the sum of first n natural numbers. The using '*do … while*' statement, may be of the form:

The above instruction is denoted in the programming language Pascal by

Repeat S until (not Q)

Example 1.6.2.3: Algorithm Sum_First_N_2

Begin {*of algorithm*}
read (n);

```
x←0          ;          SUM← 0
      do                                          (α2)
                    x  ← x + 1
                    SUM← SUM + x

      while (x < n)  …….                          (β2)
end {of algorithm}.
```

If number n of the times S is to be executed is known in advance, the following notation may be used:

for v varying from i to (i+n–1) do begin S end;

OR

for v ← i to (i + −1) do

begin S end;

where v is some integer variable assuming initial value i and increasing by 1 after each execution of S and final execution of S takes place after v assumes the value i+n–1.

Then the execution of the for-loop is terminated. Again *begin S do* is called the body of the *for-loop*. The variable x is called *index variable* of the for-loop.

Example 1.6.2.4: Again, consider the problem of finding the sum of first n natural numbers, algorithm using '*for* …' may be of the form:

algorithm Sum_First_N_3

```
begin
      read (n);

        SUM← 0
        for x ← 1 to n do                         (α 3)
          begin
            SUM ← SUM + x                         (β 3)
         end;
      write ('The sum of the first first', n, natural numbers numbers', SUM)
end. {of the algorithm}
```

In the algorithm Sum_First_N_3 there is *only* one statement in the body of the for-loop. Therefore, the bracket words *begin* and *end* may not be used in the for-loop. In this algorithm, also, it may be noted that only the variable SUM is initialized. The variable x is not initialized explicitly. The variable x is implicitly initialised to 1 through the construct '*for x varying from 1 to n do*'. And, after each execution of the body of the *for-loop*, x is implicitly incremented by 1.

A noticeable feature of the constructs (structuring rules) viz., sequencing, selection and iteration, is that each defines a control structure with a single entry point and single exit point. It is this property that makes them simple but powerful building blocks for more complex control structures, and ensures that the resultant structure remains comprehensible to us.

Structured Programming, a programming style, allows only those structuring rules which follow '*single entry, single exit*' rule.

Ex.4) Write an algorithm that finds the *real* roots, if any, of a quadratic equation $ax^2 + bx + c = 0$ with $a \neq 0$, b, c as real numbers.

Ex.5) Extend your algorithm of Ex. 4 above *to find roots* of equations of the form $ax^2 + bx + c = 0$, in which a, b, c may be arbitrary real numbers, including 0.

Ex.6) (i) Explain how the algorithm Sum_First_N_2 finds the sum of the first 3 natural numbers.

(ii) Explain how the algorithm Sum_First_N_3 finds the sum of the first 3 natural numbers.

1.6.3 Procedure and Recursion

Though the above-mentioned three control structures, viz., direct sequencing, selection and repetition, are sufficient to express any algorithm, yet the following *two advanced control structures* have proved to be quite useful in facilitating the expression of complex algorithms viz.

(i) Procedure
(ii) Recursion

Let us first take the advanced control structure *Procedure*.

1.6.3.1 Procedure

Among a number of terms that are used, in stead of *procedure,* are *subprogram* and even *function*. These terms may have shades of differences in their usage in different programming languages. However, the basic idea behind these terms is the same, and is explained next.

It may happen that a sequence frequently occurs either in the same algorithm repeatedly in different parts of the algorithm or may occur in different algorithms. In such cases, writing repeatedly of the same sequence, is a wasteful activity. *Procedure* is a mechanism that provides a method of checking this wastage.

Under this mechanism, the sequence of instructions expected to be repeatedly used in one or more algorithms, is written only once and outside and independent of the algorithms of which the sequence could have been a part otherwise. There may be many such sequences and hence, there is need for an identification of each of such sequences. For this purpose, each sequence is prefaced by statements in the following format:

Procedure <Name> (<parameter-list>) [*: < type>*]
    <declarations>
    <sequence of instructions expected to be occur repeatedly>
end; (1.6.3.1)

where *<name>, <parameter-list>* and other expressions with in the angular brackets as first and last symbols, are *place-holders* for suitable values that are to be substituted in their places. **For example**, suppose finding the sum of squares of two variables is a frequently required activity, then we may write the code for this activity independent of the algorithm of which it would otherwise have formed a part. And then, in (1.6.3.1), *<name>* may be replaced by '*sum-square*' and *<parameter-list>* by the two-element sequence *x, y*. The variables like x when used in the definition of an algorithm, are called **formal parameters or simply parameters**. Further, whenever the code which now forms a part of a procedure, say *sum-square* is required at any place in an algorithm, then in place of the intended code, a statement of the form

sum-square (a, b); (1.6.3.2)

is written, where values of *a and b* are defined before the location of the statement under (1.6.3.2) within the algorithm.

Further, the pair of brackets in [: < type >] indicates that ': < type >' is optional. If the procedure passes some value computed by it to the calling program, then ': < type >' is used and then <type> in (1.6.3.1) is replaced by the type of the value to be passed, in this case *integer*.

In cases of procedures which pass a value to the calling program another basic construct (in addition to *assignment, read and write*) viz., *return (x)* is used, where x is a variable used for the value to be passed by the procedure.

There are various mechanisms by which values of *a and b* are respectively associated with or transferred to *x and y*. *The variables like a and b*, defined in the calling algorithm to pass data to the procedure (*i.e., the called algorithm*), which the procedure may use in solving the particular instance of the problem, are called **actual parameters** or **arguments**.

Also, there are different mechanisms by which *statement* of the form *sum-square* (*a, b*) of an algorithm is associated with the *code* of the procedure for which the statement is used. However, all these mechanisms are named as '*calling the procedure*'. The main algorithm may be called as the '***calling algorithm***' and the procedure may be called '***the called algorithm'***. The discussion of these mechanisms may be found in any book on concepts of programming languages*.

In order to explain the involved ideas, let us consider the following simple examples of a procedure and a program that calls the procedure. In order to simplify the discussion, in the following, we assume that the inputs etc., are always of the required types only, and make other simplifying assumptions.

Example 1.6.3.1.1

Procedure sum-square (a, b : integer) : integer;

{denotes the inputs a and b are integers and the output is also an integer}
S: integer;
{to store the required number}
begin
S ← $a^2 + b^2$
Return (S)
end;

Program Diagonal-Length

{the program finds lengths of diagonals of the sides of right-angled triangles whose lengths are given as integers. The program terminates when the length of any side is not positive integer}

L1, L2: integer; *{given side lengths}*
D: real;
{to store diagonal length}
read (L_1, L_2)
while ($L_1 > 0$ and $L_2 > 0$) do

begin
D← square–root (sum-square (L1, L2))
write ('For sides of given lengths', L_1, L_2, 'the required diagonal length is' D);
read (L_1, L_2);
end.

* For the purpose Ravi Sethi (1996) may be consulted.

In order to explain, how diagonal length of a right-angled triangle is computed by the program *Diagonal-Length* using the procedure *sum-square*, let us consider the side lengths being given as 4 and 5.

First Step: In program *Diagonal-Length* through the statement read (L_1, L_2), we read L_1 as 4 and L_2 as 5. As $L_1 > 0$ and $L_2 > 0$. Therefore, the program enters the while-loop. Next the program, in order to compute the value of the diagonal calls the procedure *sum-square* by associating with ***a*** the value of L_1 as 4 and with ***b*** the value of L_2 as 5. After these associations, the procedure sum-square takes control of the computations. The procedure computes S as 41 = 16 + 25. The procedure returns 41 to the program. At this point, the program again takes control of further execution. The program uses the value 41 in place of sum-square (L_1, L_2). The program calls the procedure *square-root*, which is supposed to be built in the computer system, which temporarily takes control of execution. The procedure *square-root* returns value $\sqrt{41}$ and also returns control of execution to the program *Diagonal-Length* which in turn assigns this value to D and prints the statement:

For sides of given lengths 4 and 5, the required diagonal length is $\sqrt{41}$**.**

The program under while-loop again expects values of L_1 and L_2 from the user. If the values supplied by the user are positive integers, whole process is repeated after entering the while-loop. However, if either $L_1 \leq 0$ (say -34) or $L_2 \leq 0$, then while-loop is not entered and the program terminates.

We summarise the above discussion about procedure as follows:

A procedure is a self-contained algorithm which is written for the purpose of plugging into another algorithm, but is written independent of the algorithm into which it may be plugged.

1.6.3.2 Recursion

Next, we consider another important control structure namely recursion. In order to facilitate the discussion, we recall from Mathematics, one of the ways in which the factorial of a natural number n is defined:

factorial (1) = 1
factorial (n) = n* factorial (n−1). (1.6.3.2.1)

For those who are familiar with recursive definitions like the one given above for factorial, it is easy to understand how the value of (n!) is obtained from the above definition of factorial of a natural number. However, for those who are not familiar with recursive definitions, let us compute factorial (4) using the above definition.
By definition
factorial (4) = 4 * factorial (3).
Again by the definition
factorial (3) = 3 * factorial (2)
Similarly
factorial (2) = 2* factorial (1)
And by definition
factorial (1) = 1
Substituting back values of factorial (1), factorial (2) etc., we get
factorial (4) = 4.3.2.1=24, as desired.

This definition suggests the following procedure/algorithm for computing the factorial of a natural number n:

In the following procedure factorial (n), let *fact* be the variable which is used to pass the value by the procedure *factorial* to a calling program. The variable fact is initially assigned value 1, which is the value of factorial (1).

```
Procedure factorial (n)
        fact: integer;
  begin
     fact ← 1
    if n equals 1 then return fact
    else begin
    fact ← n * factorial (n − 1)
    return (fact)
   end;
end;
```

In order to compute *factorial (n − 1)*, procedure *factorial* is called by itself, but this time with (simpler) argument (n − 1). The repeated calls with simpler arguments continue until factorial is called with argument 1. Successive multiplications of partial results with 2,3, ….. upto n finally deliver the desired result.

Though, it is already mentioned, yet in view of the significance of the matter, it is repeated below. Each procedure call defines a variables *fact*, however, the various variables *fact* defined by different calls are different from each other. In our discussions, we may use the names fact1, fact2, fact3 etc. However, if there is no possibility of confusion then we may use the name *fact* only throughout.

Let us consider how the procedure executes for n = 4 compute the value of factorial (4).

Initially, 1 is assigned to the variable *fact*. Next the procedure checks whether the argument n equals 1. This is not true (*as n is assumed to be 4*). Therefore, the next line with n = 4 is executed i.e.,

fact is to be assigned the value of 4* factorial (3).

Now n, the parameter in the heading of procedure factorial (n) is replaced by 3. Again as n ≠ 1, therefore the next line with n = 3 is executed i.e.,

fact = 3 * factorial (2)

On the similar grounds, we get fact as 2* factorial (1) and at this stage n = 1. The value 1 of fact is returned by the last call of the procedure factorial. And here lies the difficulty in understanding how the desired value 24 is returned. After this stage, the recursive procedure under consideration executes as follows. When factorial procedure is called with n = 1, the value 1 is assigned to fact and this value is returned. However, this value of factorial (1) is passed to the statement fact ←2 * factorial (1) which on execution assigns the value 2 to the variable *fact*. This value is passed to the statement fact ← 3 * factorial (2) which on execution, gives a value of 6 to fact. And finally this value of *fact* is passed to the statement fact← 4 * factorial (3) which in turn gives a value 24 to *fact*. And, finally, this value 24 is returned as value of factorial (4).

Coming back from the definition and procedure for computing factorial (n), let us come to general discussion.

Summarizing, a recursive mathematical definition of a function suggests the definition of a procedure to compute the function. The suggested procedure calls itself recursively with simpler arguments and terminates for some simple argument the required value for which, is directly given within the algorithm or procedure.

Definition: A procedure, which can call itself, is said to be **recursive procedure/algorithm**. For successful implementation of the concept of recursive procedure, the following conditions should be satisfied.

(i) There must be in-built mechanism in the computer system that supports the calling of a procedure by itself, e.g, there may be in-built stack operations on a set of stack registers.

(ii) There must be conditions within the definition of a recursive procedure under which, after finite number of calls, the procedure is terminated.

(iii) The arguments in successive calls should be simpler in the sense that each succeeding argument takes us towards the conditions mentioned in (ii).

In view of the significance of the concept of procedure, and specially of the concept of recursive procedure, in solving some complex problems, we discuss another recursive algorithm for the problem of finding the sum of first n natural numbers, discussed earlier. For the discussion, we assume n is a non-negative integer

```
procedure SUM (n : integer) : integer
        s : integer;
      If n = 0 then return (0)
      else
      begin s← n + SUM (n −1);
            return (s)
      end;
end;
```

Ex.7) Explain how SUM (5) computes sum of first five natural numbers.

1.7 OUTLINE OF ALGORITHMICS

We have already mentioned that *not* every problem has an *algorithmic* solution. The problem, which has at least one algorithmic solution, is called ***algorithmic or computable problem***. Also, we should note that there is *no systematic method* (i.e., algorithm) for designing algorithms even for algorithmic problems. In fact, designing an algorithm for a general algorithmic/computable problem is a difficult intellectual exercise. It requires creativity and insight and no general rules can be formulated in this respect. As a consequence, a discipline called **algorithmics** has emerged that comprises large literature about tools, techniques and discussion of various issues like efficiency etc. related to the design of algorithms. In the rest of the course, we shall be explaining and practicing algorithms. Actually, *algorithmics* could have been an alternative name of the course. We enumerate below some well-known techniques which have been found useful in designing algorithms:

i) Divide and conquer
ii) Dynamic Programming
iii) The Greedy Approach
iv) Backtracking
v) Branch and Bound
vi) Searches and Traversals.

Most of these techniques will be discussed in detail in the text.

In view of the difficulty of solving algorithmically even the computationally solvable problems, some of the problem types, enumerated below, have been more rigorously studied:

(i) Sorting problems

(ii) Searching problems
(iii) Linear programming problems
(iv) Number-theory problems
(v) String processing problems
(vi) Graph problems
(vii) Geometric problems
(viii) Numerical problems.

Study of these specific types of problems may provide useful help and guidance in solving new problems, possibly of other problem types.

Next, we *enumerate* and briefly discuss the *sequence of steps*, which generally, one goes through for designing algorithms for solving (algorithmic) problems, *and analyzing* these algorithms.

1.7.1 Understanding the Problem

Understanding allows appropriate action. This step forms the basis of the other steps to the discussed. For understanding the problem, we should read the statement of the problem, if required, a number of times. We should try to find out

(i) **the type of problem,** so that if a method of solving problems of the type, is already known, then the known method may be applied to solve the problem under consideration.
(ii) **the type of inputs and the type of expected/desired outputs,** specially, the illegal inputs, i.e., inputs which are not acceptable, are characterized at this stage. For example, in a problem of calculating income-tax, the income can not be non-numeric character strings.
(iii) **the range of inputs,** for those inputs which are from ordered sets. For example, in the problem of finding whether a large number is prime or not, we can not give as input a number greater than the Maximum number (Max, mentioned above) that the computer system used for the purpose, can store and arithmetically operate upon. For still larger numbers, some other representation mechanism has to be adopted.
(iv) **special cases of the problem,** which may need different treatment for solving the problem. For example, **if** for an expected quadratic equation $ax^2+bx+c=0$, a, the coefficient of x^2, happens to be zero **then** usual method of solving quadratic equations, discussed earlier in this unit, can not be used for the purpose.

1.7.2 Analyzing the problem

This step is useful in determining the characteristics of the problem under consideration, which may help in solving the problem. Some of the characteristics in this regards are discussed below:

(i) **Whether the problem is decomposable into independent smaller or easier subproblems**, so that programming facilities like procedure and recursion etc. may be used for designing a solution of the problem. For example, the problem of evaluating

$$\int (5x^2 + \sin^2 x \cos^2 x)\,dx$$

can be do decomposed into smaller and simpler problems viz.,

$$5\int x^2\,dx \quad \text{and} \quad \int \text{Sin}^2 x \;\; \text{Cos}^2 x\;dx$$

(ii) **Whether steps in a proposed solution or solution strategy of the problem, may or may not be ignorable, recoverable or inherently irrecoverable,**

i.e., irrecoverable by the (very) nature of the problem. Depending upon the nature of the problem, the solution strategy has to be decided or modified. For example,

a) While proving a theorem, if an unrequired lemma is proved, we may ignore it. The only loss is the loss of efforts in proving the lemma. Such a problem is called *ignorable-step* problem.

b) Suppose we are interested in solving 8-puzzle problem of reaching *from some initial state* say

2	8	7
1	3	5
	6	4

to some final state say

1	2	3
8		4
7	6	5

by sliding, any one of the digits *from* a cell adjacent to the blank cell, *to* the blank cell. Then a wrong step *cannot be* ***ignored but has to be recovered***. By recoverable, we mean that we are allowed to move back to the earlier state from which we came to the current state, if current state seems to be less desirable than the earlier state. The 8-puzzle problem has recoverable steps, or, we may say the problem is a *recoverable* problem

c) However if, we are playing chess, then a wrong step may not be *even* recoverable. In other words, we may not be in a position, because of the adversary's move, to move back to earlier state. Such a problem is called an *irrecoverable step* problem.

Depending on the nature of the problem as ignorable-step, recoverable-step or irrecoverable-step problem, we have to choose our tools, techniques and strategies for solving the problem.

For example, for *ignorable-step problems*, simple control structures for sequencing and iteration may be sufficient. However, if the problem additionally has *recoverable-step* possibilities then facilities like back-tracking, as are available in the programming language PROLOG, may have to be used. Further, if the problem additionally has *irrecoverable-step* possibilities then *planning tools* should be available in the computer system, so that entire sequence of steps may be *analyzed in advance to find out where the sequence may lead to, before the first step is actually taken.*

There are many other characteristics of a problem viz.,

- Whether the problem has certain outcome or uncertain outcome
- Whether a good solution is absolute or relative
- Whether the solution is a state or a path
- Whether the problem essentially requires interaction with the user etc.

which can be known through analyzing the problem under consideration, and the knowledge of which, in turn, may help us in determining or guessing a correct sequence of actions for solving the problem under consideration

1.7.3 Capabilities of the Computer System

We have already discussed the importance of the step in Section 1.5, where we noticed how, because of change in computational capabilities, a totally different

algorithm has to be designed to solve the same problem (*e.g., that of multiplying two natural numbers*).

Most of the computer systems used for educational purposes are PCs based on Von-Neumann architecture. *Algorithms*, that are designed to be executed on such machines are called **sequential algorithms.**

However, new more powerful machines based on *parallel/distributed architectures*, are also increasingly becoming available. Algorithms, that exploit such additional facilities, are called parallel/ distributed; such parallel/distributed algorithms, may not have much resemblance to the corresponding sequential algorithms for the same problem.

However, we restrict ourselves to sequential algorithms only.

1.7.4 Approximate vs Exact Solution

For some problems *like finding the square root* of a given natural number n, it may not be possible to find exact value for all n's (e.g., n = 3). We have to determine in advance what approximation is acceptable, e.g., in this case, the acceptable error may be, say, less than .01.

Also, there are problems, for which finding the exact solutions may be possible, but the cost (*or complexity, to be defined later*) may be too much.

In the case of such problems, unless it is absolutely essential, it is better to use an alternative algorithm which gives reasonably approximate solution, which otherwise may not be exact. For example, consider the **Travelling Salesperson Problem:** A salesperson has a list of, say n cities, each of which he must visit exactly once. There are direct roads between each pair of the given cities. Find ***the shortest possible*** route that takes the salesperson on the round trip starting and finishing in any one of the n cities and visiting other cities exactly once.

In order to find the shortest paths, *one should find the cost of covering each of the n! different paths covering the n given cities*. Even for a problem of visiting 10 cities, n!, the number of possible distinct paths is more than 3 million. In a country like India, a travelling salesperson may be expected to visit even more than 10 cities. To find out exact solution in such cases, though possible, is very time consuming. In such case, a reasonably good approximate solution may be more desirable.

1.7.5 Choice of Appropriate Data Structures

In complex problems particularly, the efficiencies of solutions depend upon choice of appropriate data structures. The importance of the fact has been emphasized long back in 1976 by one of the pioneer computer scientists, *Nickolus Wirth*, in his book entitled "*Algorithms + Data Structures= Programs"*.

In a later paradigm of problem-solving viz., object-oriented programming, choice of appropriate data structure continues to be crucially important for design of efficient programs.

1.7.6 Choice of Appropriate Design Technology

A design technique is a general approach for solving problem that is applicable to computationally solvable problems from various domains of human experience.

We have already enumerated various design techniques and also various problem domains which have been rigorously pursued for computational solutions. For each problem domain, a particular set of techniques have been found more useful, though

other techniques also may be gainfully employed. A major part of the material of the course, deals with the study of various techniques and their suitability for various types of problem domains. Such a study can be a useful guide for solving new problems or new problems types.

1.7.7 Specification Methods for Algorithms

In the introduction of the unit , we mentioned that an algorithm is a description or statement of a sequence of activities that constitute a process of getting desired output from the given inputs. Such description or statement needs to be specified in some notation or language. We briefly mentioned about some possible notations for the purpose. Three well-known notations/languages used mostly for the purpose, are enumerated below:

(i) **Natural language (NL):** An NL is highly expressive in the sense that it can express algorithms of all types of computable problems. However, main problem with an NL is the inherent ambiguity, i.e., a statement or description in NL may have many meanings, all except one, may be unintended and misleading.

(ii) **A Pseudo code** notation is a sort of dialect obtained by mixing some programming language constructs with natural language descriptions of algorithms. The pseudo-code method of notation is the frequently used one for expressing algorithms. However, there is no uniform/standard pseudo-code notation used for the purpose, though, most pseudo-code notations are quite similar to each other.

(iii) **Flowchart** is a method of expressing algorithms by a collection of geometric shapes with imbedded descriptions of algorithmic steps. However, the technique is found to be too cumbersome, specially, to express complex algorithms.

1.7.8 Proving Correctness of an Algorithm

When an algorithm is designed to solve a problem, it is highly desirable that it is **proved** that it satisfies the specification of the problem, i.e., for each possible legal input, it gives the required output. However, the issue of proving correctness of an algorithm, is quite complex. The state of art in this regard is far from its goal of being able to establish efficiently the correctness/incorrectness of an arbitrary algorithm.

The topic is beyond the scope of the course and shall not discussed any more.

1.7.9 Analyzing an Algorithm

Subsection 1.7.2 was concerned with *analyzing the problem* in order to find out special features of the *problem*, which may be useful in designing an algorithm that solves the problem. Here, we assume that one or more algorithms are already designed to solve a problem. The purpose of *analysis of algorithm* is to determine the requirement of computer resources for each algorithm. And then, if there are more than one algorithms that solve a problem, the analysis is also concerned with choosing the better one on the basis of comparison of requirement of resources for different available algorithms. The lesser the requirement of resources, better the algorithm. Generally, the resources that are taken into consideration for analysing algorithms, include

(i) *Time* expected to be taken in executing the instances of the problem generally as a function of the size of the instance.

(ii) *Memory space* expected to be required by the computer system, in executing the instances of the problem, generally as a function of the size of the instances.

(iii) Also sometimes, *the man-hour or man-month* taken by the *team developing the program*, is also taken as a resource for the purpose.

The concerned issues will be discussed from place to place throughout the course material.

1.7.10 Coding the Algorithm

In a course on Design & Analysis of Algorithm, *this step is generally neglected*, assuming that once an algorithm is found satisfactory, writing the corresponding program should be a trivial matter. However, choice of appropriate language and choice of appropriate constructs are also very important. As discussed earlier, if the problem is of recoverable type then, a language like PROLOG which has backtracking mechanism built into the language itself, may be more appropriate than any other language. Also, if the problem is of irrecoverable type, then a language having some sort of PLANNER built into it, may be more appropriate.

Next, even an *efficient algorithm* that solves a problem, may be coded into an *inefficient program*. Even a *correct* algorithm may be encoded into an *incorrect* program.

In view of the facts mentioned earlier that the state of art for proving an algorithm/program correct is still far from satisfactory, we have to rely on testing the proposed solutions. However, testing of a proposed solution can be effectively carried out by executing *the program* on a computer system (*an algorithm, which is not a program can not be executed*). Also by executing different algorithms if more than one algorithm is available, on *reasonably sized* instances of the problem under consideration, we may empirically compare their relative efficiencies. Algorithms, which are not programs, can be hand-executed only for toy-sized instances.

1.8 SUMMARY

1. In this unit the following concepts have been formally or informally defined and discussed:

   Problem, Solution of a Problem, Algorithm, Program, Process *(all section 1.1)* . Instance of a problem *(Section 1.2)*

2. The differences between the related concepts of

   (i) algorithm, program and process *(Section 1.1)*
   (ii) problem and instance of a problem *(Section 1.2)*
   (iii) a general method and an algorithm *(Section 1.4) and*
   (iv) definiteness and effectiveness of an algorithm *(Section 1.4)*

   ,

   are explained

3. The following well-known problems are defined and discussed:

   (i) The Four-Colour Problem *(Section 1.2)*
   (ii) The Fermat's Last Theorem *(Section 1.3)*
   (iii) Travelling Salesperson Problem *(Section 1.7)*
   (iv) 8-puzzle problem *(Section 1.7)*
   (v) *Goldbach conjecture (Solution of Ex.1)*
   (vi) *The Twin Prime Conjecture (Solution of Ex.1)*

4. The following characteristic properties of an algorithm are discussed *(Section 1.4)*

(i) Finiteness
(ii) Definiteness
(iii) Inputs
(iv) Outputs
(v) Effectiveness

5. In order to emphasize the significant role that *available tools* play in the design of an algorithm, the problem of multiplication of two natural numbers is solved in three different ways, each using a different set of available tools. *(Section 1.5)*

6. In Section 1.6, the building blocks of an algorithm including

(a) the instructions viz.,
(i) *assignment* (ii) *read and* (iii) *Write* and
(b) control structures viz.,
(i) *sequencing* (ii) *selection and* (iii) *repetition*
are discussed

7. The important concepts of *procedure* and *recursion* are discussed in Section 1.6.

8. In Section 10, the following issues which play an important role in designing, developing and choosing an algorithm for solving a given problem, are discussed:

(i) understanding the problem
(ii) analysing the problem
(iii) capabilities of the computer system used for solving the problem
(iv) whether required solution must be exact or an approximate solution may be sufficient
(v) choice of appropriate technology
(vi) notations for specification of an algorithm
(vii) proving correctness of an algorithm
(viii) analysing an algorithm and
(ix) coding the algorithm.

1.9 SOLUTIONS/ANSWERS

Ex.1)

Example Problem 1: Find the roots of the Boolean equation

$$ax^2 + bx + c = 0,$$

where $x,y,a,b,c \in \{0,1\}$ and $a \neq 0$

Further, values of a, b and c are given and the value of x is to be determined. Also $x^2 = x.x$ is defined by the equations $0.0 = 0.1 = 1.0 = 0$ and $1.1 = 1$

The problem has only four instances viz

$x^2 + x + 1 = 0$	(for a = 1 = b = c)
$x^2 + x = 0$	(for a = 1 = b, c = 0)
$x^2 + 1 = 0$	(for a = 1 = c, b = 0)
$x^2 = 0$	(for a = 1, b = 0 = c)

Example Problem 2: *(Goldbach Conjecture)*: In 1742, Christian Goldbach conjectured that *every even integer n with n>2, is the sum of two prime numbers.* For example 4=2+2, 6=3+3, 8=5+3 and so on.

The conjecture seems to be very simple to prove or disprove. However, so far the conjecture could neither be established nor refuted, despite the fact that the conjecture has been found to be true for integers more than 4.10^{14}. *Again the Goldbach conjecture is a single instance problem.*

Example Problem 3: *(The Twin Prime Conjecture)*: Two primes are said to be twin primes, if these primes differ by 2. For example, 3 and 5, 5 and 7, 11 and 13 etc. *The conjecture asserts that there are infinitely many twin primes*. Though, twin primes have been found each of which has more than 32,220 digits, but still the conjecture has neither been proved or disproved. *Again the Twin Prime Conjecture is a single instance problem*.

Ex.2)

(i) **A method which is not finite**

Consider the Alternating series

S=1−1/2 +1/3−1/4+1/5……..
S can be written in two different ways, *showing* ½<S<1:
S= 1− (1/2−1/3) – (1/4 −1/5) − (1/6 – 1/7)……
{*Showing S<1 as all the terms within parentheses are positive*}
S= (1−1/2) + (1/3−1/4) + (1/5 −1/6)+……..
{*Showing ½<S, again as all terms within parenthesis are positive and first term equals ½*}

The following method for calculating exact value of S is not finite.

Method Sum-Alternate- Series
begin
$S \leftarrow 1$; $n \leftarrow 2$
While $n \geq 2$ *do*
begin
$S \leftarrow S + (-1)^{(n+1)} . \left(\frac{1}{n}\right)$
$n \leftarrow n+1$
end;
end.

(ii) **A method which is not definite**

Method Not-Definite
Read (x)
{*Let an Urn contain four balls of different colours viz., black, white, blue and red. Before taking the next step, take a ball out of the urn without looking at the ball*}
Begin

If Color-ball= ‘black’ *then*
x← x+1;
else
If color-ball= ‘white’ *then*
x←x+2;
else if colour-ball = ‘blue’ then x←x+3

else
x←x+4;
end.

Then we can see that for *the same value* of x, the method *Not-Definite* may return *different values* in its *different executions*.

(iii) **If any of the following is a part of a method then the method is not effective but finite.**

(a) If the speaker of the sentence:
'I am telling lies'[*]
is actually telling lies,
then x←3
else x← 4

(b) If the statement:
'If the word Heterological is heterological'[**]
is true
then x←3
else x← 4

Ex.3)

The algorithm is obtained by modifying a' la russe method. To begin with, we illustrate our method through an example.

Let two natural numbers to be multiplied be *first* and *second*.

Case 1: When first is divisible by 3, say
first = 21 and second =16
*Then first * second= 21 * 16 = (7* 3)*16*
= 7(3*16)=[first/3]*(3*second)*

---

[*] It is not *possible* to tell whether the speaker is actually telling lies or not. Because , if the speaker is telling lies, then the statement: '*I am telling lies*' should be false. Hence the speaker is not telling lies. Similarly, if the speaker is not telling lies then the statement: *'I am telling lies'* should be true. Therefore, the speaker is telling lies. Hence, it is *not possible* to tell whether the statement is true or not. Thus, the part of the method, and hence the method itself, is not effective. But this part requires only finite amount of time to come to the conclusion that the method is not effective.

[**] A word is said to be **autological** if it is an adjective and the property denoted by it applies to the word itself. For example, each of the words English, polysyllabic are autological. The word single is a single word, hence single is autological. Also, the word autological is autological.

A word is *heterological*, if it is an adjective and the property denoted by the word, does not apply to the word itself. For example, the word *monosyllabic is not monosyllabic.* Similarly, *long* is not a long word. *German* is not a German (language) word. *Double* is not a double word. Thus, each of the words: *monosyllabric , long, German, and double* is heterological. But, if we think of the word *heterological* , which is an adjective, in respect of the matter of determining whether it is heterological or not, then it is not possible to make either of the two statements:

(i) Yes, heterological is heterological
(ii) No, heterological is not heterological.

The reason being that either of these the assertions alongwith definition of heterological leads to the assertion of the other. However, both of (i) and (ii) above can not be asserted simultaneously. Thus it is not possible to tell whether the word heterological is heterological or not.

Case 2: When on division of first by 3, remainder=1

Let first=22 and second=16

Then

*first*second = 22*16= (7*3+1)*16*

*=7*3*16+1*16 = 7*(3*16)+1*16*

=[first/3](3*second)+1*second*

= [first/3] (3* second)+ remainder * second*

Case 3: When on division of first by 3, remainder=2

Let first=23 and second=16. Then

*First * Second = 23*16=(7*3+2)*16*

*=(7*3)* 16+2*16*

=7(3*16)+2*16=[first/3]*(3*second)+2*second*

= [first/3] (3* second)+ remainder * second*

After these preliminary investigations in the nature of the proposed algorithm, let us come to the proper solution of the Ex. 3, i.e.,

Problem: To find the product m * n using the conditions stated in Ex. 3.

The required algorithm which uses variables First, Second, Remainder and Partial-Result, may be described as follows:

***Step 1*:** Initialise the variables First, Second and Partial-Result respectively with m (*the first given number*), n (*the second given number*) and 0.

***Step 2*: If** First or Second* is zero, **then** return Partial-result as the final result and then **stop**. Else

$First_1$=[First/3] ; $Remainder_1 \leftarrow$ First – $First_1$*3;

$Partial\text{-}Result_1 \leftarrow$ $First_1$*$Second_1$;

$Partial\text{-}Result \leftarrow Partial\text{-}Result_1 + Remainder_1$*Second;

Step 3:

{For computing $Partial\text{-}Result_1$, replace First by $First_1$; Second by $Second_1$, and Partial-Result by $Partial\text{-}Result_1$ in Step 2 and repeat Step 2}

First$\leftarrow$ $First_1$; Second=Second

Partial-Result $\leftarrow$ $Partial\text{-}Result_1$

And Go To Step2

---

* If, initially, Second $\neq$ 0, then Second $\neq$ 0 in the subsequent calculations also.
{Remainder is obtained through the equation

** $First = 3*First_1 + Remainder_1$

with $0 \leq Remainder_1 < 2$

$Second_1 = 3*Second$

$Partial\text{-}Result_1 = First_1 * Second_1$

$Partial\text{-}Result = First * Second = (First_1*3+Remainder_1)*(Second)$

$=(First_1*3)*Second+Remainder_1*Second$

$=First_1*(3*Second)+Remainder_1*Second$

$=First_1*Second_1+Remainder_1*Second$

$=Partial\text{-}Result_1+Remainder_1*Second$

Where $0 \leq Remainder_1 < 2$

Thus at every stage, we are multiplying and dividing, if required by at most 3

	First	Second	Remainder on division by 3	Partial Result
Initially:	52	19		0
Step 2	As value of First ≠ 0, hence continue		1	19
Step 3	17	57		
Step 2	Value of first ≠ 0, continue,		2	2*57+19 =133
Step 3	5	171		
Step 2	Value of First ≠ 0, continue		2	2*171+133 = 475
Step 3	1	513		
Step 2	Value of first ≠ 0, continue		1	513+475=988
Step 3	0	304		

As the value of the First is 0, the value 988 of Partial Result is returned as the result and stop.

Ex. 4)

Algorithm Real-Roots-Quadratic

{this algorithm finds the real roots of a quadratic equation $ax^2+bx+c=0$, in which the coefficient a of x^2 is assumed to be non-zero. Variable temp is used to store results of intermediate computations. Further, first and second are variable names which may be used to store intermediate results and finally to store the values of first real root (if it exists) and second real root (if it exists) of the equation}.

```
begin   {of algorithm}
        read (a);
        while (a=0) do {so that finally a ≠ 0}
            read (a);
        read (b,c);
        temp ← b*b – 4*a*c
        If temp<0 then
        begin
                write ('the quadratic equation has no real roots'.)
                STOP
        end;
        else
        if      temp=0          then
        begin
                first= – b/(2*a)
                write ('The two roots of the quadratic equation
                                equal'. The root is' first)
        end;
        {as 'first' is outside the quotes, therefore, value of first will
        be given as output and not the word first will be output}.
        else
        {ie., when temp>0, i.e., when two roots are distinct}
        begin
                temp← sq-root (temp);
                first← (– b+temp)/(2*a);
                second← (– b – temp)/(2*a);
```

```
                write ('The quadratic equation has two distinct roots,
                viz'.,  first, second);
        end;
                {of case when temp>0}
        end; {of algorithm}
```

Ex. 5)

Algorithm Real-Roots-General-Quadratic

{*In this case, a may be zero for the quadratic equation'*
Variable name temp is used to store results of intermediate computations. Further, first and second are variable names which may be used to store intermediate results and finally, first and second store the values of first real root (if it exists) and second real root (if it exists) of the equation}.

```
begin    {of algorithm}
read (a,b,c)
        If  (a=0)          then
        {i.e., when the equation is of the form bx+c=0}
        begin
                if (b=0)          then
                {i.e., when equation is of the form c=0}
                begin
                        if c=0
        {i.e., when equation is of the form 0.x^2+0x+0=0; which is satisfied by
        every real number}
                        write ('every real number is a root of the given equation')
    end;  {of the case a=0, b=0, c=0}
      else {when a=0, b=0 and c≠ 0 then 0x^2+0x+c=0 can not be satisfied by
            any real number}
        begin
                write ('The equation has no roots')
        end {of case a=0, b=0 and c≠ 0}
        end {of case a=0, b=0}
 else {when a=0 but b ≠0 i.e., when
    the equation has only one root viz (−c/b)}
    begin {when a=0, b≠ 0}
        first : = – c/b
    Write ('The equation is linear and its root is' first)
        end {when a=0, b≠ 0}
        else {when a ≠ 0}
    {The complete part of the algorithm of Ex.4 that starts with the statement
        temp: ← b*b – 4 *a*c.
        comes here}
```

Ex.6) (i)

Initially the value of variable n is read as 3. Each of the variables *x* and *Sum* is assigned value 0. Then without any condition the algorithm enters the *do…while* loop. The value x is incremented to 1 and the execution of statement SUM←Sum +1 makes SUM as 1.

Next the condition *x<n* is tested which is true, because x = 1 and n = 3. Once the condition is true the body of the *do..while* loop is entered again and executed second time. By this execution of the loop, x becomes 2 and SUM becomes 1+2=3.

As x = 2< 3 = n, the body of the do..while loop is again entered and executed third time. In this round/iteration, x becomes 3 and SUM becomes 3+3 =6. Again the condition $x<n$ is tested But, now x = 3 n = 3, therefore, $x < n$ is false. Hence the body of the *do..while* loop is no more executed i.e., the loop is terminated. Next, the write statement gives the following output:

The sum of the first 3 natural numbers is 6.

The last statement consisting of *end* followed by dot indicates that the algorithm is to be terminated. Therefore, the algorithm terminates.

Ex 6 (ii)

The algorithm Sum_First_N_3 reads 3 as value of the variable n. Then the algorithm enters the *for-loop*. In the *for-loop,* x is implicitly initialised to 1 and the body of the for-loop is entered. The only statement in the body of the *for-loop*, viz.

SUM← Sum+x

is executed to give the value 1 to SUM. After executing once the body of the for-loop, the value of the *index variable* x is implemental incremented by 1 to become 2.

After each increment in the *index variable,* the value of the *index variable* is compared with the *final value*, which in this case, is n equal to 3. If index variable is less than or equal to n (*in this case*) then body of *for-loop* is executed once again.

As x =≤ 3 = n hence SUM 3 ← SUM+x is executed once more, making SUM equal to 1+2 = 3. Again the index variable x is incremented by 1 to become 3. As 3≤ n (=3) therefore once again the body of the for-loop containing the only statement SUM← SUM+x is executed making 6 as the value of SUM. Next x is automatically incremented by 1 to make x as 4. But as 4 is not less than n (=3). Hence the for-loop is terminated. Next, the write statement gives the output: The sum of the first 3 natural numbers is 6.

The last statement consisting of *end* followed by dot, indicates that the algorithm is to be terminated. Hence, the algorithm is terminated.

Ex.7)

For computing SUM (5) by the algorithm
As n=5 ≠ 0
therefore

$S_5 \leftarrow$ n+ SUM (n-1) = 5+SUM(4)

{It may be noted that in different calls of the procedure SUM, the variable names occurring within the procedure, denote different variables. This is why instead of S we use the names S_i for i=5,4,3,……}

Therefore, in order to compute SUM(5), we need to compute SUM (4)

n=4 ≠ 0, therefore
$S_4 \leftarrow$ 4 + SUM (3)

Continuing like this, we get

$S_3 \leftarrow$ 3 + SUM (2)
$S_2 \leftarrow$ 2 + SUM (1)
$S_1 \leftarrow$ 1 + SUM (0)

At this stage n=0, and accordingly, the algorithm returns value 0. Substituting the value o of SUM (0) we get

S_1= 1+0=1 which is returned by SUM(1).

Substituting this value we get S_2=3. Continuing like this, we get S_3 =6, S_4=10 and S_5=15

1.10 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).

2. *Algoritmics: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).

3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).

4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).

5. *Fundamentals of Computer Algorithms,* E. Horowitz & S. Sahni: (Galgotia Publications).

6. *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).

7. *Programming Languages* (*Second Edition*) – *Concepts and Constructs,* Ravi Sethi: (Pearson Education, Asia, 1996).

UNIT 2 SOME PRE-REQUISITES AND ASYMPTOTIC BOUNDS

Structure **Page Nos.**

2.0 INTRODUCTION

We have already mentioned that there may be more than one algorithms, that solve a given problem. In Section 3.3, we shall discuss eight algorithms to sort a given list of numbers, each algorithm having its own merits and demerits. Analysis of algorithms, the basics of which we study in Unit 3, is an essential tool for making well-informed decision in order to choose the most suitable algorithm, out of the available ones if any, for the problem or application under consideration.

A number of mathematical and statistical tools, techniques and notations form an essential part of the baggage for the analysis of algorithms. We discuss some of these tools and techniques and introduce some notations in Section 2.2. However, for detailed discussion of some of these topics, one should refer to the course material of *MCS-013*.

Also, in this unit, we will study a number of well-known *approximation functions*. These approximation functions which calculate approximate values of quantities under consideration, prove quite useful in many situations, where some of the involved quantities are calculated just for comparison with each other. And the correct result of comparisons of the quantities can be obtained even with approximate values of the involved quantities. In such situations, the advantage is that the approximate values may be calculated much more efficiently than can the actual values.

The understanding of the theory of a routine may be greatly aided by providing, at the time of construction one or two statements concerning the state of the machine at well chose points…In the extreme form of the theoretical method a watertight mathematical proof is provided for the assertions. In the extreme form of the experimental method the routine is tried out one the machine with a variety of initial conditions and is pronounced fit if the assertions hold in each case. Both methods have their weaknesses.

A.M. Turing
Ferranti Mark 1
Programming Manual (1950)

2.1 OBJECTIVES

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

- use a number of mathematical notations e.g., Σ, $\prod$, $\lfloor\ \rfloor$, $\lceil\ \rceil$, mod, log, e etc.

- define and use a number of concepts like function, 1-1 function, onto function, monotonic function, floor and ceiling functions, mod function, exponentiation, logarithm functions etc
- define and use Mathematical Expectation
- Use of Principle of Mathematical Induction in establishing truth of infinitely many statements and

- define and use the five asymptotic function viz.,

(i) O: (O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) Ω: (Ω (n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) Θ: (Θ (n^2) is pronounced as 'theta of n^2')

(iv) o: (o (n^2) is pronounced as 'little-oh of n^2')

(v) ω: (ω (n^2) is pronounced as 'little-omega of n^2').

2.2 SOME USEFUL MATHEMATICAL FUNCTIONS & NOTATIONS

Let us start with some mathematical definitions.

2.2.1 Functions & Notations

Just to put the subject matter in proper context, we recall the following notations and definitions.

Unless mentioned otherwise, we use the letters N, I and R in the following sense:

N = {1, 2, 3, …}
I = {…, − 2, −, 0, 1, 2, ….}
R = set of Real numbers.

Notation 2.2.1.1: If $a_1, a_2 \ldots a_n$ are n real variables/numbers then

(i) Summation:

The expression
$a_1 + a_2 + \ldots + a_i + \ldots + a_n$
may be denoted in shorthand as

$$\sum_{i=1}^{n} a_i$$

(ii) Product

The expression
$a_1 \times a_2 \times \ldots \times a_i \times \ldots \times a_n$
may be denoted in shorthand as

$$\prod_{i=1}^{n} a_i$$

Definition 2.2.1.2:

Function:

For two given sets A and B (*which need not be distinct, i.e., A may be the same as B*) a **rule** f which associates with **each** element of A, a **unique** element of B, is called a function from A to B. If f is a function from a set A to a set B then we denote the fact by **f: A $\rightarrow$ B.** Also, for x $\in$ A, f(x) is called **image** of x in B. Then, A is called the **domain** of f and B is called the **Codomain** of f.

Example 2.2.1.3:

Let f: I $\rightarrow$ I be defined such that

f(x) = x^2 for all x $\in$ I

Then

f maps -4 to 16
f maps 0 to 0
f map 5 to 25

Remark 2.2.1.4:

We may note the following:

(i) if f: x $\rightarrow$ y is a function, then there may be more than one elements, say x_1 and x_2 such that

$f(x_1) = f(x_2)$

For example, in the Example 2.2.1.3

$f(2) = f(-2) = 4$

By putting restriction that $f(x) \neq f(y)$ if $x \neq y$, we get special functions, called 1-1 or injective functions and shall be defined soon.

(ii) Though for each element x $\in$ X, there must be at least one element y $\in$ Y s.t f(x) = y. However, it is not necessary that for each element y $\in$ Y, there must be an element x $\in$ X such that f(x) = y. For example, for $y = -3 \in Y$ there is no x $\in$ X s.t $f(x) = x^2 = -3$.

By putting the restriction on a function f, that for each y $\in$ Y, there must be *at least one* element x of X s.t f(x) = y, we get special functions called onto or surjective functions and shall be defined soon.

Next, we discuss some important functions.

Definition 2.2.1.5:

1-1 **or Injective Function:** A function f: A $\rightarrow$ B is said **to** 1-1* **or injective** if for x, y $\in$ A, if f(x) = f(y) then x = y

We have already seen that the function defined in Example 2.2.1.3 is **not** 1-1. However, by changing the domain, through defined by the same rule, f becomes a 1-1 function.

Example 2.2.1.2:

In this particular case, if we change the domain from I to N = {1,2,3…} then we can easily check that function

* Some authors write 1–to-1 in stead of 1-1. However, other authors call a function 1-to-1 if f is both 1-1 and onto (to be defined 0 in a short while).

$f : N \rightarrow I$ defined as
$f(x) = x^2$, for all $x \in N$,

is 1-1.

Because, in this case, for each $x \in N$ its negative $-x \notin N$. Hence for $f(x) = f(y)$ implies $x = y$. For example, If $f(x) = 4$ then there is **only** one value of x, viz, $x = 2$ s.t $f(2) = 4$.

Definition 2.2.1.7:

Onto/Surjective function: A function $f: X \rightarrow Y$ is said to **onto, or surjective** if to **every element of Y,** the codomain of f, there is an element $x \in X$ s.t $f(x) = y$.

We have already seen that the function defined in Example 2.2.1.3 is **not onto**.

However, in this case either, by changing the codomain Y or changing the rule, (or both) we can make f as Onto.

Example 2.2.1.8: (*Changing the domain*)

Let $X = I = \{\ldots -3, -2, -1, 0, 1, 2, 3, \ldots\}$, but, we change Y as
$Y = \{0, 1, 4, 9, \ldots\} = \{y \mid y = n^2 \text{ for } n \in X\}$

then it can be seen that

$f: X \rightarrow Y$ defined by
$f(x) = x^2$ for all $x \in X$ is Onto

Example 2.2.1.9: (***Changing the rule***)

Here, we change the rule so that $X = Y = \{\ldots -3, -2, -1, 0, 1,2, 3\ldots\}$
But $f: X \rightarrow Y$ is defined as
$F(x) = x + 3$ for $x \in X$.

Then we apply the definition to show that f is onto.

If $y \in Y$, then, by definition, for f to be onto, there must exist an $x \in X$ such that $f(x) = y$. So the problem is to find out $x \in X$ s.t $f(x) = y$ (*the given element of the codomain y*).

Let us *assume* that $x \in X$ exists such that

$f(x) = y$
i.e., $x + 3 = y$
i.e., $x = y - 3$

But, as y is given, x is known through the above equation. Hence f is onto.

Definition 2.2.1.10:

Monotonic Functions: For the definition of monotonic functions, we consider only functions

$f: R \rightarrow R$

where, R is the set of real numbers*.

* Monotonic functions
$f : X \rightarrow Y$,
may be defined even when each of X and Y, in stead of being R, may be any **ordered** sets. But, such general definition is not required for our purpose.

A function f: R $\rightarrow$ R is said to be **monotonically increasing** if for x, y $\in$ R and x $\leq$ y we have f(x) $\leq$ f(y).

In other words, as x increases, the value of its image f(x) also increases for a monotonically increasing function.

Further, f is said to be ***strictly monotonically*** **increasing**, if x < y then f(x) < f(y)

Example 2.2.1.11:

Let f: R $\rightarrow$ R be defined as f(x) = x + 3, for x $\in$ R

Then, for x_1, x_2 $\in$ R, the domain, if $x_1 \geq x_2$ then $x_1 + 3 \geq x_2 + 3$, (*by using monotone property of addition*), which implies $f(x_1) \geq f(x_2)$. Hence, f is monotonically increasing.

We will discuss after a short while, useful functions called ***Floor and Ceiling functions*** *which are monotonic but* ***not strictly*** *monotonic.*

A function f: R $\rightarrow$ R is said to be **monotonically decreasing**, if, for x, y $\in$ R and x $\leq$ y then f(x) $\geq$ f(y).
In other words, as x increases, value of its image decreases.

Further, f is said to be *strictly monotonically decreasing*, if x < y then f(x) > f(y).

Example 2.2.1.12:

Let f: R $\rightarrow$ R be defined as
F(x) = $-$ x + 3
if $x_1 \geq x_2$ then $-x_1 \leq -x_2$ implying $-x_1+3 \leq -x_2 + 3$,
which further implies $f(x_1) \leq f(x_2)$
Hence, f is monotonically decreasing.

Next, we define Floor and Ceiling functions which map every *real* number to an *integer*.

Definition 2.2.1.13:

Floor Function: maps each *real* number x to the *integer*, which is the greatest of all integers less than or equal to x. **Then the image of x is denoted by $\lfloor x \rfloor$.**

Instead of $\lfloor x \rfloor$, the notation [x] is also used.

For example: $\lfloor 2.5 \rfloor = 2, \lfloor -2.5 \rfloor = -3, \lfloor 6 \rfloor = 6$.

Definition 2.2.1.14:

Ceiling Function: *maps each* real *number x to the* integer, *which is the least of all integers greater than or equal to x.* ***Then the image of x is denoted by $\lceil x \rceil$.***

For example: $\lceil 2.5 \rceil = 3, \lceil -2.5 \rceil = -2, \lceil 6 \rceil = 6$

Next, we state a useful result, without proof.

Result 2.1.1.10: For every real number x, **we have**

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1.$$

Example 2.2.1.15:

Each of the floor function and ceiling function is a *monotonically increasing* function but not **strictly** *monotonically increasing function*. Because, for real numbers x and y, if $x \le y$ then $y = x + k$ for some $k \ge 0$.

$\lfloor y \rfloor = \lfloor x + k \rfloor$ = integral part of $(x + k) \ge$ integral part of $x = \lfloor x \rfloor$

Similarly

$\lceil y \rceil = \lceil x + k \rceil$ = least integer greater than or equal to $x + k \ge$ least integer greater than or equal to $x = \lceil x \rceil$.

But, each of floor and ceiling function is **not strictly** increasing, because

$\lfloor 2.5 \rfloor = \lfloor 2.7 \rfloor = \lfloor 2.9 \rfloor = 2$

and

$\lceil 2.5 \rceil = \lceil 2.7 \rceil = \lceil 2.9 \rceil = 3$

2.2.2 Modular Arithmetic/Mod Function

We have been implicitly performing modular arithmetic in the following situations:

(i) If, we are following 12-hour clock, (*which usually we do*) and if it

11 O'clock now then after 3 hours, it will be 2 O'clock and not 14 O'clock (*whenever the number of o'clock exceeds 12, we subtract n = 12 from the number*)

(ii) If, it is 5th day (*i.e., Friday*) of a week, after 4 days, it will be 2nd day

(*i.e., Tuesday*) and not 9th day, *of course of another,* week (*whenever the number of the day exceeds 7, we subtract n = 7 from the number, we are taking here Sunday as 7th day, in stead of 0th day*)

(iii) If, it is 6th month (*i.e., June*) of a year, then after 8 months, it will be 2nd month (*i.e., February*) of, *of course another,* year *(whenever, the number of the month exceeds 12, we subtract n = 12*)

In general, with minor modification, we have

Definition 2.2.2.1:

b mod n: if n is a given *positive* integer and b is *any* integer, then

$b \bmod n = r$ where $0 \le r < n$
and $b = k * n + r$

In other words, r is obtained by subtracting multiples of n from b so that the remainder r lies between 0 and $(n - 1)$.

For example: if $b = 42$ and $n = 11$ then
$b \bmod n = 42 \bmod 11 = 9$.

If $b = -42$ and $n = 11$ then
$b \bmod n = -42 \bmod 11 = 2$ (Θ $-42 = (-4) \times 11 + 2$)

Mod function can also be expressed in terms of the floor function as follows:

$b \pmod n = b - \lfloor b/n \rfloor \times n$

Definition 2.2.2.2:

Factorial: For $N = \{1,2,3,\ldots\}$, the factorial function
factorial: $N \cup \{0\} \rightarrow N \cup \{0\}$
given by

factorial (n) = n × factorial (n – 1)
has already been discussed in detail in Section 1.6.3.2.

Definition 2.2.2.3:

Exponentiation Function Exp: is a function of two variables x and n where x is any non-negative real number and n is an integer **(*though n can be taken as non-integer also, but we restrict to integers only*)**

Exp (x, n) denoted by x^n, is defined recursively as follows:

For n = 0

Exp (x, 0) = $x^0 = 1$

For n > 0

Exp (x, n) = x × Exp (x, n – 1)

i.e

$x^n = x \times x^{n-1}$

For n < 0, let n = – m for m > 0

$$x^n = x^{-m} = \frac{1}{x^m}$$

In x^n, n is also called the exponent/power of x.

For example: if x = 1.5, n = 3, then

Also, Exp (1.5, 3) = $(1.5)^3 = (1.5) \times [(1.5)^2] = (1.5)\,[1.5 \times (1.5)^1]$
= $1.5\,[(1.5 \times (1.5 \times (1.5)^0))]$
= $1.5[(1.5 \times (1.5 \times 1))] = 1.5\,[(1.5 \times 1.5)]$
= $1.5\,[2.25] = 3.375$

$$\text{Exp}(1.5, -3) = (1.5)^{-3} = \frac{1}{(1.5)^3} = \frac{1}{3.375}$$

Further, the following rules apply to exponential function.

For two integers m and n and a real number b the following identities hold:

$$\left((b)^m\right)^n = b^{mn}$$
$$\left(b^m\right)^n = \left(b^n\right)^m$$
$$b^m \,.\, b^n = b^{m+n}$$

For $b \geq 1$ and for all n, the function b^n is monotonically increasing in n. In other words, if $n_1 \geq n_2$ then $b^{n_1} \geq b^{n_2}$ if $b \geq 1$.

Definition 2.2.2.4:

Polynomial: A polynomial in n of degree k, where k is a non-negative integer, over R, the set of real numbers, denoted by P(n), is of the form

$P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$,

where $a_k \neq 0$ and $a_i \in R$, i = 0, 1, …, k.

Using the summation notation

$$P_k(n) = \sum_{i+0}^{k} a_i\, n^i \qquad a_k \neq 0, \quad a_i \in R$$

Each of $(a_i n^i)$ is called a term.

Generally the suffix k in $P_k(n)$ is dropped and in stead of $P_k(n)$ we write P(n) only

We may note that $P(n) = n^k = 1.n^k$ for any k, is a single-term polynomial. If $k \geq 0$ then $P(n) = n^k$ is monotonically increasing. Further, if $k \leq 0$ then $p(n) = n^k$ is monotonically decreasing.

Notation: Though 0^o is not defined, yet, unless otherwise mentioned, we will take $0^o = 1$. The following is a very useful result relating the exponentials and polynomials

Result 2.2.2.5: For any constants b and c with b > 1

$$\lim_{n\to\infty} \frac{n^c}{b^n} = 0$$

The result, in non-mathematical terms, states that for any given constants b and c, but with b > 1, the terms in the sequence $\frac{1^c}{b^1}, \frac{2^c}{b^2}, \frac{3^c}{b^3}, ..., \frac{k^c}{b^k}, ...$ gradually decrease and approaches zero. **Which further means that for constants b and c, and integer variable n, the exponential term b^n, for b > 1, increases at a much faster rate than the polynomial term n^c.**

Definition 2.2.2.6:

The **letter e** is used to denote the quantity

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...,$$

and is taken as the base of natural logarithm function, then for all real numbers x,

we define the exponential function

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

For, all real numbers, we have $e^x \geq 1 + x$

Further, if $|x| \leq 1$ then $1+x \leq e^x \leq 1 + x + x^2$

The following is another useful result, which we state without proof:

Result 2.2.2.7: $\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x$

Definition 2.2.2.8:

Logarithm: The concept of logarithm is defined indirectly through the definition of Exponential defined earlier. If a > 0, b > 0 and c > 0 are three real numbers, such that

$$c = a^b$$

Then $b = \log_a c$ (*read as log of c to the base a*)

Then a is called the base of the algorithm.

For example: if $2^6 = 64$, the $\log_2 64 = 6$,

i.e., 2 raised to power 6 gives 64.

Generally two bases viz 2 and e are very common in scientific and computing fields and hence, the following **specials notations** for these bases are used:

(i) lg n denotes $\log_2 n$ (base 2)
(ii) ln n denotes $\log_e n$ (base e);
where the *letter l* in *ln* denotes *logarithm* and the *letter n* in *ln* denotes *natural.*

The following important properties of logarithms can be derived from the properties of exponents. However, we just state the properties without proof.

Result 2.2.2.9:

For n, a natural number and real numbers a, b and c all greater than 0, the following identities are true:

(i) $\log_a (bc) = \log_a b + \log_a c$

(ii) $\log_a (b^n) = n \log_a b$

(iii) $\log_b a = \log_a b$

(iv) $\log_a (1/b) = -\log_b a$

(v) $\log_a b = \frac{1}{\log_b a}$

(vi) $a^{\log_b c} = c^{\log_b a}$

2.3 MATHEMATICAL EXPECTATION

In average-case analysis of algorithms, to be discussed in Unit 3, we need the concept of **Mathematical expectation.** In order to understand the concept better, let us first consider an example.

Example 2.1: Suppose, the students of MCA, who completed all the courses in the year 2005, had the following distribution of marks.

Range of marks	**Percentage of students who scored in the range**
0% to 20%	08
20% to 40%	20
40% to 60%	57
60% to 80%	09
80% to 100%	06

If a student is picked up randomly from the set of students under consideration, what is the % of marks *expected* of such a student? After scanning the table given above, we *intuitively* expect the student to score around the 40% to 60% class, because, more than half of the students have scored marks in and around this class.

Assuming that marks within a class are uniformly scored by the students in the class, the above table may be approximated by the following more concise table:

% marks	Percentage of students scoring the marks
10*	08
30	20
50	57
70	09
90	06

As explained earlier, we *expect* a student picked up randomly, to score around 50% because more than half of the students have scored marks around 50%.

This *informal* idea of *expectation* may be formalized by giving to each percentage of marks, weight in proportion to the number of students scoring the particular percentage of marks in the above table.

Thus, we assign weight (8/100) to the score 10% (Θ *8, out of 100 students, score on the average 10% marks*); (20/100) to the score 30% and so on.

Thus

Expected % of marks = $10\times\frac{8}{100}+30\times\frac{20}{100}+50\times\frac{57}{100}+70\times\frac{9}{100}+90\times\frac{6}{100}=47$

The final calculation of expected marks of 47 is roughly equal to our intuition of the expected marks, according to our intuition, to be around 50.

We generalize and formalize these ideas in the form of the following definition.

Mathematical Expectation

For a given set S of items, let to each item, one of the n values, say, $v_1, v_2,\ldots,v_n$, be associated. Let the probability of the occurrence of an item with value v_i be p_i. If an item is picked up at random, then its expected value E(v) is given by

$$E(v) = \sum_{i-1}^{n} p_i v_i = p_1.v_1 + p_2.v_2 + ...p_n.v_n$$

2.4 PRINCIPLE OF MATHEMATICAL INDUCTION

Frequently, in establishing the truth of a set of statements, where each of the statement is a function of a natural number n, we will be using the Principle of Mathematical Induction. In view of the significance of the principle, we discuss it here briefly.

The principle is quite useful in establishing the **correctness of a countably many infinite number of statements**, through **only finite number of steps.**

The method consists of the following three major steps:

1. **Verifying/establishing** the correctness of the **Base** Case
2. **Assumption** of Induction Hypothesis for the given set of statements
3. **Verifying/Establishing** the Induction Step

We explain the involved ideas through the following example:

* 10 is the average of the class boundaries 0 and 20.

Let us consider the following sequence in which nth term S(n) is the sum of first (n−1) powers of 2, e.g.,

$S(1) = 2^0 = 2 - 1$
$S(2) = 2^0 + 2^1 = 2^2 - 1$
$S(3) = 2^0 + 2^1 + 2^2 = 2^3 - 1$

We (*intuitively*) feel that

S(n) = $2^0 + 2^1 + \ldots + 2^{n-1}$ should be $2^n - 1$ for all $n \geq 1$.
We may establish the correctness of the intuition, i.e., correctness of all the **infinite** number of statements

$S(n) = 2^n - 1$ for all $n \geq 1$,

through only the following **three** steps:

(i) **Base Case:** (*In this example, n = 1*) we need to show that
$S(1) = 2^1 - 1 = 1$

But, by definition $S(1) = 2^0 = 1 = 2 - 1 = 2^1 - 1$ is correct

(ii) **Induction Hypothesis: Assume**, for some k > base-value (*=1, in this case*) that
$S(k) = 2^k - 1$.

(iii) **Induction Step:** Using (i) & (ii) establish that (*in this case*)

$S(k+1) = 2^{k+1} - 1$

In order to establish
$S(k+1) = 2^{k+1} - 1$, (A)
we use the definition of S(n) and Steps (i) and (ii) above

By definition

$S(k+1) = 2^0 + 2^1 + \ldots + 2^{k+1-1}$
$= (2^0 + 2^1 + \ldots + 2^{k-1}) + 2^k$ (B)

But by definition
$2^0 + 2^1 + \ldots + 2^{k-1} = S(k)$. (C)

Using (C) in (B) we get
$S(k+1) = S(k) + 2^k$ (D)

and by Step (ii) above
$S(k) = 2^k - 1$ (E)

Using (E) in (D), we get

$S(k+1) = (2^k - 1) + 2^k$
$\therefore S(k+1) = 2.2^k - 1 = 2^{k+1} - 1$
which establishes (A).

Ex.1) By using Principle of Mathematical Induction, show that 6 divides $n^3 - n$, where n is a non-negative integer.

Ex.2) Let us assume that we have unlimited supply of postage stamps of Rs. 5 and Rs. 6 then

(i) through, direct calculations, find what amounts can be realized in terms of only these stamps.

(ii) Prove, using Principle of Mathematical Induction, the result of your efforts in part (i) above.

2.5 CONCEPT OF EFFICIENCY OF AN ALGORITHM

If a problem is algorithmically solvable then it may have more than one algorithmic solutions. In order to choose the best out of the available solutions, there are criteria for making such a choice. The complexity/efficiency measures or criteria are based on requirement of computer resources by each of the available solutions. The solution which takes *least resources* is taken as the *best solution* and is generally chosen for solving the problem. However, it is very difficult to even enumerate all possible computer resources e.g., time taken by the *designer* of the solution and the time taken by the *programmer* to encode the algorithm.

Mainly the two computer resources taken into consideration for efficiency measures, are *time and space* requirements for *executing* the program corresponding to the solution/algorithm. *Until it is mentioned otherwise, we will restrict to only time complexities of algorithms of the problems.*

In order to understand the complexity/efficiency of an algorithm, it is very important to understand the notion of the **size of an instance** of the problem under consideration and the role of size in determining complexity of the solution.

It is easy to realize that given an algorithm for multiplying two $n \times n$ matrices, the time required by the algorithm for finding the product of two 2×2 matrices, is expected to take much less time than the time taken by the same algorithm for multiplying say two 100×100 matrices. This explains intuitively *the notion of the size of an instance of a problem* and also the role of size in determining the (*time*) complexity of an algorithm. **If the *size* (to be later considered formally) of general *instance is n* then time complexity of the algorithm solving the *problem* (not just the instance) under consideration is *some function of n.***

In view of the above explanation, the notion of *size* of an instance of a problem plays an important role in determining the complexity of an algorithm for solving the problem under consideration. However, it is difficult to *define precisely* the concept of size in general, for all problems that may be attempted for algorithmic solutions.

*Formally, one of the definitions of the **size of an instance** of a problem may be taken as the **number of bits** required in representing the instance.*

However, for all types of problems, this does not serve properly the purpose for which the notion of size is taken into consideration. Hence different measures of size of an instance of a problem, are used for different types of problem. For example,

(i) In sorting and searching problems, the *number of elements*, which are to be sorted or are considered for searching, is taken *as the size of the instance* of the problem of sorting/searching.

(ii) In the case of solving polynomial equations or while dealing with the algebra of polynomials, the *degrees* of polynomial instances, may be taken as the sizes of the corresponding instances.

There are *two approaches* for determining complexity (or time required) for executing an algorithm, viz.,

(i) empirical (*or a posteriori*) and
(ii) theoretical (*or a priori*).

In the **empirical approach** (the programmed) algorithm is *actually* executed on various *instances* of the *problem* and the *size* (s) and *time* (*t*) of execution for each instance is noted. And then by some numerical or other technique, *t* is determined as a function of s. This function then, is taken as complexity of the *algorithm* under consideration.

In the theoretical approach, we *mathematically* determine the time needed by the algorithm, for a *general instance* of size, say, n of the problem under consideration. In this approach, generally, each of the basic instructions like *assignment, read* and *write* and each of the basic operations like '+', comparison of pair of integers etc., is assumed to take one or more, but some constant number of, (basic) units of time for execution. Time for execution for each *structuring rule*, is assumed to be some function of the times required for the constituents of the structure.
Thus starting from the basic instructions and operations and using structuring rules, one can calculate the time complexity of a program or an algorithm.

The theoretical approach has a number of advantages over the empirical approach including the ones enumerated below:

(i) The approach does not depend on the programming language in which the algorithm is coded and on how it is coded in the language,

(ii) The approach does not depend on the computer system used for executing (a programmed version of) the algorithm.

(iii) In case of a comparatively inefficient algorithm, which ultimately is to be rejected, the computer resources and programming efforts which otherwise would have been required and wasted, will be saved.

(iv) In stead of applying the algorithm to many different-sized instances, the approach can be applied for a *general size say n* of an arbitrary instance of the problem under consideration. In the case of theoretical approach, the size n may *be arbitrarily large*. However, in empirical approach, because of practical considerations, only the instances of *moderate sizes* may be considered.

Remark 2.5.1:

In view of the advantages of the theoretical approach, we are going to use it as *the only approach* for computing complexities of algorithms. As mentioned earlier, in the approach, no particular computer is taken into consideration for calculating time complexity. But different computers have different execution speeds. However, the speed of one computer is generally some constant multiple of the speed of the other.

Therefore, this fact of differences in the speeds of computers by constant multiples is taken care of, in the complexity functions *t* for general instance sizes *n*, by writing the complexity function as *c.t(n)* where c is an arbitrary constant.

*An **important consequence of the above discussion** is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem.* Thus, functions differing from each other by constant factors, when treated as time complexities should not be treated as different, i.e., should be treated as complexity-wise equivalent.

Remark 2.5.2:

Asymptotic Considerations:

Computers are generally used to solve problems involving *complex* solutions. The complexity of solutions may be either because of the large number of involved computational steps and/or because of large size of input data. The plausibility of the claim apparently follows from the fact that, when required, computers are used *generally not to* find the product of two 2×2 matrices *but to find* the product of two $n \times n$ matrices for large n, running into hundreds or even thousands.

Similarly, computers, when required, are generally used *not only to find roots* of quadratic equations but for finding roots of complex equations including polynomial equations of *degrees more than hundreds or sometimes even thousands.*

The above discussion leads to the conclusion that when considering time complexities $f_1(n)$ and $f_2(n)$ of (computer) solutions of a problem of size n, we need to consider and compare the behaviours of the two functions only for large values of n. If the relative behaviours of two functions for smaller values conflict with the relative behaviours for larger values, then we may ignore the conflicting behaviour for smaller values. For example, if the earlier considered two functions

$$f_1(n) = 1000\, n^2 \qquad \text{and}$$
$$f_2(n) = 5n^4$$

represent time complexities of two solutions of a problem of size n, then despite the fact that

$$f_1(n) \geq f_2(n) \qquad \text{for } n \leq 14,$$

we would still prefer the solution having $f_1(n)$ as time complexity because

$$f_1(n) \leq f_2(n) \qquad \text{for all } n \geq 15.$$

This explains the reason for the presence of the phrase 'n ≥ k' in the definitions of the various measures of complexities and approximation functions, discussed below:

Remark 2.5.3:

Comparative Efficiencies of Algorithms: Linear, Quadratic, Polynomial Exponential

Suppose, for a given problem P, we have two algorithms say A_1 and A_2 which solve the given problem P. Further, assume that we also know time-complexities $T_1(n)$ and $T_2(n)$ of the two algorithms for problem size n. How do we know which of the two algorithms A_1 and A_2 is better?

The difficulty in answering the question arises from the difficulty in comparing time complexities $T_1(n)$ and $T_2(n)$.

For example, let $\mathbf{T_1(n) = 1000n^2}$ and $\mathbf{T_2(n) = 5n^4}$
Then, for all n, neither $T_1(n) \leq T_2(n)$ nor $T_2(n) \leq T_1(n)$.

More explicitly

$$T_1(n) \geq T_2(n) \text{ for } n \leq 14 \qquad \text{and}$$
$$T_1(n) \leq T_2(n) \text{ for } n \geq 15.$$

The issue will be discussed in more detail in Unit 3. However, here we may mention that, in view of the fact that we generally use computers to solve problems of *large*

sizes, **in the above case, the algorithms A_1 with time-complexity $T_1(n) = 1000n^2$ is preferred over the algorithm A_2 with time-complexity $T_2(n) = 5n^4$, because $T_1(n) \leq T_2(n)$ for all $n \geq 15$.**

In general if a problem is solved by two algorithms say B_1 and B_2 with time-complexities $BT_1(N)$ and $BT_2(n)$ respectively, then

(i) **if** $BT_1(n)$ is a *polynomial* in n, i.e,

$$BT_1(n) = a_k n^k + a_{k-1} + \ldots + a_i n^i + \ldots + a_1 n + a_0$$

for some $k \geq 0$ with a_i's as real numbers and $a_k > 0$, and

$BT_2(n)$ is an *exponential* function of n, i.e., $BT_2(n)$ is of the form $BT_2(n) = c\, a^n$ where c and a are some real numbers with $a > 1$,

then generally, for large values of n, $BT_1(n) \leq BT_2(n)$.

Hence, algorithm B_1 with *polynomial time* complexity is assumed to the more efficient and is preferred over algorithm B_2 with *exponential time* complexity.

(ii) If, again a problem is solved by two algorithms D_1 and D_2 with respectively polynomial time complexities DT_1 and DT_2 then if

degree (DT_1) < degree (DT_2),

then the algorithm D_1 is assumed to be more efficient and is preferred over D_2.

Certain complexity functions occur so frequently that special names commensurate with their usage may be given to such functions. For example, complexity function $c\,n$ is called *linear time complexity* and corresponding algorithm, is called as *linear algorithm*.

Similarly, the terms '***quadratic***' *and* '***polynomial time***' *complexity functions and algorithms* are used when the involved complexity functions are respectively of the forms $c\,n^2$ and $c_1 n^k + \ldots\ldots + c_k$.

In the next section we find examples of linear and quadratic algorithms.

Remark 2.5.4:

For all practical purposes, *the use of c, in* ($c\,t(n)$) *as time complexity measure*, offsets properly the effect of differences in the speeds of computers. *However, we need to be on the guard, because in some rarely occurring situations, neglecting the effect of c may be misleading*.

For example, if two algorithms A_1 and A_2 respectively take n^2 *days* and n^3 *secs* for execution of an instance of size n of a particular problem. *But a 'day' is a constant multiple of a 'second'*. Therefore, as per our conventions we may take the two complexities as of $C_2 n^2$ *and* $C_3 n^3$ for some constants C_2 and C_3. As, we will discuss later, the algorithm A_1 taking $C_2 n^2$ time is *theoretically* preferred over the algorithm A_2 with time complexity $C_3 n^3$. The preference is based on asymptotic behaviour of complexity functions of the algorithms. *However* in this case, *only* for instances requiring millions of years, the algorithm A_1 requiring $C_2 n^2$ time outperforms algorithms A_2 requiring $C_3 n^3$.

Remark 2.2.5:

Unit of Size for Space Complexity: Though most of the literature discusses the complexity of an algorithm only in terms of expected time of execution, generally

neglecting the space complexity. However, space complexity has one big advantage over time complexity.

*In the case of space complexity, unit of measurement of space requirement can be well defined as a **bit.** But in the case of time complexity such obvious choice is not available. The problem lies in the fact that unlike 'bit' for space, there is no standard time unit such as 'second' since, we do not have any standard computer to which single time measuring unit is applicable.*

Ex.3) For a given problem P, two algorithms A_1 and A_2 have respectively time complexities T_1 (n) and T_2 (n) in terms of size n, where

$T_1(n) = 4n^5 + 3n$ and
$T_2(n) = 2500n^3 + 4n$

Find the range for n, the size of an instance of the given problem, for which A_1 is more efficient than A_2.

2.6 WELL KNOWN ASYMPTOTIC FUNCTIONS & NOTATIONS

We often want to know a quantity only approximately, and not necessarily exactly, just to compare with another quantity. And, in many situations, correct comparison may be possible even with approximate values of the quantities. The advantage of the possibility of correct comparisons through even approximate values of quantities, is that the time required to find approximate values may be much less than the times required to find exact values.

We will introduce five approximation functions and their notations.

The purpose of these asymptotic growth rate functions to be introduced, is to facilitate the recognition of *essential character of a complexity function* through some simpler functions delivered by these notations. For example, a complexity function $f(n) = 5004\ n^3 + 83\ n^2 + 19\ n + 408$, has essentially the same behaviour as that of $g(n) = n^3$ as the problem size n becomes larger and larger. But $g(n) = n^3$ is much more comprehensible and its value easier to compute than the function f(n).

2.6.1 Enumerate the five well-known approximation functions and how these are pronounced

(i) O: (O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) Ω: (Ω (n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) Θ: (Θ (n^2) is pronounced as 'theta of n^2')

(iv) o: (o (n^2) is pronounced as 'little-oh of n^2')

(v) ω: (ω (n^2) is pronounced as 'little-omega of n^2').

The O-notation was introduced by Paul Bachman in his book Analytische Zahlentheorie (1894)

These approximations denote relations from functions to functions.

For example, if functions

f, g: N→N are given by

$f(n) = n^2 - 5n$ and
$g(n) = n^2$

then

$O(f(n)) = g(n)$ or $O(n^2 - 5n) = n^2$

To be more precise, each of these notations is a mapping that associates a *set of* functions to each function under consideration. For example, if f (n) is a polynomial of degree k then the set O (f (n)) includes all polynomials of degree less than or equal to k.

Remark 2.6.1.1:

In the discussion of any one of the five notations, generally two functions say f and g are involved. The functions have their domains and codomains as N, the set of natural numbers, i.e.,

f: N→N
g: N→N

These functions may also be considered as having domain and codomain as R.

2.6.2 The Notation O

Provides asymptotic *upper bound* for a given function. Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number constants C and k such that

$f(x) \leq C\, g(x)$ for all $x \geq k$ (A)

(*The restriction of being positive on integers/reals is justified as all complexities are positive numbers*).

Example 2.6.2.1: For the function defined by

$f(x) = 2x^3 + 3x^2 + 1$
show that

(i) $f(x) = O(x^3)$
(ii) $f(x) = O(x^4)$
(iii) $x^3 = O(f(x))$
(iv) $x^4 \neq O(f(x))$
(v) $f(x) \neq O(x^2)$

Solution:

Part (i)

Consider

$f(x) = 2x^3 + 3x^2 + 1$
$\leq 2x^3 + 3x^3 + 1 \quad x^3 = 6x^3$ for all $x \geq 1$

(by replacing each term x^i by the highest degree term x^3)

$\therefore$ there exist $C = 6$ and $k = 1$ such that
$f(x) \leq C.\ x^3$ for all $x \geq k$

Thus we have found the required constants C and k. Hence f(x) is $O(x^3)$.

Part (ii)

As above, we can show that

$f(x) \leq 6\ \ x^4$ for all $x \geq 1$.

However, we may also, by computing some values of f(x) and x^4, find C and k as follows:

$f(1) = 2+3+1 = 6$; $(1)^4 = 1$
$f(2) = 2.2^3 + 3.2^2 + 1 = 29$; $(2)^4 = 16$
$f(3) = 2.3^3 + 3.3^2 + 1 = 82$; $(3)^4 = 81$

for $C = 2$ and $k = 3$ we have
$f(x) \leq 2.\ x^4$ for all $x \geq k$

Hence f(x) is $O(x^4)$.

Part (iii)

for $C = 1$ and $k = 1$ we get
$x^3 \leq C\ (2x^3 + 3x^2 + 1)$ for all $x \geq k$

Part (iv)

We prove the result by contradiction. Let there exist positive constants C and k such that

$x^4 \leq C\ (2x^3 + 3x^2 + 1)$ for all $x \geq k$
$\therefore x^4 \leq C\ (2x^3 + 3x^3 + x^3) = 6Cx^3$ for $x \geq k$
$\therefore\ x^4 \leq 6\ C\ x^3$ for all $x \geq k$.

implying $x \leq 6C$ for all $x \geq k$
But for $x = \max$ of $\{ 6\ C + 1, k\}$, the previous statement is not true.
Hence the proof.

Part (v)

Again we establish the result by contradiction.

Let $O\ (2\ x^3+3x^2+1) = x^2$

Then for some positive numbers C and k

$2x^3 + 3x^2+1 \leq C\ x^2$ for all $x \geq k$,

implying

$x^3 \leq C\ x^2$ for all $x \geq k$ *(Θ $x^3 \leq 2x^3+3x^2+1$ for all $x \geq 1$)*

implying

$x \leq C$ for $x \geq k$

Again for x = max $\{C+1, k\}$

The last inequality does not hold. Hence the result.

Example 2.6.2.2:

The big-oh notation can be used to estimate S_n, the sum of first n positive integers

Hint: S_n=1+2+3+…….+n ≤ n+n +……+ n = n^2

Therefore, $S_n = O(n^2)$.

Remark 2.6.2.2:

It can be easily seen that for given functions f(x) and g(x), if there exists one pair of C and k with f(x) ≤ C.g (x) for all x ≥ k, then there exist infinitely many pairs (C_i, k_i) which satisfy

f(x) ≤ C_i g(x) for all x ≥ k_i.

Because for **any C_i** ≥ C and **any k_i** ≥ k, the above inequality is true, if f(x)≤ c.g(x) for all x ≥ k.

2.6.3 The Ω Notation

The Ω Notation provides an asymptotic *lower bound* for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

f(x) ≥ C (g(x)) whenever x ≥ k

Example 2.6.3.1:

For the functions

f(x) = $2x^3 + 3x^2 + 1$ and h (x) = $2x^3 - 3x^2 + 2$

show that

(i) f(x) = Ω (x^3)

(ii) h(x)= Ω (x^3)

(iii) h(x)= Ω (x^2)

(iv) x^3 = Ω (h(x))

(v) $x^2 \neq$ Ω (h(x))

Solutions:

Part (i)

For C =1, we have
f(x) ≥ C x^3 for all x ≥ 1

Part (ii)

$h(x) = 2x^3 - 3x^2 + 2$
Let C and k > 0 be such that
$2x^3 - 3x^2 + 2 \geq C\, x^3$ for all $x \geq k$
i.e., $(2 - C)\, x^3 - 3x^2 + 2 \geq 0$ for all $x \geq k$

Then C = 1 and $k \geq 3$ satisfy the last inequality.

Part (iii)

$$2x^3 - 3x^2 + 2 = \Omega(x^2)$$

Let the above equation be true.

Then there exists positive numbers C and k
s.t.

$$2x^3 - 3x^2 + 2 \geq C\, x^2 \quad \text{for all } x \geq k$$
$$2\, x^3 - (3 + C)\, x^2 + 2 \geq 0$$

It can be easily seen that lesser the value of C, better the chances of the above inequality being true. So, to begin with, let us take C = 1 and try to find a value of k s.t

$$2x^3 - 4x^2 + 2 \geq 0.$$

For $x \geq 2$, the above inequality holds

$\therefore$ k=2 is such that

$2x^3 - 4x^2 + 2 \geq 0$ for all $x \geq k$

Part (iv)

Let the equality

$$x^3 = \Omega(2x^3 - 3x^2 + 2)$$

be true. Therefore, let C>0 and k > 0 be such that

$$x^3 \geq C(2(x^3 - 3/2\, x^2 + 1))$$

For C = ½ and k = 1, the above inequality is true.

Part (v)

We prove the result by contradiction.

Let $x^2 = \Omega(3x^3 - 2x^2 + 2)$

Then, there exist positive constants C and k such that

$$x^2 \geq C\,(3x^3 - 2x^2 + 2) \quad \text{for all } x \geq k$$

$$\text{i.e., } (2C + 1)\, x^2 \geq 3C\, x^3 + 2 \geq C\, x^3 \text{ for all } x \geq k$$

$$\frac{2C + 1}{C} \geq x \quad \text{for all } x \geq k$$

But for any $x \geq 2 \frac{(2C+1)}{C}$,

The above inequality can not hold. Hence contradiction.

2.6.4 The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

(*Note the last inequalities represent two conditions to be satisfied simultaneously viz., $C_2\, g(x) \leq f(x)$ and $f(x) \leq C_1\, g(x)$*)

We state the following theorem without proof, which relates the three functions O, Ω, Θ.

Theorem: For any two functions f(x) and g(x), $f(x) = \Theta(g(x))$ if and only if
$f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

Examples 2.6.4.1: For the function
$f(x) = 2x^3 + 3x^2 + 1$,
show that

(i) $f(x) = \Theta(x^3)$

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

(iii) $f(x) \neq \Theta(x^4)$

Solutions

Part (i)

for $C_1 = 3$, $C_2 = 1$ and $k = 4$

1. $C_2 x^3 \leq f(x) \leq C_1 x^3$ for all $x \geq k$

Part (ii)

We can show by contradiction that no C_1 exists.

Let, if possible for some positive integers k and C_1, we have $2x^3+3x^2+1 \leq C_1 . x^2$ for all $x \geq k$

Then

$x^3 \leq C_1 x^2$ for all $x \geq k$

i.e.,

$x \leq C_1$ for all $x \geq k$

But for

$x = \max \{C_1 + 1, k\}$

The last inequality is not true.

Part (iii)

$f(x) \neq \Theta(x^4)$

We can show by contradiction that there does not exist C_2 s.t

$$C_2 x^4 \leq (2x^3 + 3x^2 + 1)$$

If such a C_2 exists for some k then $C_2 x^4 \leq 2x^3 + 3x^2 + 1 \leq 6x^3$ for all $x \geq k \geq 1$,

implying

$$C_2 x \leq 6 \text{ for all } x \geq k$$

But for $x = \left(\frac{6}{C_2} + 1\right)$

the above inequality is false. Hence, proof of the claim by contradiction.

2.6.5 The Notation o

The asymptotic upper bound provided by big-oh notation may or may not be tight in the sense that if $f(x) = 2x^3 + 3x^2 + 1$

Then for $f(x) = O(x^3)$, though there exist C and k such that

$$f(x) \leq C(x^3) \text{ for all } x \geq k$$

yet there may also be some values for which the following equality also holds

$$f(x) = C(x^3) \qquad \text{for } x \geq k$$

However, if we consider

$$f(x) = O(x^4)$$

then there can not exist positive integer C s.t

$$f(x) = C x^4 \quad \text{for all } x \geq k$$

The case of $f(x) = O(x^4)$, provides an example for the next notation of small-oh.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let $C > 0$ **be any number**, then $f(x) = o(g(x))$ (pronounced as little oh of g of x) if there exists natural number k satisfying

$$f(x) < C\, g(x) \text{ for all } x \geq k \geq 1 \qquad \text{(B)}$$

Here we may note the following points

(i) In the case of little-oh the constant C does not depend on the two functions f (x) and g (x). Rather, we can *arbitrarily* choose C >0

(ii) The inequality (B) is strict whereas the inequality (A) of big-oh is not necessarily strict.

Example 2.6.5.1: For $f(x) = 2x^3 + 3x^2 + 1$, we have

(i) $f(x) = o(x^n)$ for any $n \geq 4$.

(ii) $f(x) \neq o(x^n)$ for $n \leq 3$

Solutions:

Part (i)

Let C > 0 **be given and** to find out k satisfying the requirement of little-oh.
Consider

$$2x^3 + 3x^2 + 1 < C\ x^n$$
$$= 2 + \frac{3}{x} + \frac{1}{x^3} < C\ x^{n-3}$$

Case when n = 4

Then above inequality becomes

$$2 + \frac{3}{x} + \frac{1}{x^3} < C\ x$$

if we take $k = \max\left\{\frac{7}{C}, 1\right\}$

then

$2x^3 + 3x^2 + 1 < C\ x^4$ for $x \geq k$.

In general, as $x^n > x^4$ for $n \geq 4$,

therefore

$2x^3 + 3x^2 + 1 < C\ x^n$ for $n \geq 4$
for all $x \geq k$
with $k = \max\left\{\frac{7}{c}, 1\right\}$

Part (ii)

We prove the result by contradiction. Let, if possible, $f(x) = 0(x^n)$ for $n \leq 3$.

Then there exist positive constants C and k such that $2x^3+3x^2+1 < C\ x^n$
for all $x \geq k$.

Dividing by x^3 throughout, we get

$$2 + \frac{3}{x} + \frac{1}{x^2} < C\ x^{n-3}$$

$n \leq 3$ and $x \geq k$

As C is arbitrary, we take

C = 1, then the above inequality reduces to

$$2+ \frac{3}{x} + \frac{1}{x^2} < C.\ x^{n-3} \quad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

Also, it can be easily seen that

$$x^{n-3} \leq 1 \qquad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

$$\therefore\ 2+ \frac{3}{x} + \frac{1}{x^2} \leq 1 \qquad \text{for } n \leq 3$$

However, the last inequality is not true. Therefore, the proof by contradiction.

Generalising the above example, we get the

Example 2.6.5.3: If f(x) is a polynomial of degree m and g(x) is a polynomial of degree n. Then

f(x) = o(g(x)) if and only if n>m.

We state (without proof) below two results which can be useful in finding small-oh upper bound for a given function.

More generally, we have

Theorem 2.6.5.3: Let f(x) and g(x) be functions in definition of small-oh notation.

Then f(x) = o(g(x) if and only if

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

Next, we introduce the last asymptotic notation, namely, small-omega. The relation of small-omega to big-omega is similar to what is the relation of small-oh to big-oh.

2.6.6 The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then

f(x) = ω (g(x))

if there exist a positive integer k s.t

f(x) > C g(x) for all x ≥ k

Example 2.6.6.1:

If $f(x) = 2x^3 + 3x^2 + 1$

then

$f(x) = \omega(x)$

and also

$f(x) = \omega(x^2)$

Solution:

Let C be any positive constant.

Consider

$2x^3 + 3x^2 + 1 > C\,x$

To find out $k \geq 1$ satisfying the conditions of the bound ω.

$2x^2 + 3x + \frac{1}{x} > C$ *(dividing throughout by x)*

Let k be integer with $k \geq C+1$

Then for all $x \geq k$

$2x^2 + 3x + \frac{1}{x} \geq 2x^2 + 3x > 2k^2 + 3k > 2C^2 + 3C > C.$ $(\Theta\ k \geq C+1)$

$\therefore f(x) = \omega(x)$

Again, consider, for any $C > 0$,

$2x^3 + 3x^2 + 1 > C\,x^2$

then

$2x + 3 + \frac{1}{x^2} > C$ Let k be integer with $k \geq C+1$

Then for $x \geq k$ we have

$2x + 3 + \frac{1}{x^2} \geq 2x + 3 > 2k + 3 > 2C + 3 > C$

Hence

$f(x) = \omega(x^2)$

In general, we have the following two theorems (stated without proof).

Theorem 2.6.6.2: If f(x) is a polynomial of degree n, and g(x) is a polynomial of degree n, then

$f(x) = \omega(g(x))$ if and only if $m > n$.

More generally

Theorem 2.6.6.3: Let f(x) and g(x) be functions in the definitions of little-omega Then $f(x) = \omega(g(x))$ if and only if

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

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

Ex.4) Show that $n! = O(n^n)$.

Ex.5) Show that $n^2 + 3\log n = O(n^2)$.

Ex.6) Show that $2^n = O(5^n)$.

2.7 SUMMARY

In this unit, first of all, a number of mathematical concepts are defined. We defined the concepts of function, 1-1 function, onto function, ceiling and floor functions, mod function, exponentiation function and log function. Also, we introduced some mathematical notations.

In Section 2.3, the concept of mathematical expectation is introduced, which is useful in average-case analysis. Formally, mathematical expectation is defined as follows:

Mathematical Expectation

For a given set S of items, let to each item, one of the n values, say, $v_1, v_2,\ldots,v_n$, be associated. Let the probability of the occurrence of an item with value v_i be p_i. If an item is picked up at random, then its expected value E(v) is given by

$$E(v) = \sum_{i-1}^{n} p_i v_i = p_1.v_1 + p_2.v_2 + \ldots\ldots\ldots p_n.v_n$$

Also, the Principle of Mathematical Induction is discussed, which is useful in establishing truth of infinitely many statements.
The method of Mathematical Induction consists of the following three major steps:

4. **Verifying/establishing** the correctness of the **Base** Case
5. **Assumption** of Induction Hypothesis for the given set of statements
6. **Verifying/Establishing** the Induction Step

Complexity of an algorithm: There are two approaches to discussing and computing complexity of an algorithm viz

(i) empirical

(ii) theoretical.

These approaches are briefly discussed in Section 2.5. However, in the rest of the course *only theoretical* approach is used for the analysis and computations of complexity of algorithms. Also, it is mentioned that analysis and computation of complexity of an algorithm may be considered in terms of

(i) *time* expected to be taken or

(ii) *space* expected to be required for executing the algorithm.

Again, we will throughout consider *only the time complexity*. Next, the concepts of linear, quadratic, polynomial and exponential time complexities and algorithms, are discussed.

Next, five **Well Known Asymptotic Growth Rate functions are defined and corresponding notations are introduced. Some important results involving these are stated and/or proved**

The notation O provides asymptotic *upper bound* for a given function.
Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.
Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \leq C\, g(x)$ for all $x \geq k$

The Ω notation provides an asymptolic *lower bound* for a given function

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \geq C\,(g(x))$ whenever $x \geq k$

The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let C > 0 **be any number**, then f(x) = o(g(x)) (pronounced as little oh of g of x) if there exists natural number k satisfying

$f(x) < C\, g(x)$ for all $x \geq k \geq 1$

The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is as follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then

$f(x) = \omega\,(g(x))$

if there exist a positive integer k s.t

$f(x) > C\ \ g(x)$ for all $x \geq k$

2.8 SOLUTIONS/ANSWERS

Ex. 1) We follow the three-step method explained earlier.
Let S(n) be the statement: 6 divides $n^3 - n$

Base Case: For n = 0, we **establish** S(0); i.e., we establish that 6 divides $0^3 - 0 = 0$

But $0 = 6 \times 0$. Therefore, 6 divides 0. Hence S(0) is correct.

Induction Hypothesis: For any positive integer k **assume** that S(k) is correct, i.e., assume that 6 divides $(k^3 - k)$.

Induction Step: Using the conclusions/assumptions of earlier steps, to **show** the correctness of S(k+1), i.e., to *show* that 6 divides $(k+1)^3 - (k+1)$.

Consider $(k+1)^3 - (k+1) = (k^3 + 3k^2 + 3k + 1) - (k + 1)$
$= k^3 + 3k^2 + 2k = k(k+1)(k+2)$

If we show that k (k+1) (k+2) is divisible by 6 than by Principle of Mathematical Induction the result follows.

Next, we prove that 6 divides k (k+1) (k+2) for all non-negative integers.

As k, (k+1) and (k+2) are three consecutive integers, therefore, at least one of these is even, i.e., divisible by 2. Hence it remains to show that k(k+1) (k+2) is divisible by 3. This we establish through the following **case analysis:**

(i) If k is divisible by 3, then, of course, k(k+1) (k+2) is also divisible by 3.

(ii) If on division of k, remainder is 1, i.e., if $k = 3t + 1$ for some integer t then $k(k+1)(k+2) = (3t+1)(3t+2)(3t+3) = 3(3t+1)(3t+2)(t+1)$ is divisible by 3.

(iii) If on division of k, remainder is 2, i.e., if $k = 3t + 2$ then $k(k+1)(k+2) = (3t+2)(3t+3)(3t+4) = 3(3t+2)(t+1)(3t+4)$ is again divisible by 3.

Ex. 2) **Part (i):** With stamps of Rs. 5 and Rs. 6, we can make the following the following amounts

5	=	$1\times5 + 0\times6$	
6	=	$0\times5 + 1\times6$	using 2 stamps
10	=	$2\times5 + 0\times6$	
11	=	$1\times5 + 1\times6$	using 2 stamps
12	=	$0\times5 + 2\times6$	
15	=	$3\times5 + 0\times6$	
16	=	$2\times5 + 1\times6$	
17	=	$1\times5 + 2\times6$	using 3 stamps
18	=	$0\times5 + 3\times6$	
19 is not possible			
20	=	$4\times5 + 0\times6$	
21	=	$3\times5 + 1\times6$	
22	=	$2\times5 + 2\times6$	using 4 stamps
23	=	$1\times5 + 3\times6$	
24	=	$0\times5 + 4\times6$	

25	=	$5\times5 + 0\times6$	using 5 stamps
26	=	$4\times5 + 1\times6$	
27	=	$3\times5 + 2\times6$	
28	=	$2\times5 + 3\times6$	
29	=	$1\times5 + 4\times6$	
30	=	$0\times5 + 5\times6$	

It appears that for any amount A ≥ 20, it can be realized through stamps of only Rs. 5 and Rs. 6.

Part (ii): We attempt to show using Principle of Mathematical Induction, that any amount of Rs. 20 or more can be realized through using a number of stamps of Rs. 5 and Rs. 6.

Base Step: For **amount = 20,** we have shown earlier that $20 = 4.5 + 0.6$.

Hence, Rs. 20 can be realized in terms of stamps of Rs. 5 and Rs. 6.

Induction Hypothesis: Let for some **k ≥ 20**, amount k can be realized in terms of some stamps of Rs. 5 and some stamps of Rs. 6.

Induction Step: Next, the case of the amount k+1 is **analysed** as follows:

Case (i): If at least one Rs. 5 stamp is used in making an amount of Rs. k, then we replace one Rs. 5 stamp by one Rs. 6 stamp to get an amount of Rs. (k+1). Hence the result in this case.

Case (ii): If all the 'Stamps' in realizing Rs. k ≥ 20 is through only Rs. 6 stamps, then k must be at least 24 and hence at least 4 stamps must be used in realzing k. Replace 4 of Rs. 6 stamps by 5 of Rs. 5 stamps. So that out of amount k, $6 \times 4 = 24$ are reduced through removing of 4 stamps of Rs. 6 and an amount of Rs. $5 \times 5 = 25$ is added. Thus we get an amount of $k - 24 + 25 = k+1$. This completes the proof.

Ex. 3) Algorithm A_1 is more efficient than A_2 for those values of n for which

$$4n^5 + 3n = T_1(n) \leq T_2(n) = 2500\, n^3 + 4n$$

i.e.,

$$4n^4 + 3 \leq 2500\, n^2 + 4$$

i.e.,

$$4n^2 - 2500 \leq 1/n^2 \qquad \text{(i)}$$

Consider

$$4n^2 - 2500 = 0$$

then $n = 50/2 = 25$

for n ≤ 25

$$4n^2 - 2500 \leq (25)^2 - 2500 \leq 0 \leq 1/n^2$$

Hence (i) is satisfy

Next, consider n ≥ 26

$$4n^2 - 2500 \geq 4(26)^2 - 2500 = 2704 - 2500$$
$$= 204 > 1 > \frac{1}{(26)^2} \geq \frac{1}{n^2}$$

Therefore, **for n ≥ 26, (i)is not satisfied**

Conclusion: For problem sizes n, with $1 \leq n \leq 25$, A_1 is more efficient than A_2. However, for $n \geq 25$, A_2 is more efficient than A_1

Ex. 4)

$$n!/n^n = (n/n)\ ((n-1)/n)\ ((n-2)/n)\ ((n-3)/n)\ldots(2/n)(1/n)$$
$$= 1(1-(1/n))\ (1-(2/n))\ (1-(3/n))\ldots(2/n)(1/n)$$

Each factor on the right hand side is less than equal to 1 for all value of n. Hence, The right hand side expression is always less than one.

Therefore, $n!/n^n \leq 1$

or, $n! \leq n^n$

Therefore, $n! = O(n^n)$

Ex. 5)

For large value of n, $3\log n << n^2$

Therefore, $3\log n/ n^2 << 1$

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

or, $(n^2 + 3\log n)/ n^2 < 2$

or, $n^2 + 3\log n = O(n^2)$.

Ex.6) We have, $2^n/5^n < 1$

or, $2^n < 5^n$

Therefore, $2^n = O(5^n)$.

2.9 FURTHER READINGS

1. *Discrete Mathematics and Its Applications* (*Fifth Edition*) K.N. Rosen: Tata McGraw-Hill (2003).

2. *Introduction to Alogrithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice – Hall of India (2002).

UNIT 3 BASICS OF ANALYSIS

Structure **Page Nos.**

3.0 INTRODUCTION

Analysis of algorithms is an essential tool for making well-informed decision in order to choose the most suitable algorithm, out of the available ones, if any, for the problem or application under consideration. For such a choice of an algorithm, which is based on some efficiency measures relating to computing resources required by the algorithm, there is no systematic method. To a large extent, it is a matter of judgment and experience. However, there are ***some basic techniques and principles*** that help and guide us in analysing algorithms. These techniques are mainly for

(i) analysing control structures and

(ii) for solving recurrence relations, which arise if the algorithm involves recursive structures.

In this unit, we mainly discuss models, techniques and principles for analyzing algorithms.

Also, sorting algorithms, which form good sources for learning how to design and analyze algorithms, are discussed in detail in this unit.

It appears that the whole of the conditions which enable a **finite** machine to make calculations of **unlimited** extent are fulfilled in the Analytical Engine…I have converted the infinity of space, which was required by the conditions of the problem, into the infinity of time.

Charles Babbage
About
Provision of iteration & conditional branching in the design of his analytical engine (year 1834), the first general purpose digital computer

3.1 OBJECTIVES

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

- explain and use various types of analyses of algorithms;
- tell how we compute complexity of an algorithm from the complexities of the basic instructions using the structuring rules;
- solve recurrence equations that arise in recursive algorithms, and
- explain and use any one of the several well-known algorithms discussed in the text, for sorting a given array of numbers.

3.2 ANALYSIS OF ALGORITHMS – SIMPLE EXAMPLES

In order to discuss some simple examples of analysis of algorithms, we write two algorithms for solving the problem of computing prefix averages (*to be defined*). And then find the complexities of the two algorithms. In this process, we also find that how minor change in an algorithm may lead to substantial gain in the efficiency of an algorithm. To begin with we define the *problem of computing prefix average*.

Computing Prefix Averages: For a given array A[1..n] of numbers, the problem is concerned with finding an array B[1..n] such that

B[1] = A[1]
B[2] = average of first two entries = (A[1] + A [2])/2
B[3] = average of first 3 entries = (A[1] + A[2] + A[3])/3

and in general for $1 \leq i \leq n$,

B[i] = average of first i entries in the array A[1..n]
= (A[1] + A[2] + …+A[i])/i

Next we discuss two algorithms that solve the problem, in which second algorithm is obtained by minor modifications in the first algorithm, but with major gains in algorithmic complexity – the first being a *quadratic* algorithm, whereas the second algorithm is *linear*. Each of the algorithms takes array A[1..n] of numbers as input and returns the array B[1..n] as discussed above.

Algorithm First-Prefix-Average (A[1..n])

```
begin {of algorithm}
  for i ← 1 to n do
     begin {first for-loop}
       Sum ← 0;
        {Sum stores the sum of first i terms, obtained in different
          iterations of for-loop}

     for j ← 1 to i do
      begin {of second for-loop}
        Sum ← Sum + A[j];
      end {of second for- loop}
          B[i] ← Sum/i
      end {of the first for-loop}
end {of algorithm}
```

Analysis of First-Prefix-Averages:

***Step 1*:** Intitalization step for setting up of the array A[1..n] takes constant time say C_1, in view of the fact that for the purpose, *only address of A (or of A[1]) is to be passed*. Also after all the values of B[1..n] are computed, then returning the array B[1..n] also takes constant time say C_2, again for the same reason.

***Step 2*:** The body of the algorithm has two nested for-loops, the outer one, called the *first for-loop* is controlled by i and is executed n times. Hence the *second for-loop alongwith its body*, which form a part of the *first for-loop,* is executed n times. Further each construct within *second for-loop*, controlled by j, is executed i times just because of the iteration of the *second for-loop*. However, the *second for-loop* itself is being executed n times because of the first for-loop. Hence each instruction within the *second for-loop* is executed (n.i) times for each value of i = 1, 2, …n.

***Step 3*:** In addition, each controlling variable i and j is incremented by 1 after each iteration of i or j as the case may be. Also, after each increment in the control variable, it is compared with the (*upper limit + 1*) of the loop to stop the further execution of the for-loop.

Thus, the *first for-loop* makes n additions (*to reach* (*n+1*)) and n comparisons with (n+1).

The second for-loop makes, for each value of i=1,2,…,n, one addition and one comparison. Thus total number of each of additions and comparisons done just for controlling variable j

$= (1+2+\ldots+n) = \frac{n(n+1)}{2}$.

***Step 4*:** Using the explanation of Step 2, we count below the number of times the various operations are executed.

(i) Sum $\leftarrow$ 0 is executed n times, once for each value of i from 1 to n

(ii) On the similar lines of how we counted the number of additions and comparisons for the control variable j, it can be seen that the number of each of additions ($Sum \leftarrow Sum + A[j]$) and divisions ($B[i] \leftarrow Sum/i$) is $\frac{n(n+1)}{2}$.

Summerizing, the total number of operations performed in executing the First-Prefix-Averages are

(i) (*From Step* 1) Constant C_1 for initialization of A[1..n] and Constant C_2 for returning B[1..n]

(ii) (*From Step 3*)

Number of additions for control variable i = n.
Number of Comparisons with (n+1) of variable i = n
Number of additions for control variable j = $\frac{n(n+1)}{2}$
Number of comparisons with (i+1) of control variable j = $\frac{n(n+1)}{2}$
Number of initializations ($Sum \leftarrow 0$) = n.
Number of additions ($Sum \leftarrow Sum + A[j]$) = $\frac{n(n+1)}{2}$
Number of divisions (*in Sum/i*) = $\frac{n(n+1)}{2}$

Number of assignments in ($Sum \leftarrow Sum + A[j]$) = $\frac{n(n+1)}{2}$

Number of assignments in ($B[i] \leftarrow Sum/i$) = $\frac{n(n+1)}{2}$

Assuming each of the operations counted above takes some constant number of unit operations, then total number of all operations is a **quadratic function of n**, the size of the array A[1..n].

Next, we show that a minor modification in the First-Prefix-Algorithm, may lead to an algorithm, to be called Second-Prefix-Algorithm and defined below of *linear complexity only.*

The main change in the algorithm is that the partial sums $\sum_{k+1}^{i} A[k]$ are not computed afresh, repeatedly (*as was done in First-Prefix-Averages*) but $\sum_{k+1}^{i} A[k]$ is computed only once in the variable *Sum* in Second_Prefix_Average. The new algorithm is given below:

Algorithm Second-Prefix-Averages (A[1..n]);

```
begin {of algorithm}
        Sum ← 0
  for i ← 1 to n do
    begin {of for loop}
          Sum ← Sum + A[i];
          B[i] ← Sum/i.
    end; {of for loop}
  return (B[1..n]);
end {of algorithm}
```

Analysis of Second-Prefix-Averages

***Step1*:** As in First-Prefix-Averages, the intialization step of setting up the values of A[1..n] and of returning the values of B[1..n] each takes a constant time say C_1 and C_2 (*because in each case only the address of the first element of the array viz A[1] or B[1] is to be passed*)

The assignment through Sum ← 0 is executed once

The *for loop* is executed exactly n times, and hence

***Step 2*:** There are *n additions* for incrementing the values of the loop variable and *n comparisons* with (n+1) in order to check whether *for loop* is to be terminated or not.

***Step 3*:** There *n additions*, one for each i (*viz Sum + A* [*i*]) and n *assignments*, again one for each i (*Sum* ← *Sum* + *A*[*i*]). Also there are n divisions, one for each i (*viz Sum/i*) and n (more) assignments, one for each i (*viz B*[*i*] ← *Sum/i*).

Thus we see that overall there are

(i) 2 n additions
(ii) n comparisons
(iii) (2n+1) assignments
(iv) n divisions
(v) C_1 and C_2, constants for initialization and return.

As each of the operations, viz addition, comparison, assignment and division takes a constant number of units of time; therefore, the total time taken is C.n for some constant C.

Thus Second-Prefix-Averages is a *linear algorithm* (or has linear time complexity)

3.3 WELL KNOWN SORTING ALGORITHMS

In this section, we discuss the following well-known algorithms for sorting a given list of numbers:

1. Insertion Sort
2. Bubble Sort
3. Selection Sort
4. Shell Sort
5 Heap Sort
6. Merge Sort
7. Quick Sort

For the discussion on Sorting Algorithms, let us recall the concept of *Ordered Set*.

We know given two integers, say n_1 and n_2, we can always say whether $n_1 \leq n_2$ or $n_2 \leq n_1$. Similarly, if we are given two rational numbers or real numbers, say n_1 and n_2, then it is always possible to tell whether $n_1 \leq n_2$ or $n_2 \leq n_1$.

Ordered Set: Any set S with a relation, say, ≤, is said to be ordered if for any two elements x and y of S, either $x \leq y$ or $y \leq x$ is true. Then, we may also say that (S, ≤) is an ordered set.

Thus, if I, Q and R respectively denote set of integers, set of rational numbers and set of real numbers, and if '≤' *denotes* '*the less than or equal to*' relation then, each of (I, ≤), (Q, ≤) and (R, ≤) is an ordered set. However, it can be seen that the set $C = \{x + iy : x, y \in R \text{ and } i^2 = -1\}$ of complex numbers *is not ordered* w.r.t '≤'. For example, it is not possible to tell for at least one pair of complex numbers, say 3 + 4i and 4+3i, whether 3 + 4i ≤ 4 +3i, or 4 + 3i ≤ 3 + 4i.

Just to facilitate understanding, we use the list to be sorted as that of numbers. However, the following discussion about sorting is equally valid for a list of elements from an **arbitrary ordered set**. In that case, we use the word *key* in stead of *number* in our discussion.

The general treatment of each sorting algorithm is as follows:

1. First, the method is briefly described in English.
2. Next, an example explaining the method is taken up.
3. Then, the algorithm is expressed in a pseudo- programming language.
4. The analysis of these algorithm is taken up later at appropriate places.

All the sorting algorithms discussed in this section, are for sorting numbers in *increasing order*.

Next, we discuss sorting algorithms, which form a rich source for algorithms. Later, we will have occasions to discuss general polynomial time algorithms, which of course include linear and quadratic algorithms.

One of the important applications for studying Sorting Algorithms is the area of designing efficient algorithms for searching an item in a given list. If a set or a list is ***already sorted***, then we can have more efficient searching algorithms, which include

binary search and B-Tree based search algorithms, each taking ($c.\ log\ (n)$) time, where n is the number of elements in the list/set to be searched.

3.3.1 Insertion Sort

The insertion sort, algorithm for sorting a list L of n numbers represented by an array A[1..n] proceeds by picking up the numbers in the array from left one by one and each newly picked up number is placed at its relative position, w.r.t the sorting order, among the earlier ordered ones. The process is repeated till the each element of the list is placed at its correct relative position, i.e., when the list is sorted.

Example 3.3.1.1

A[1]	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**A[6]**	
80	**32**	**31**	**110**	**50**	**40**	{← *given list, initially*}

{*Pick up the left-most number 80 from the list, we get the sorted sublist*}

80

{*Pick up the next number 32 from the list and place it at correct position relative to 80, so that the sublist considered so far is sorted*}.

32 80

{*We may note in respect of the above sorted sublist, that in order to insert 32 before 80, we have to shift 80 from first position to second and then insert 32 in first position.*

The task can be accomplished as follows:

1. *First 32 is copied in a location say m*
2. *80 is copied in the location A[2] = 32 so that we have*

A[1]	*A[2]*	*m*
80	*80*	*32*

3. *32 is copied in A[1] from m so that we have*

A[1]	*A[2]*	*m*
32	*80*	*32*

thus we get the sorted sublist given above}

32 80

{*Next number 31 is picked up, compared first with 80 and then* (*if required*) *with 32. in order to insert 31 before 32 and 80, we have to shift 80 to third position and then 32 to second position and then 31 is placed in the first position*}.

The task can be accomplished as follows:

1. *First 31 is copied in a location say m*
2. *80 is copied in the location A[3] = 31 so that we have*

A[1]	*A[2]*	*A[3]*	*m*
32	*80*	*80*	*31*

3. *32 is copied in A[2] from A[1] so that we have*

A[1]	*A[2]*	*A[3]*	*m*
32	*32*	*80*	*31*

4. *the value 31 from m is copied in A[1] so that we get*

A[1]	*A[2]*	*A[3]*	*m*
31	*32*	*80*	*31*

thus we get the sorted sublist}

31 32 80

{Next 110 is picked up, compared with 80. As 110>80, therefore, no shifting and no more comparisons. 110 is placed in the first position after 80}.

31 32 80 110

{Next, number 50 is picked up. First compared with 110, found less; next compared with 80, again found less; again compared with 32. The correct position for 50 is between 32 and 80 in the sublist given above. Thus, each of 110 and 80 is shifted one place to the right to make space for 50 and then 50 is placed over there

The task can be accomplished as follows:

1. *First 50 is copied in a location say m*
2. *110 is copied in the location A[5] = 50 so that we have*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 80 110 110 50
3. *80 is copied in A[4] from A[3] so that we have*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 80 80 110 50
4. *the value 50 from m is copied in A[3] so that we get*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 50 80 110 50

thus we get the following sorted sublist}

31 32 50 80 110

{Next in order to place 40 after 32 and before 50, each of the values 50, 80 and 110 need to be shifted one place to the right as explained above. However, values 31 and 32 are not to be shifted. The process of inserting 40 at correct place is similar to the ones explained earlier}.

31 32 40 50 80 110

Algorithm: The Insertion Sort

The idea of Insertion Sort as explained above, may be implemented through procedure *Insertion-Sort* given below. It is assumed that the numbers to be sorted are stored in an array A[1..n].

```
Procedure Insertion-Sort (A[1..n ] : real)
 begin {of procedure}
  if n = 1
  then
   write ('A list of one element is already sorted')
   else
    begin {of the case when n ≥ 2}
     for  j ← 2 to n do

        {to find out the correct relative position for A[ j] and insert it there  among
         the already sorted elements A[1] to A [j − 1]}

   begin {of for loop}
     If A[j] < A[j − 1]  then begin
       {We shift entries only if A [j] < A [j − 1]}
       i ← j − 1;   m ← A[j]
```

```
{In order to find correct relative position we store A[j] in m and start with the
last element A[j−1] of the already sorted part. If m is less than A[j−1], then
we move towards left, compare again with the new element of the array. The
process is repeated until either m ≥ some element of the array or we reach the
left-most element A[1]}.

while (i > 0 and m < A[i]) do
begin {of while loop}
  A[i+1] ← A[i]
    i ← i − 1
end

{After finding the correct relative position, we move all the elements of the
array found to be greater than m = A[j], one place to the right so as to make
a vacancy at correct relative position for A[j]}
end; {of while loop}

    A[i +1] ← m
      {i.e., m = A[j] is stored at the correct relative position}
  end {if}
  end; {of for loop}
 end; {of else part}
end; {of procedure}
```

Ex. 1) Sort the following sequence of number, using Insertion Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.2 Bubble Sort

The Bubble Sort algorithm for sorting of n numbers, represented by an array A[1..n], proceeds by scanning the array from left to right. At each stage, compares *adjacent pairs* of numbers at positions A[i] and A[i+1] and whenever a pair of adjacent numbers is found to be out of order, then the positions of the numbers are swapped. The algorithm repeats the process for numbers at positions A[i+1] and A[i+2].

Thus in the first pass after scanning once all the numbers in the given list, the largest number will reach its destination, but other numbers in the array, may not be in order. In each subsequent pass, one more number reaches its destination.

3.3.2.1 Example

In the following, in each line, pairs of adjacent numbers, **shown in bold**, are compared. And if the pair of numbers are not found in proper order, then the positions of these numbers are exchanged.

The list to be sorted has n = 6 as shown in the first row below:

iteration number i = 1

80	**32**	31	110	50	40	(j = 1)
32	**80**	**31**	110	50	40	(j = 2)
32	31	**80**	**110**	50	40	
32	31	80	**110**	**50**	40	
32	31	80	50	**110**	**40**	(j = 5)
32	**31**	81	50	40	*110*	(j = 1)
					↑	
					remove from further consideration	

In the first pass traced above, the maximum number 110 of the list reaches the rightmost position (i.e 6 th position). In the next pass, only the list of remaining $(n - 1) = 5$ elements, as shown in the first row below, is taken into consideration. Again pairs of numbers in bold, in a row, are compared and exchanged, if required.

iteration number i = 2

31	**32**	**81**	50	40	(j = 2
31	32	**81**	**50**	40	(j = 3)
31	32	50	**81**	**40**	(j = 4)
31	**32**	50	40	*81*	(j = 1)

↑

remove from further consideration

In the second pass, the next to maximum element of the list viz, 81, reaches the 5th position from left. In the next pass, the list of remaining $(n - 2) = 4$ elements are taken into consideration.

iteration number i = 3

31	**32**	**50**	40	(j = 2)
31	32	**50**	**40**	(j = 3)
31	**32**	40	*50*	(j = 1)

↑

remove from further consideration

In the next iteration, only (n – 3) = 3 elements are taken into consideration.

31	**32**	**40**
31	**32**	40

In the next iteration, only n – 4 = 2 elements are considered

31 32

These elements are compared and found in proper order. The process terminates.

Procedure ***bubblesort*** **(A[1..n])**
begin
for **i ← 1** to **n – 1 do**
for j ← 1 to **(n – i).**
{*in each new iteration, as explained above, one less number of elements is taken into consideration. This is why j varies upto only* $(n - i)$}

if **A[j] > A[j+1]** then **interchange A[j]and A[j+1].**
end
{A[1..n] is in increasing order}

Note: As there is only one statement in the scope of each of the two for-loops, therefore, no 'begin' and 'end' pair is used.

Ex. 2) Sort the following sequence of numbers using Bubble Sort:
15, 10, 13, 9, 12, 17.
Further, find the number of comparisons and assignments required by the algorithm in sorting the list.

3.3.3 Selection Sort

Selection Sort for sorting a list L of n numbers, represented by an array A[1..n], proceeds by finding the maximum element of the array and placing it in the last position of the array representing the list. Then repeat the process on the subarray representing the sublist obtained from the list by excluding the current maximum element.

*The difference between Bubble Sort and Selection Sort, is that in **Selection Sort** to find the maximum number* in the array, a new variable MAX is used to keep maximum of all the values scanned upto a particular stage. On the other hand, in **Bubble Sort**, the maximum number in the array under consideration is found by comparing adjacent pairs of numbers and by keeping larger of the two in the position at the right. Thus after scanning the whole array once, the maximum number reaches the right-most position of the array under consideration.

The following steps constitute the Selection Sort algorithm:

Step 1: Create a variable MAX to store the maximum of the values scanned upto a particular stage. Also create another variable say MAX-POS which keeps track of the position of such maximum values.

Step 2: In each iteration, the whole list/array under consideration is scanned once to find out the current maximum value through the variable MAX and to find out the position of the current maximum through MAX-POS.

Step 3: At the end of an iteration, the value in last position in the current array and the (maximum) value in the position MAX-POS are exchanged.

Step 4: For further consideration, replace the list L by L ~ {MAX} {*and the array A by the corresponding subarray*} and go to Step 1.

Example 3.3.3.1:

80 32 31 **110** 50 **40** {← *given initially*}

Initially, MAX ← 80 MAX-POS ←1
After one iteration, finally; MAX ← 110, MAX-POS = 4

Numbers 110 and 40 are exchanged to get

80 32 31 **40** 50 **110**

New List, after one iteration, to be sorted is given by:

80 32 31 40 50

Initially MAX ← 80, MAX-POS ←1
Finally also MAX ← 80, MAX-POS ←1
∴ entries 80 and 50 are exchanged to get

50 32 31 40 80

New List, after second iteration, to be sorted is given by:

50 32 31 40

Initially Max ← 50, MAX-POS ←1
Finally, also Max ← 50, MAX-POS ←1

∴ entries 50 and 40 are exchanged to get

40 32 31 50

New List , after third iteration, to be sorted is given by:

40 32 31

Initially & finally

Max ← 40 ; MAX-POS ← 1

Therefore, entries 40 and 31 are exchanged to get

31 32 40

New List, after fourth iteration, to be sorted is given by:

31 32

Max ← 32 and MAX-POS ← 2

∴ No exchange, as 32 is the last entry

New List, after fifth iteration, to be sorted is given by:

31

This is a single-element list. Hence, no more iterations. The algorithm terminates This completes the sorting of the given list.

Next, we formalize the method used in sorting the list.

```
Procedure Selection Sort (A[1..n])
begin {of procedure}
   for i ← 1 to (n − 1) do
        begin {of i for loop}
           MAX ← A[i];
           MAX-POS ← i
            for j ← i+1 to n do
             begin {j for-loop}*
             If MAX < A[j] then
                  begin
                      MAX-POS ←j
                      MAX ← A[j]
                   end {of if }
             end {of j for-loop}
           A [MAX-POS] ← A[n−i+1]
           A[n−i+1] ← MAX
           {the ith maximum value is stored in the ith position from right or
             equivalently in (n − i + 1) th position of the array}
           end {of i loop}
      end {of procedure}.
```

Ex. 3) Sort the following sequence of number, using Selection Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

* as there is only one statement in j-loop, we can Omit 'begin' and 'end'.

3.3.4 Shell Sort

The sorting algorithm is named so in honour of D.L. Short (1959), who suggested the algorithm. Shell Sort is also called *diminishing –increment* sort. The essential idea behind Shell-Sort is to apply any of the other sorting algorithm (generally Insertion Sort) to each of the several interleaving sublists of the given list of numbers to be sorted. In successive iterations, the sublists are formed by stepping through the file with an increment INC_i taken from some pre-defined decreasing sequence of step-sizes $INC_1 > INC_2 > \ldots > INC_i \ldots > 1$, which must terminate in 1.

Example 3.3.4.2: Let the list of numbers to be sorted, be represented by the next row.

13 3 4 12 14 **10** 5 1 8 2 **7** 9 11 6 (*n = 14*)

Initially, we take INC as 5, i.e.,

Taking sublist of elements at 1st , 6th and 11th positions, viz sublist of values 13, 10 and 7. After sorting these values we get the sorted sublist

7 10 13

Taking sublist of elements at 2nd, 7th and 12th positions, viz sublist of values 3, 5 and 9. After sorting these values we get the sorted sublist.

3 5 9

Taking sublist of elements at 3rd, 8th and 13th positions, viz sublist of values 4, 1 and 11. After sorting these values we get the sorted sublist.

1 4 11

Similarly, we get sorted sublist

6 8 12

Similarly, we get sorted sublist

2 14

{*Note that, in this case, the sublist has only two elements* Θ *it is 5th sublist and n = 14 is less than* $\left(\left\lfloor \frac{14}{\text{INC}} \right\rfloor \bullet \text{INC} + 5\right)$ where INC = 5}

After merging or interleaving the entries from the sublists, while maintaining the initial relative positions, we get the **New List:**

7 3 1 **6** 2 10 **5** 4 8 **14** 13 9 **11** 12

Next, take INC = 3 and repeat the process, we get sorted sublists:

5 6 7 11 14,
2 3 4 12 13 and
1 8 9 10

After merging the entries from the sublists, while maintaining the initial relative positions, we get the **New List:**

New List

5 2 1 6 3 8 7 4 9 11 12 10 14 13

Taking INC = 1 and repeat the process we get the sorted list

1 2 3 4 5 6 7 8 9 10 11 12 13 14

Note: *Sublists should not be chosen at distances which are mutliples of each other e.g. 8, 4, 2 etc. otherwise, same elements may be technique compared again and again.*

```
Procedure Shell-Sort (A[1..n]: real)
        K: integer;
{to store value of the number of increments}
        INC [1..k]: integer
{to store the values of various increments}
       begin {of procedure}
        read (k)
        for i ← 1 to (k – 1) do
         read (INC [i])
          INC [k] ← 1
           {last increment must be 1}
            for i ← 1 to k do
             {this i has no relation with previous occurrence of i}
                 begin
                   j ← INC [i]
                   r ← [n/j];
                    for t ← 1 to k do
                     {for selection of sublist starting at position t}
                     begin
                       if n < r * j + t
                       then  s ← r – 1

                        else s ← r
                      Insertion-Sort (A [t… (t + s * j)])
                    end {of t-for-loop}
                end {i for-loop}
               end {of procedure}.
```

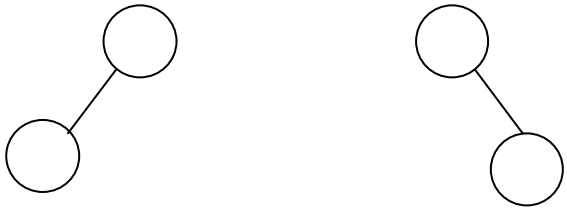
Ex. 4) Sort the following sequence of number, using Shell Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.
Also, initial INC = 3 and final INC = 1 for selecting interleaved sublists. For sorting sublists use Insertion Sort.

3.3.5 Heap Sort

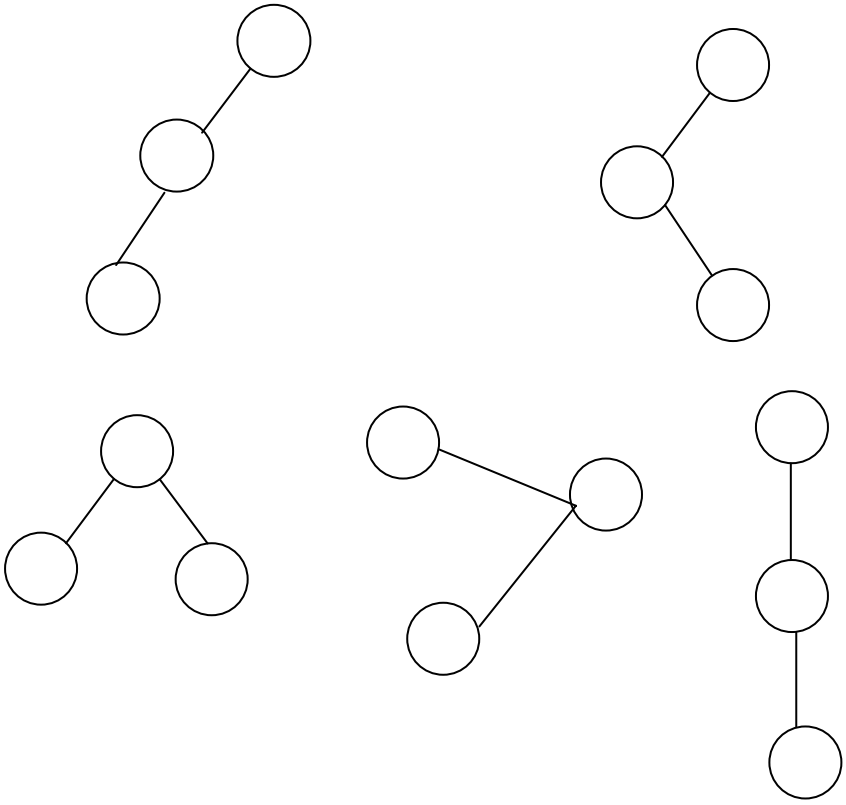
In order to discuss Heap-Sort algorithm, we recall the following definitions; where we assume that the concept of a tree is already known:

Binary Tree: A tree is called a binary tree, if it is either empty, or it consists of a node called the root together with two *binary trees* called the **left subtree** and a **right subtree**. In respect of the above definition, we make the following observations:

1. It may be noted that the above definition is a *recursive* definition, in the sense that definition of binary tree is given in its own terms (*i.e., binary tree*). In Unit 1, we discussed other examples of recursive definitions.
2. The following are all distinct and the only binary trees having two nodes.



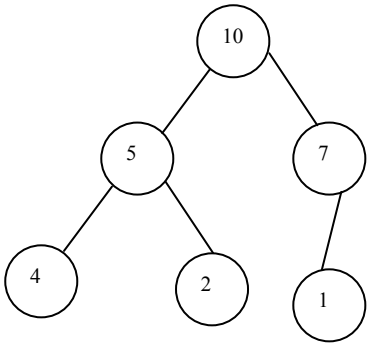
The following are all distinct and only binary trees having three nodes



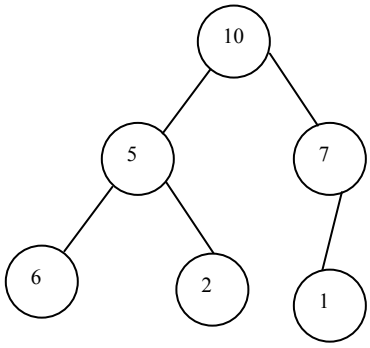
Heap: is defined as a binary tree with keys assigned to its nodes (one key per node) such that the following conditions are satisfied:

(i) The binary tree is essentially complete (or simply complete), i.e, all its levels are full except possibly the last level where only some rightmost leaves may be missing.
(ii) The key at each node is greater than or equal to the keys at its children.

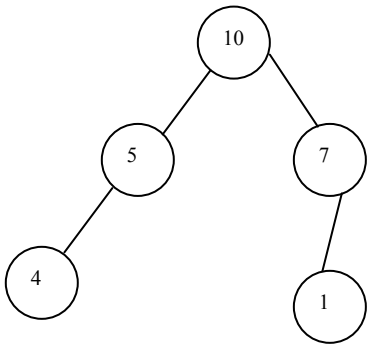
The following binary tree is a Heap



However, the following is not a heap because the value 6 in a child node is more than the value 5 in the parent node.



Also, the following is not a heap, because, some leaves (e.g., right child of 5), in between two other leaves (viz 4 and 1), are missing.



Alternative Definition of Heap:

Heap is an array H[1..n] in which every element in position i (*the parent) in the first half* of the array is greater than or equal to elements in positions 2i and and 2i+1 (*the children*):

HEAP SORT is a three-step algorithm as discussed below:

(i) *Heap Construction* for the a given array

(ii) (*Maximum deletion*) Copy the root value (which is maximum of all values in the Heap) to right-most yet-to-be occupied location of the array used to store the sorted values and copy the value in the last node of the tree (or of the corresponding array) to the root.

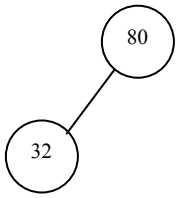
(iii) Consider the binary tree (which is not necessarily a Heap now) obtained from the Heap through the modifications through Step (ii) above and by removing currently the last node from further consideration. Convert the binary tree into a Heap by suitable modifications.

Example 3.3.5.1:

Let us consider applying Heap Sort for the sorting of the **list 80 32 31 110 50 40 120** represented by an array A[1…7]

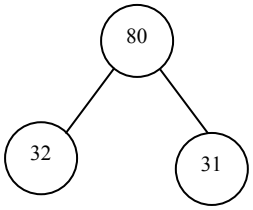
Step 1: **Construction of a Heap for the given list**
First, we create the tree having the root as the only node with node-value 80 as shown below:

Next value 32 is attached as left child of the root, as shown below



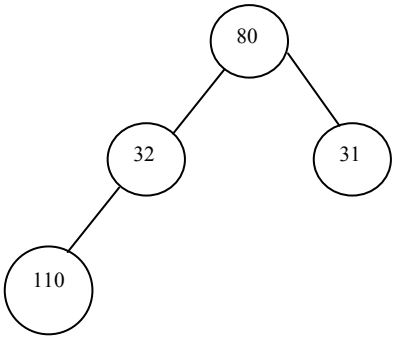
As 32 < 80, therefore, heap property is satisfied. Hence, no modification of the tree.

Next, value 31 is attached as right child of the node 80, as shown below

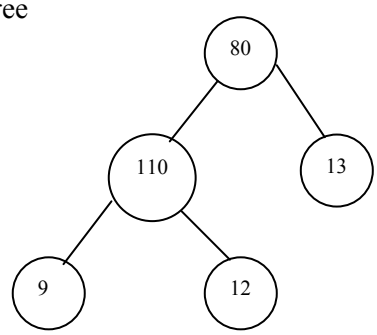


Again as 31 < 80, heap property is not disturbed. Therefore, no modification of the tree.

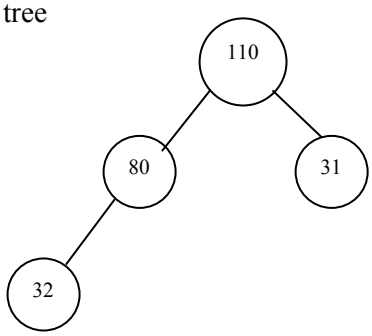
Next, value 110 is attached as left child of 32; as shown below.



However, *110 > 32*, the value in child node is more than the value in the parent node. Hence the *tree is modified* by exchanging the values in the two nodes so that, we get the following tree

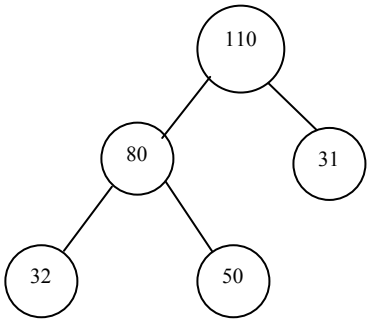


Again as *110 > 80*, the value in child node is more than the value in the parent node. Hence the *tree is modified* by exchanging the values in the two nodes so that, we get the following tree

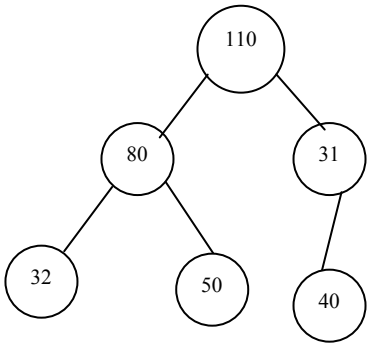


This is a Heap.

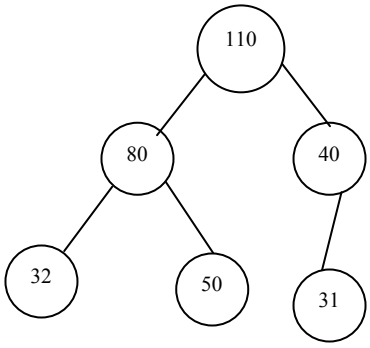
Next, number 50 is attached as right child of 80 so that the new tree is as given below



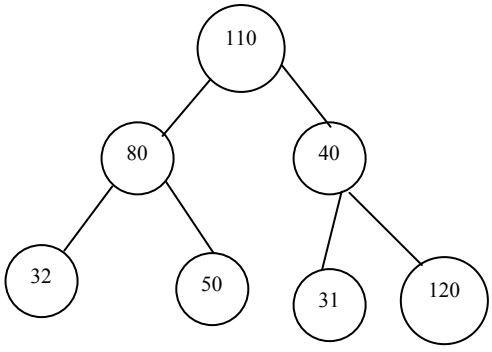
As the tree satisfies all the conditions of a Heap, we insert the next number 40 as left child of 31 to get the tree



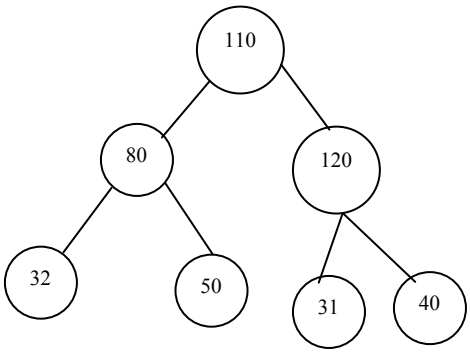
As the new insertion violates the condition of Heap, the values 40 and 31 are exchanged to get the tree which is a heap



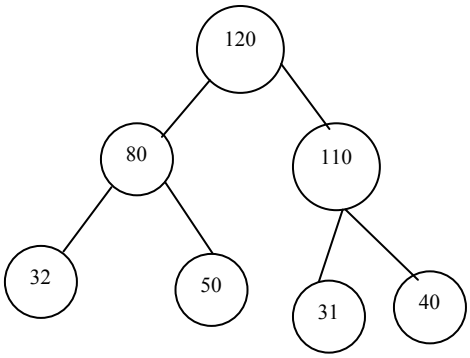
Next, we insert the last value 120 as right child of 40 to get the tree



The last insertion *violates* the conditions for a Heap. Hence 40 and 120 are exchanged to get the tree



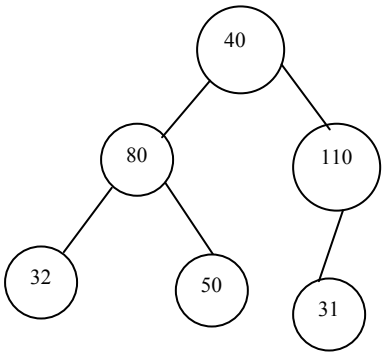
Again, due to movement of 120 upwards, *the Heap property is disturbed at nodes `110 and 120. Again 120 is moved up* to get the following tree which is a heap.



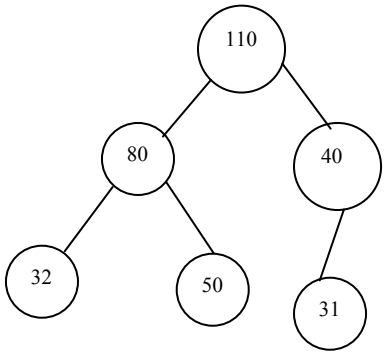
After having constructed the Heap through Step 1 above we consider

***Step 2&3*:** Consists of a sequence of actions viz (i) deleting the value of the root, (ii) **moving the last entry to the root and (iii) then** readjusting the Heap
The root of a Heap is always the maximum of all the values in the nodes of the tree. The value 120 currently *in the root is saved in the last location B[n] in our case B*[*7*] of the sorted array say B[1..n] in which the values are to be stored after sorting in increasing order.

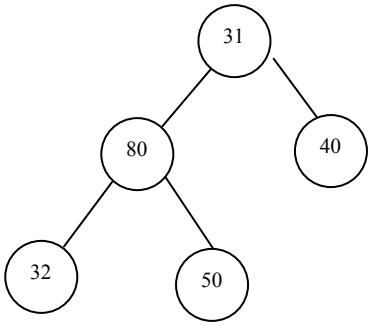
Next, value 40 is moved to the root and the node containing 40 is removed from further consideration, to get the following binary tree, which is *not* a Heap.



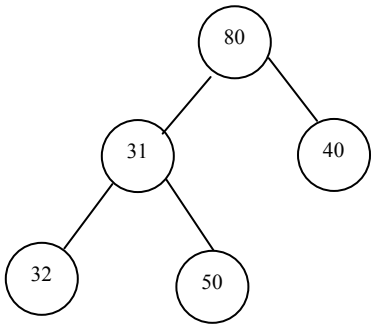
In order to restore the above tree as a Heap, the value 40 is exchanged with the maximum of the values of its two children. Thus 40 and 110 are exchanged to get the tree which is a Heap.



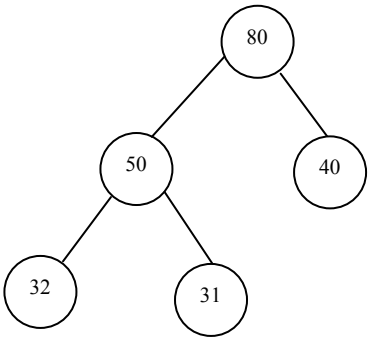
Again 110 is copied to B[6] and 31, the last value of the tree is shifted to the root and last node is removed from further consideration to get the following tree, which is not a Heap



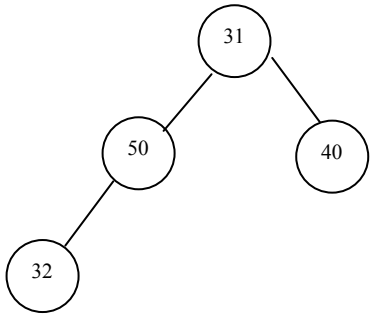
Again the root value is exchanged with the value which is maximum of its children's value i.e exchanged with value 80 to get the following tree, which again is *not* a Heap.



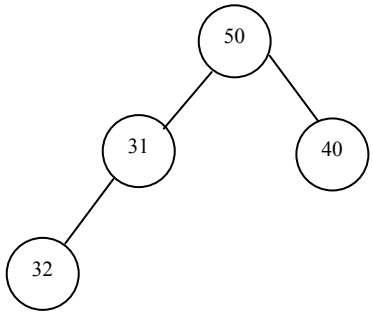
Again the value 31 is exchanged with the maximum of the values of its children, i.e., with 50, to get the tree which is a heap.



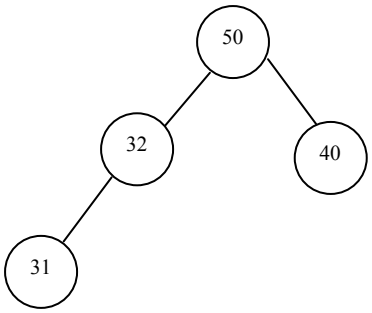
Again 80 is copied in B[5] and 31, the value of the last node replaces 80 in the root and the last node is removed from further consideration to get the tree which is not a heap.



Again, 50 the maximum of the two children's values is exchanged with the value of the root 31 to get the tree, which is *not* a heap.

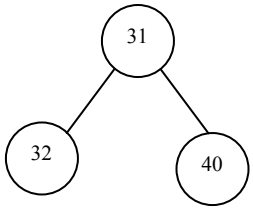


Again 31 and 32 are exchanged to get the Heap

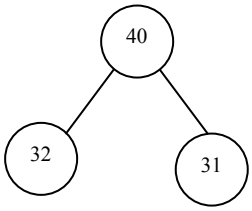


Next, 50 is copied in B[4].

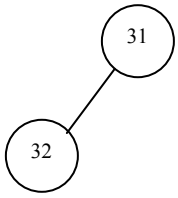
The entry 31 in the last node replaces the value in the root and the last node is deleted, to get the following tree which is *not* a Heap



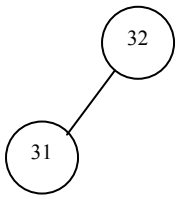
Again 40, the maximum of the values of children is exchanged with 31, the value in the root. We get the Heap



Again 40 is copied in B[3]. The value in the last node of the tree viz 31, replaces the value in the root and the last node is removed from further consideration to get the tree, which is not a Heap.



Again 32, the value of its only child is exchanged with the value of the root to get the Heap



Next, 32 is copied in B[2] and 31, the value in the last node is copied in the root and the last node is deleted, to get the tree which is a Heap.

This value is copied in B[1] and the Heap Sort algorithm terminates.

Next, we consider the two procedure

1. Build-Heap and
2. Delete-Root-n-Rebuild.

which constitute the Heap-Sort algorithm. The list L of n numbers to be sorted is represented by an array A[1..n]

The following procedure reads one by one the values from the given to-be-sorted array A[1..n] and gradually builds a Heap. For this purpose, it calls the procedure *Build-Heap*. For building the Heap, an array H[1..n] is used for storing the elements of the Heap. Once the Heap is built, the elements of A[1..n] are already sorted in H[1..n] and hence A may be used for sorting the elements of finally sorted list for which we used the array B. Then the following three steps are repeated n times, (n, the number of elements in the array); in the ith iteration.

(i) The root element H[1] is copied in A[n − i+1] location of the given array A. The first time, root element is stored in A[n]. The next time, root element is stored in A[n −1] and so on.

(ii) The last element of the array H[n − i + 1] is copied in the root of the Heap, i.e., in H[1] and H[n − i + 1] is removed from further consideration. In other words, in the next iteration, only the array H[1..(n − i)] (*which may not be a Heap*) is taken into consideration.

(iii) The procedure *Build-Heap* is called to build the array H[1..(n − i)] into a Heap.

Procedure Heap-Sort (A [1 .. n]: real)
begin {*of the procedure*}
H[1..n] ← Build-Heap (A[1..n])
For i ← 1 to n do

```
      begin {for loop}
       A [n – i + 1] ← H[1]
           H[1] ← H[n – i + 1]
       Build-Heap (H [1.. (n – i)])
      end {for-loop}
end {procedure}
```

Procedure Build-Heap

The following procedure takes an array B[1..m] of size m, which is to be sorted, and builds it into a Heap

```
Procedure Build-Heap (B[1..m]: real)
 begin {of procedure}
  for j = 2 to m do
    begin {of for-loop}
      location ← j
    while (location > 1)  do
     begin
         Parent ← [location/2]
          If A[location] ≤ A [parent] then return {i.e., quit while loop}
          else
        {i.e., of A[location] > A [parent] then}
       begin {to exchange A [parent] and A [location]}
        temp ← A [parent]
        A[parent] ← A [location]
        A [location] ← temp
       end {of if .. then .. else}
           location ← parent
   end {while loop}
    end {for loop}
 end {of procedure}
```

Ex. 5) Sort the following sequence of number, using Heap Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.6 Divide and Conquer Technique

The 'Divide and Conquer' is a technique of solving problems from various domains and will be discussed in details later on. Here, we briefly discuss how to use the technique in solving sorting problems.

A sorting algorithm based on '*Divide and Conquer*' technique has the following outline:

```
    Procedure  Sort (list)
        If the list has length 1 then return the list
        Else {i.e., when length of the list is greather than 1}
     begin
                Partition the list into two sublists say L and H,
                Sort (L)
                Sort (H)
                Combine (Sort (L)), Sort (H))
{during the combine operation, the sublists are merged in sorted order}
    end.
```

There are two well-known 'Divide and conquer' methods for sorting viz:

(i) Merge Sort
(ii) Quick Sort

3.3.7 Merge Sort

In this method, we recursively chop the list into two sublists of **almost equal** sizes and when we get lists of sizes one, then start sorted merging of lists in the reverse order in which these lists were obtained through chopping. The following example clarifies the method.

Example 3.3.7.1 of Merge Sort:

Given List: 4 6 7 5 2 1 3
Chop the list to get two sublists viz.
((4, 6, 7, 5), (2 1 3))
where the symbol '/ ' separates the two sublists

Again chop each of the sublists to get two sublists for each viz
(((4, 6), (7, 5))), ((2), (1, 3)))
Again repeating the chopping operation on each of the lists of size two or more obtained in the previous round of chopping, we get lists of size 1 each viz 4 and 6, 7 and 5, 2, 1 and 3. In terms of our notations, we get
((((4), (6)), ((7), (5))), ((2), ((1), (3))))
At this stage, we start merging the sublists in the reverse order in which chopping was applied. *However, during merging the lists are sorted.*

Start merging after sorting, we get sorted lists of at most two elements viz
(((4, 6), (5, 7)), ((2), (1, 3)))
Merge two consecutive lists, each of at most two elements we get sorted lists
((4, 5, 6, 7), (1, 2, 3))
Finally merge two consecutive lists of at most 4 elements, we get the sorted list: (1, 2, 3, 4, 5, 6, 7)

Procedure ***mergesort*** **(A [1..n])**
if **n > 1** then
m ← [n/2]
L_1← A[1..m]
L_2← A[m+1..n]
L← merge (mergesort (L_1), mergesort (L_2))
end

begin {*of the procedure*}
{***L is now sorted with elements in nondecreasing order***}
Next, we discuss merging of already sorted sublists
Procedure ***merge*** **(L_1, L_2: lists)**
L ← empty list
While **L_1 and L_2 are both nonempty do**
begin
Remove smaller of the first elements of L_1 and L_2 from the list it is in and place it in L, immediately next to the right of the earlier elements in L. If **removal of this element makes one list empty** then **remove all elements from the other list and append them to L keeping the relative order of the elements intact.**
else **repeat the process with the new lists L_1 and L_2**
end
{***L is the merged list with elements sorted in increasing order***}

Ex. 6) Sort the following sequence of number, using Merge Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.8 Quick Sort

Quick Sort is also a '*divide and conquer*' method of sorting. It was designed by C.A.R. Hoare, one of the pioneers of Computer Science and also Turing Award Winner for the year 1980. This method does **more work** in the **first step of partitioning the list** into two sublists. Then combining the two lists becomes trivial.

To partition the list, we first choose **some** value from the list for which, we hope, about half the values will be less than the chosen value and the remaining values will be more than the chosen value.

Division into sublists is done through the choice and use of a **pivot value,** which is a value in the given list so that all values in the list less than the pivot are put in one list and rest of the values in the other list. The process is applied recursively to the sublists till we get sublists of lengths one.

Remark 3.3.8.1:

The choice of pivot has significant bearing on the efficiency of Quick-Sort algorithm. Sometime, the very first value is taken as a pivot.
However, the **first values** of given lists may be poor choice, specially when the given list is already ordered or nearly ordered. Because, then one of the sublists may be empty. For example, for the list
7 6 4 3 2 1
the choice of first value as pivots, yields the list of values greater than 7, the pivot, as empty.

Generally, some **middle value** is chosen as a pivot.

Even, choice of middle value as pivot may turn out to be very poor choice, e.g, for the list
2 4 6 **7** 3 1 5
the choice of middle value, viz 7, is not good, because, 7 is the maximum value of the list. Hence, with this choice, one of the sublists will be empty.

A *better method for the choice of pivot position* is to use a random number generator to generate a number j between 1 and n, for the position of the pivot, where n is the size of the list to be sorted. Some simpler methods take median value of the of a sample of values between 1 to n, as pivot position. For example, the median of the values at first, last and middle (or one of the middle values, if n is even) may be taken as pivot.

Example 3.3.8.1 of Quick Sort

We use two indices (*viz i & j, this example*) one moving from left to right and other moving from right to left , **using the first element as pivot**
In each iteration, the index i while moving from left to the right, notes the position of the value first from the left that is greater than pivot and stops movement for the iteration. Similarly, in each iteration, the index j while moving from right to left notes the position of the value first from the right that is less than the pivot and stops movement for the iteration. The values at positions i and j are exchanged. Then, the next iteration starts with current values of i and j onwards.

5 3 1 **9** 8 2 **4** 7 {← *Given list*}
i → ← j

5 3 1 **9** 8 2 **4** 7
i j

{*the value 9 is the first value from left that is greater than the pivot viz 5 and the value 4 is the first value from right that is less than the pivot. These values are exchanged to get the following list*}

5 3 1 4 **8** **2** 9 7
i j

{*Moving i toward right and j toward left and i stops, if it reaches a value greater than the pivot and j stops if j meets a value less than the pivot. Also both stop if j ≤ i. We get the list*}

5 3 1 4 8 2 9 7
i ↔ j

{*After exchanging the values 2 and 8 we get the list*}

5 3 1 4 **2** 8 9 7
i j

{*The next moves of i to the right and j to the left make j < i and this indicates the completion of one iteration of movements of i and j to get the list with positions of i and j as follows*}

5 3 1 4 2 8 9 7
j i

{*At this stage, we exchange the pivot with value at position j, i.e., 2 and 5 are exchanged so that pivot occupies almost the middle position as shown below.*}

2 3 1 4 **5** 8 9 7

{It may be noted that all values to the left of 5 are less than 5 and all values to the right of 5 are greater than 5. Th*en the two sublists viz 2,3,1,4 and 8,9,7 are sorted independently*}

2 **3** **1** 4 and 8 7 9
i → ← j i → ← j

2 3 1 4 and 8 7 9
i ↔ j i j

2 1 3 4 and 8 7 9
i j j i

{as j < i, e*xchanging the pivot values with 7, value at jth position, we get*}

{*moving i and j further we get*}

2 **1** 3 4 and 7 8 9
j i i j

{*as j < i, exchange pivot 2 with value 1 at j, we get neglecting the pivot the two sublists to be sorted as 1 and 3, 4*}

1 **2** **3** 4
and i j,
3 4
j i

{*Pivot position = 1 = j. And pivot position is neglected therefore, we need to sort the remaining lists which are respectively empty and {4}, which are already sorted. This completes the sorting. Merging does not require any extra time as already the entries are in sorted order*}

On basis of the discussion above, we now formalize the Quick Sort algorithm/procedure

```
Procedure      Quick-Sort (A[min… max])
  {min is the lower index and max the upper index of the array to be sorted using Quick Sort}
      begin
        if  min < max then
              p ← partition (A[min..max]);
                 {p is the position s.t for  min ≤ i ≤ p − 1,      A [i] ≤ A [p] and  for all j ≥ p+1,
                  A[j] ≥ A [p]}
               Quick-Sort (A [min .. p − 1]);
               Quick-Sort (A[p+1 .. max]);
       end;
```

In the above procedure, the procedure *partition* is called. Next, we define the procedure partition.

```
Procedure      partition (A [min .. max])
i, j: integer;
s: real;
   {In this procedure the first element is taken as pivot; the index i, used below moves from left to right
    to find first value i = 1_i  from left such that A[v_1 ] > A[1].  Similarly j moves from right to left  and
    finds first value j = v_2  such that A[v_2] < A[1]. If j >i, then A[i] and A[j] are exchanged.  If j ≤ I
    then A[1] and A[j] are exchanged}.
    s ← A [min]; i ← min + 1; j ← max
while (i < j)   do

      begin
        While (A[i] < p)       do
              i ← i +1
         While (A[j] > p)       do
              j ← j − 1
         exchange (A[i], A[j])
           {the exchange operation involves three assignments, viz, temp ← A [i ];  A[i] ← A [j] and
           A[j] ← temp, where
             temp is a new variable }
      end;   {of while loop}
      exchange  (A[1], A[j]);
      return j
          {the index j is s.t in the next iteration sorting is to be done for  the two subarray
           A[ min .. j−1] and  A[j+1 .. max]
```

Ex. 7) Sort the following sequence of number, using Quick Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.9 Comparison of Sorting Algorithms

We give below *just the summary* of the time complexities of the various algorithms discussed above. Some of these time complexities will be derived at different places, after sufficient tools are developed for the purpose. Here these are included so that we may have an idea of the comparative behaviors of the algorithms discussed above. Each algorithm requires time for execution of two different types of actions, viz

(i) Comparison of keys
(ii) assignment of values

The following table gives the time complexities requirement in terms of size n of the list to be sorted, for the two types of actions for executing the algorithms. Unless mentioned otherwise, the complexities are for *average case behaviors* of the algorithms:

Name of the algorithm	Comparison of Keys	Assignments
Selection Sort	$0.5n^2 + O(n)$	$3.0\ n + O(1)$
Insertion Sort	$0.25n^2 + O(n)$ (*average)*	$0.25n^2 + O(n)$
Shell Sort	$n^{1.25}$ to $1.6n^{1.25}$ for large n (*empirically*)	
Heap Sort	$2\ n \log n + O(n)$ (*worst case*)	$n \log n + O(n)$
Bubble Sort	$\frac{1}{2}(n^2 - n \log n)$ (*average*)	$\frac{1}{4}(n^2 - n)$ (*average*)
	$\frac{1}{2}(n^2 - n)$ (*worst*)	$\frac{1}{2}(n^2 - n)$ (*worst*)
Quick Sort	$1.39\ n \log n$	$0.69\ n \log n$
Merge Sort	$n \log n$ to $(n \log n - 1.583n + 1)$ (*for linked lists*)	$n \log n$ (*for contiguous lists*)

Merge Sort is good for linked lists but poor for contiguous lists. On the other hand, Quick Sort is good for contiguous lists but is poor for linked lists.
In context of complexities of a sorting algorithm, we state below an important theorem without proof.

Theorem: Any algorithm, *based on comparison of keys*, that sorts a list of n elements, in its average case must perform at least $\log (n!) \approx (n \log n + O(n))$ number of comparisons of keys.

3.4 BEST-CASE AND WORST CASE ANALYSES

For certain algorithms, the times taken by *an algorithm on two different instances*, may differ considerably. For this purpose, consider the algorithm *Insertion-Sort, already discussed, for sorting n given numbers, in* ***increasing order***.

Again, for this purpose, one of the two lists is TS[1..n] in which the *elements are already sorted* in *increasing* order, and the other is **TR[1..n]** in which *elements are in reverse-sorted order*, i.e., elements in the list are in *decreasing order*.

The algorithm *Insertion-Sort* for the array TS[1..n] already sorted in increasing order, does not make any displacement of entries (i.e., values) for making room for out-of-order entries, because, there are no out-of-order entries in the array.

In order to find the complexity of Insertion Sort for the array TS[1..n], let us consider a specific case of sorting.

The already sorted list {1,3,5,7,9}. Initially A[1] = 1 remains at its place. For the next element A[2] = 3, at its place. For the next element A[2] = 3, the following operations are performed by Insertion Sort:

(i) $j \leftarrow 2$
(ii) $i \leftarrow j - 1$
(iii) $m \leftarrow A[j]$
(iv) $A[i + 1] \leftarrow m$
(v) $m < A[i]$ in the while loop.

We can see that these are the minimum operation performed by Insertion Sort irrective of the value of A[2]. Further, had A[2] been less than A[1], then more operations would have been performed, as we shall see in the next example.

To conclude, as if A[2] ≥ A[1] (as is the case, because A[2] = 3 and A[1] = 1) then Insertion Sort performs 4 assignments and 1 comparison. We can see that in general, if A[l+1] ≥ A[l] then we require (exactly) 4 additional assignments and 1 comparison to place the value A[l+1] in its correct position, viz (l+1)th.

Thus in order to use Insertion Sort to attempt to sort an already, in proper order, sorted list of n elements, we need 4(n−1) assignments and (n − 1) comparison.

Further, we notice, that these are the minimum operations required to sort a list of n elements by Insertion Sort.

Hence in the case of *array TS*[*1..n*], the Insertion-Sort algorithm *takes linear time* for an already sorted array of n elements. **In other words, Insertion-Sort has linear *Best-Case* complexity for TS[1..n].**

Next, we discuss the number of operations required by Insertion Sort for sorting TR[1..n] which is sorted in *reverse order.* For this purpose, let us consider the sorting of the list {9, 7, 5, 3, 1} stored as A[1] = 9, A[2] = 7, A[3] = 5 A[4] = 3 and A[5] = 1. Let m denote the variable in which the value to be compared with other values of the array, is stored. As discussed above in the case of already properly sorted list, 4 assignments and one comparison, of comparing A[k +1] with A[k], is *essentially* required to start the process of putting each new entry A[k+1] after having already sorted the list A[1] to A[k]. However, as 7 = A[2] = m < A[1] = 9. A[1] is copied to A[2] and m is copied to A[1]. Thus Insertion Sort requires one more assignment (viz A[2] ← A[1]).
At this stage A[1] = 7 and A[2] = 9. *It is easily seen that for a list of two elements, to be sorted in proper order using Insertion-Sort, at most one comparison and 5 (= 4 +1) assignments are required.*

Next, m ← A[3] = 5, and m is first compared with A[2] = 9 and as m < A[2], A[2] is copied to A[3]. So, at this stage both A[2] and A[3] equal 9.

Next, m = 5 is compared with A[1] = 7 and as m < A[1] therefore A[1] is copied to A[2], so that, at this stage, both A[1] and A[2] contain 7. Next 5 in m is copied to A[1]. Thus at this stage A[1] = 5, A[2] = 7 and A[3] = 9. And, during the last round, we made two additional comparisons and two addition assignments (viz A[3] ← A[2] ← A[1]), and hence total (4 + 2) assignments, were made.

Thus, so far we have made 1+ 2 comparisons and 5 + 6 assignments.
Continuing like this, for placing A[4] = 3 at right place in the sorted list {5,7,9}, we make 3 comparisons 7 assignments. And for placing A[5] = 1 at the right place in the sorted list {3,5,7,9}, we make 4 comparisons and 8 assignments. Thus, in the case of a list of 5 elements in reverse order, the algorithm takes $1+2+3+4 = \frac{4 \times 5}{2}$ comparisons and 5+6+7+8+9 = 4 × 5 + (1+2+3+4+5) assignments.

In general, for a list of n elements, *sorted in reverse order,* the Insertion Sort algorithm makes $\frac{(n-1)n}{2} = \frac{1}{2}(n^2 - n)$ comparisons and $4n + \frac{1}{2}(n^2 + n)$ $= \frac{1}{2}(n^2 + 9n)$ assignments.

Again, it can be easily seen that for a list of n elements, Insertion-Sort algorithm should make *at most* $\frac{1}{2}$ ($n^2 - n$) comparisons and $\frac{1}{2}$ ($n^2 + 9n$) assignments.

Thus, the total number of operations

$$=\left(\frac{1}{2}\left(n^2 - n\right)\right)+\left(\frac{1}{2}\left(n^2 + n\right)+4n\right) = n^2 + 4n$$

If we assume that time required for a comparison takes a constant multiple of the time than that taken by an assignment, then time-complexity of Insertion-Sort *in the case of* ***reverse-sorted*** *list* is a **quadratic in n.**

When actually implemented for n = 5000, the Insertion-Sort algorithm took 1000 times more time for TR[1..n], the array which is sorted in reverse order than for TS[1..n], the array which is already sorted in the required order, to sort both the arrays in increasing order.

3.4.1 Various Analyses of Algorithms

Earlier, we talked of time complexity of an algorithm as a function of the size n of instances of the problem intended to be solved by the algorithm. However, from the discussion above, we have seen ***that for a given algorithm and instance size n*,** the execution times for two different instances of the same size n, may differ by a factor of 1000 or even more. This necessitates for us to consider *further differentiation or classification of complexity analyses* of an algorithm, which take into consideration not only the size n but also types of instances. Some of the well-known ones are:

(i) worst case analysis
(ii) best case analysis
(iii) average case analysis
(iv) amortized analysis

We discuss worst-case analysis and best-case analysis in this section and the other two analyses will be discussed in Section 2.9.

3.4.2 Worst-Case Analysis

Worst-Case analysis of an algorithm for a given problem, involves finding the **longest** of all the times that can (*theoretically*) be taken by various instances of a given size, say n, of the problem. The worst-case analysis may be carried out by first (i) finding the instance types for which algorithm runs slowest and then (ii) finding running time of the algorithm for such instances. If **c-worst (n)** denotes the worst-case complexity for instances of size n, then by the definition, it is guaranteed that no instance of size n of the problem, shall take more time then c-worst (n).
Worst-Case analysis of algorithms is appropriate for problems in which response-time is critical. For example, in the case of problems in respect of controlling of nuclear power plants, it is important to know an upper limit on the system's response time than to know the time of execution of particular instances.

In the case of the Insertion-Sort algorithm discussed earlier, we gave an outline of argument in support of the fact that Insertion-Sort algorithm takes *longest time*

(i) for lists sorted in reverse order and

(ii) if the size of such a list is n, then the longest time should correspond to $\frac{1}{2}\left(n^2 + n\right)$ comparisons and $\frac{1}{2}\left(n^2 + 3n\right)$ assignments.

In other words, the worst time complexity of Insertion-Sort algorithm is a quadratic polynomial in the size of the problem instance.

3.4.3 Best-Case Analysis

Best- Case Analysis of an algorithm for a given problem, involves finding the **shortest** of all the times that can (*theoretically*) be taken by the various instances of a given size, say n, of the problem. In other words, the best-case analysis is concerned with

(i) finding the instances (types) for which algorithm runs **fastest** and then
(ii) with finding running time of the algorithm for such instances (types).

If ***C-best* (*n*) denotes** the best-case complexity for instances of size n, then by definition, it is guaranteed that *no instance* of size n, of the problem, shall take *less time than*
C-best (*n*).

The best-case analysis is not as important as that of the worst-case analysis. However, the best-case analysis may be useful guide for application to situations, which need not necessarily correspond to the instance types taking shortest time, yet which are close to such instance types. For example, a telephone directory of a metropolitan city like Delhi, contains millions of entries already properly sorted. Each month, if a few thousand of new entries are to made, then if these entries are put in the beginning of the directory, then in this form, without further processing, the directory is a *nearly sorted* list. And, the Insertion-Sort algorithm which makes only n comparisons and no shifts for an already properly sorted list of n elements sorted in the required order, may be useful for application to this *slightly out-of-order* new list obtained after addition of a few thousand entries in the beginning of the earlier sorted list of millions of entries.

In general, it can be shown that

(i) for an already sorted list of w elements, sorted in the required order, the

Insertion-Sort algorithm will make $(n - 1)$ comparisons and $4 \times (n - 1)$ assignments and

(ii) $(n - 1)$ comparisons and $4 (n - 1)$ assignments are the minimum that are required of sorting any list of n elements, that is already sorted in the required order.

Thus the best time complexity of Insertion Sort algorithm is a linear polynomial in the size of the problem instance.

3.5 ANALYSIS OF NON-RECURSIVE CONTROL STRUCTURES

Analysis of algorithms is generally a *bottom-up process* in which, first we consider the time required for *executing individual instructions*. Next, we determine recursively time required by more complex structures where times required for less complex structures are already calculated, already known, given or assumed. We discuss below how the time required for structures obtained by applying *basic structuring rules* for sequencing, repetition and recursion, are obtained, provided times required for individual instructions **or component program fragments**, are already known or given. First we consider the structuring rule: sequencing.

3.5.1 Sequencing

Let F_1 and F_2 be two program fragments, with t_1 and t_2 respectively the time required for executing F_1 and F_2. Let the program fragment $F_1 ; F_2$ be obtained by sequencing the two given program fragments, i.e, *by writing* F_1 *followed by* F_2.

Then **sequencing rule** states that the time required for the program fragment $F_1 ; F_2$ *is* $t_1 + t_2$.

Word of Caution: *The sequencing rule, mentioned above, is valid only under the assumption that no instruction in fragment F_2 depends on any instruction in Fragment F_1. Otherwise, instead of $t_1 + t_2$, the time required for executing the fragment $F_1 ; F_2$ may be some more complex function of t_1 and t_2, depending upon the type of dependency of instruction(s) of F_2 on instructions of F_1. Next, we consider the various iterative or looping structures, starting with "For" loops.*

3.5.2 *For* Construct

In order to understand better the ideas involved, let us first consider the following two simple examples involving *for* construct.

Example 3.5.2.1: The following program fragment may be used for computing sum of
first n natural numbers:

```
for i = 1 to n do
    sum = sum + i.
```

The example above shows that the instruction *sum = sum + i* depends upon the loop variable 'i'. Thus, if we write

```
P(i) :  sum = sum + i
```

then the above-mentioned 'for' loop may be rewritten as

```
for i= 1 to n do
  P (i),
end {for}
```

where i in P(i) indicates that the program fragment P depends on the loop variable i.

Example 3.5.2.2: The following program fragment may be used to find the sum of n numbers, each of which is to be supplied by the user:

```
for i = 1 to n do
    read (x);
   sum = sum + x;
end {for}.
```

In the latter example, the program fragment P, consisting of two instructions viz., *read* (*x*) and *sum = sum + x*, do not involve the loop variable i. But still, there is nothing wrong if we write P as P(i). This is in view of the fact that a function f of a variable x, given by

$f(x) = x^2$

may also be considered as a function of the two variables x and y, because

$f(x,y) = x^2 + 0.y$

Remark 3.5.2.3:

The *for* loop
for i = 1 to n do
P(i) ;
end for,
is actually a shorthand for the following program fragment

i = 1
while i ≤ n do
P(i) ;
i =i + 1
end while ;

Therefore, we need to take into consideration the above-mentioned fact while calculating the time required by the *for loop* fragment.

Remark 3.5.24:

The case when n = 0 in the loop *for i = 1 to n do P* (*i*) would not be treated as an error. The case *n = 0* shall be interpreted that P(i) *is not executed even once*.

Let us now calculate the time required for executing the loop

for i = 1 to n do
P(i).
end for

For this purpose, we use the expanded definition considered under Remark and, in addition, we use the following notations:

fl : the time required by the 'for' loop

a : the time required by each of the assignments
i= 1 and i = i+ 1;

c : for each of the test i ≤ n and
s: for sequencing the two instructions P(i) and i = i+1 in the While loop.

t : the time required for executing P(i) once.

Then, it can be easily seen that

fl	=	a	*for i = 1*
		+ (n+1) c	*for (n+1) times testing i ≤ n*
		+ n t	*for n times execution of P(i)*
		+ n a	*for n times execution of i = i+1*
		+ n s	*for n times sequencing P(i) and i = i + 1.*

i.e.

fl = (n+1)a + (n+1)c + n s + n t

In respect of the last equality, we make the following observations

(i) the quantity on R.H.S is bounded below by nt, n times the time of execution of P(i)

(ii) if t, the time of execution of P(i) is much larger than each of

(a) a, the time for making an assignment
(b) c, the time for making a comparison and
(c) s, the time for sequencing two instructions, one after the other,
then fl, the time to be taken by the 'for' loop is approximately nt, i.e., fl ≈ nt

(iii) If n = 0 or n is negative, the approximation fl ≈ nt is completely wrong. Because , w.r.t Remark 2.6.2.1 above, at least the assignment i = 1 is executed and also the test i ≤ n is executed at least once and hence fl ≤ 0 or fl ≈ 0 can not be true.

3.5.3 *While* and *Repeat* Constructs

From the point of view of time complexity, the analysis of *while* or *repeat* loops is more difficult as compared to that of *for* loops. This is in view of the fact that in the case of *while* or *repeat* loops, we do not know, in advance, how many times the loop is going to be executed. But, in the case of *for* loops, we can know easily, in advance the number of times, the loop is going to be executed.

One of the frequently used techniques for analysing while/repeat loops, is *to define a function* say f of the involved loop variables, in such a way that

(i) the value of the *function f* is an integer that decreases in successive iterations of the loop.

(ii) the *value of f remains non-negative* throughout the successive executions of the loop, and as a consequence.

(iii) the value of f reaches some minimum non-negative value, when the loop is to terminate, of course, only if the loop under consideration is a terminating loop.

Once, such a *function* f, if it exists, is found, the analysis of the *while/repeat* loop gets simplified and can be accomplished just by close examination of the sequence of successive values of f.

We illustrate the techniques for computing the time complexity of a ***while loop*** through an example given below. The ***repeat loop*** analysis can be handled on the similar lines.

Example 3.5.3.1:

Let us analyze the following *Bin-Search algorithm* that finds the location of a *value v* in an already sorted *array A[1..n*], where it is given that v occurs in A[1.. n]
*The Bin-Search algorithm, as defined below, is, in its rough version, intuitively applied by us in finding the meaning of a given word in a dictionary or in finding out the telephone number of a person from the telephone directory, where the name of the person is given. In the case of dictionary search, if the word to be searched is say **CARTOON**, then in view of the fact that the word starts with letter **C** which is near the beginning of the sequence of the letters in the alphabet set, we open the pages in the dictionary near the beginning. However, if we are looking for the meaning of the word REDUNDANT, than as R is 18th letter out of 26 letters of English alphabet, we generally open to pages after the middle half of the dictionary.*

However, in the case of search of a known value v in a given sorted array, the values of array are not know to us. Hence we do not know the relative position of v. This is why, we find the value at the middle position $\left\lceil \frac{1+n}{2} \right\rceil$ in the given sorted array A[1..n] and then compare v with the value A $\left\lceil \frac{1+n}{2} \right\rceil$. These cases arise:

(i) If the value v = A $\left\lceil \frac{n+1}{2} \right\rceil$, then search is successful and stop. Else,

(ii) if $v < A\left[\frac{1+n}{2}\right]$, then we search only the part $A\left[1..\left(\left[\frac{1+n}{2}\right]-1\right)\right]$ of the given array. Similarly

(iii) if $v > A\left[\frac{1+n}{2}\right]$, than we search only the part $A\left[\left(\frac{1+n}{2}\right)+1...n\right]$ of the array.

And repeat the process. The explanation for searching v in a sorted array, is formalized below as function Bin-Search.

```
Function Bin-Search ( A[1..n], v)
begin
  i = 1  ;    j = n
   while   i < j     do
     {i.e.,  A[ i ]  ≤ v ≤ A[ j ] }

        k  =  [(i+ j)   ÷  2]

        Case   v < A[ k ]:      j = k−1
               v = A[ k ]:      { return k }
               v > A[ k ]:      i = k + 1
        end case
      end while { return i }
end function;
```

We explain, through an example, how the Bin-Search defined above works.

Let the given sorted array A[1..12] be

1	4	7	9	11	15	18	21	23	24	27	30

and let v = 11. Then i = 1, j = 12 and

$$k = \left[\frac{1=12}{2}\right] = \left[\frac{13}{2}\right] = 6$$

and A[6] = 15
As v = 11 < 15 = A[6]

Therefore, for the next iteration

j = 6 − 1 = 5
i = 1 (unchanged)

$$\text{hence } k = \left[\frac{1+5}{2}\right] = 3$$

A [3] = 7
As v = 11 > 7 = A[3]

For next iteration
∴ i becomes (k+1) = 4, j remains unchanged at 5.

Therefore new value of $k = \left[\frac{4+5}{2}\right] = 4$

As v = 11 > 9 = A[k]

Therefore, in new iteration i becomes k + 1 = 5, j remains unchanged at 5.
Therefore new

$$k = \left\lfloor \frac{5+5}{2} \right\rfloor = 5$$

And A[k] = A[5] = 11 = v

Hence, the search is complete.

In order to analyse the above algorithm Bin-Search, let us define

Before analyzing the algorithm formally, let us consider tentative complexity of algorithm informally. From the algorithm it is clear that if in one iteration we are considering the array A [i...j] having j − i +1 elements then next time we consider either A [i .. k − 1] or A[(k + 1)..j] each of which is of length less than half of the length of the earlier list. Thus, at each stage length is getting at least halved. Thus, we expect the whole process to be completed in $\log 2^n >$ iterations.

We have seen through the above illustration and also from the definition that the array which need to be searched in any iteration is A[i..j] which has (j − i + 1) number of elements

Let us take f = j – i + 1 = length of the sequence currently being searched.
Initially f = n − 1+1 = n. Then, it can be easily seen that f satisfies the conditions (i), (ii) and (iii) mentioned above.

Also if f_{old}, j_{old} and i_{old} are the values of respectively f, j and i before an iteration of the loop and f_{new} , j_{new} and i_{new} the new values immediately after the iteration, then for each of the three cases v < A [k], v = A [k] and v > A [k] we can show that

$$f_{new} \leq f_{old}/2$$

Just for explanations, let us consider the case v < A [k] as follows:
for v < A [k], the instruction j = k − 1 is executed and hence

$i_{new} = i_{old}$ and $j_{new} = [(i_{old} + j_{old}) \div 2] - 1$ and hence

$$f_{new} = j_{new} - i_{new} + 1 = [(i_{old} + j_{old}) \div 2 - 1] - i_{old} + 1$$

$$\leq (j_{old} + i_{old})/2 - i_{old}$$
$$< (j_{old} - i_{old} + 1)/2 = f_{old}/2$$

$\therefore f_{new} < fold/2$
In other words, after each execution of the *while* loop, the length of the sub array of A [1.. n], that needs to be searched, if required, is less than half of the previous subarray to be searched. As v is assumed to be one the values of A [1 .. n], therefore, in the worst case, the search comes to end when j = i, i.e., when length of the subarray to be searched is 1. Combining with $f_{new} < f_{old}/2$ after each interaction, in the worst case, there will be t iterations satisfying

$$1 = (n/2^t) \quad \text{or} \quad n = 2^t$$

i.e., $t = \lfloor \log_2 n \rfloor$

Analysis of "repeat" construct can be carried out on the similar lines.

3.6 RECURSIVE CONSTRUCTS

We explain the process of analyzing time requirement by a recursive construct/ algorithm through the following example.

Example 3.6.1:

```
Function factorial (n)
        {computes n!, the factorial of n recursively
         where n is a non-negative integer}
begin
    if  n = 0 return 1
    else return (n * factorial (n– 1))
end factorial
```

Analysis of the above recursive algorithm

We take n, the input, as the *size of an instance* of the problem of computing factorial of n. From the above (recursive) algorithm, it is clear that multiplication is its basic operation. **Let M(n) denote the number of multiplications** required by the algorithm in computing factorial (n). The algorithm uses the formula

Factorial (n) = n* factorial (n–1) for n > 0.

Therefore, the algorithm uses one extra multiplication for computing factorial (n) then for computing factorial (n–1). Therefore,

$$M(n) = M(n-1) + 1. \quad \text{for } n > 0 \qquad (3.6.1)$$

Also, for computing factorial (0), no multiplication is required, as, we are given factorial (0) = 1. Hence

$$M(0) = 0 \qquad (3.6.2)$$

The Equation (3.6.1) does *not* define M(n) *explicitly* but defines *implicitly* through M(n–1). Such an equation is called a **recurrence relation**/equation. The equation (3.6.2) is called an **initial condition**. The equations (3.6.1) and (3.6.2) together form a **system of recurrences**.

By solution of a system of recurrences, we mean an explicit formula, say for M(n) in this case, free from recurrences in terms of n only, and not involving the function to be defined, i.e., M in this case, directly or indirectly.

We shall discuss in the next section, in some detail, how to solve system of recurrences. Briefly, we illustrate a method of solving such systems, called **Method of Backward Substitution**, through solving the above system of recurrences viz.

$$M(n) = M(n-1) + 1 \qquad (3.6.1)$$

and

$$M(0) = 0 \qquad (3.6.2)$$

Replacing n by (n – 1) in (2.7.1), we get $M(n-1) = M(n-2) + 1$ (3.6.3)
Using (2.7.3) in (2.7.1) we get

$$M(n) = M(n-1) + 1$$
$$= [M(n-2) + 1] + 1,$$

Repeating the above process by replacing n in (2.7.1) successively by (n – 2), (n – 3),……, 1, we get

$M(n) = M(n-1) + 1$	$= M(n-1) + 1$
$= [M(n-2) + 1] + 1$	$= M(n-2) + 2$
$= [M(n-3) + 1] + 2$	$= M(n-3) + 3$
$= M(1) + (n-1)$	$= M(n-I) + i$

$= M(0) + n$, Using (2.7.2) we get

$M(n) = 0 + n = n$

3.7 SOLVING RECURRENCES

In the previous section, we illustrated, with an example, the method of backward substitution for solving system of recurrences. We will discuss, in the next section, some more examples of the method.
In this section, we discuss some methods of solving systems of recurrences. However, the use of the methods discussed here, will be explained in the next section though appropriate examples.

3.7.1 Method of Forward Substitution

We explain the method through the following example.

Example 3.7.1.1:

Let us consider the system of recurrences

$$F(n) = 2F(n-1) + 1 \quad \text{for} \quad n > 1 \qquad (3.7.1)$$
$$F(1) = 1 \qquad (3.7.2)$$

First few terms of the sequence < F (n) > are, as given below.

$f(1) = 1$
$f(2) = 2 * f(1) + 1 = 2 \times 1 + 1 = 3$
$f(3) = 2 * f(2) + 1 = 2 \times 3 + 1 = 7$
$f(4) = f(3) + 1 = 2 \times 7 + 1 = 15$

Then, it can be easily seen that

$F(n) = 2^n - 1$ for $n = 1, 2, 3, 4$

We (intuitively) feel that

$F(n) = 2^n - 1$ for all $n \geq 1$

We attempt to establish the correctness of this intuition /feeling through Principle of Mathematical Induction as discussed.
As we mentioned there, it is a Three-Step method as discussed below:

***Step* (*i*):** We show for n = 1

$$F(1) = 2^1 - 1 = 1,$$

But F(1) = 1 is given to be true by definition of F given above.

***Step* (*ii*):** Assume for any k > 1

$$F(k) = 2^k - 1$$

***Step* (*iii*):** Show

$$F(k+1) = 2^{k+1} - 1.$$

For showing

$$F(k+1) = 2^{k+1} - 1,$$

Consider, by definition,

$$\begin{aligned} F(k+1) &= 2\,F\,(K + 1 - 1) + 1 \\ &= 2\,F\,(k) + 1 \\ &= 2\,(2^k - 1) + 1 \text{ (by Step (ii))} \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Therefore, by Principle of Mathematical Induction, our feeling that $F(n) = 2^n - 1$ for all $n \geq 1$ is mathematically correct.

Next, we discuss methods which can be useful in many practical problems.

3.7.2 Solving Linear Second-Order Recurrences with Constant Coefficients

Definitions: Recurrences of the form

$$a\,F(n) + b\,F(n-1) + cF(n-2) = g(n) \qquad (3.7.2.1)$$

where a, b, c are real numbers, $a \neq 0$, are called linear second-order recurrences with constant coefficients. Further, if $g(n) = 0$ then the recurrence is called **Homogeneous**. Otherwise, it is called **inhomogeneous**. Such systems of recurrences can be solved by neither **backward** substitution method nor by **forward** substitution method.

In order to solve recurrences of the form (2.8.2.1).

First we consider only the homogeneous case, i.e., when $g(n) = 0$. The recurrence becomes

$$a\,F(n) + b\,F(n-1) + c\,F(n-2) = 0 \qquad (3.7.2.2)$$

The above equation has infinitely many solutions except in the case when both $b = 0$ and $c = 0$.

With equation (2.8.2.2) we associate an auxiliary quadratic equation, called **characteristic equation**, given by

$$ax^2 + bx + c = 0 \qquad (3.7.2.3)$$

Then the solutions of the recurrences (2.8.2.1) are given by the following theorem, which we state without proof.

Theorem 3.7.2.1:

Let x_1 and x_2 be the solutions of the auxiliary quadratic equation

$a\,x^2 + bx + c = 0$. Then

Case 1: If x_1 and x_2 are real and distinct then solution of (2.8.2.2) is given by

$$F(n) = \alpha\, x_1^n + \beta\, x_2^n \qquad (3.7.2.4)$$

Where α and β are two arbitrary real constants.

Case II: If the roots x_1 and x_2 are real but $x_1 = x_2$ then solutions of (3.7.2.2) are given by

$$F(n) = \alpha x_1^n + \beta n x_1^n, \qquad (3.7.2.5)$$

Where, again, α and β are arbitrary real constants.

Case III: If x_1 and x_2 are complex conjugates given by u + iv, where u and v are real numbers. Then solutions of (2.8.2.2) are given by

$$F(n) = r^n [\alpha \cos n\theta + \beta \sin n\theta] \qquad (3.7.2.6)$$

where $r = \sqrt{u^2 + v^2}$ and $\theta = \tan^{-1}(v/u)$ and α and β are two arbitrary real constants.

Example 3.7.2.2:

Let the given recurrence be

$$F(n) - 4F(n-1) + 4F(n-2) = 0. \qquad (3.7.2.7)$$

Then, its characteristic equation is given by

$$x^2 - 4x + 4 = 0,$$

The two solutions equal, given by x = 2.

Hence, by (3.7.2.5) the solutions of (3.7.2.7) are given by

$$F(n) = \alpha 2^n + \beta n 2^n.$$

Next, **we discuss the solutions of the general (including inhomogeneous) recurrences**. The solution of the general second-order non-homogeneous recurrences with constant coefficients are given by the next theorem, which we state without proof.

Theorem 3.7.2.3:

The **general** solution to inhomogeneous equation

$$a F(n) + b F(n-1) + c F(n-2) = g(n) \quad \text{for } n > 1 \qquad (3.7.2.8)$$

can be obtained **as the sum** of the general solution of the homogeneous equation

$$a F(n) + b F(n-1) + c F(n-2) = 0$$

and a **particular** solution of (3.8.2.8).

The method of finding a particular solution of (3.8.2.8) and then a general solution of (3.8.2.8) is explained through the following examples.

Example 3.7.2.4

Let us consider the inhomogeneous recurrence

$$F(n) - 4F(n-1) + 4F(n-2) = 3$$

If F(n) = c is a particular solution of the recurrence, then replacing F(n), F(n−1) and F(n−2) by c in the recurrence given above, we get

$$c - 4c + 4c = 3$$

i.e., $c = 3$

Also, the general solution of the characteric equation (*of the inhomogeneous recurrence given above*) viz

$$F(n) - 4F(n-1) + 4F(n-2) = 0$$

Are (*obtained from Example 2.8.2.2, as*) $\alpha 2^n \beta n 2^n$

Hence general solution of the given recurrence is given by

$$F(n) = \alpha 2^n + \beta n 2^n + 3$$

Where α and β are arbitrary real constants.

3.8 AVERAGE-CASE AND AMORTIZED ANALYSIS

In view of the inadequacy of the best-case analysis and worst-case analysis for all types of problems, let us study two more analyses of algorithms, viz, average-case analysis and amortized analysis.

3.8.1 Average-Case Analysis

In Section 3.4, we mentioned that efficiency of an algorithm may not be the same for all inputs, even for the same problem-size n. In this context, we discussed *Best-Case analysis* and *Worst-Case analysis* of an algorithm.

However, these two analyses may not give any idea about the behaviour of an algorithm on a *typical or random* input. In this respect, *average-case analysis* is more informative than its two just-mentioned counter-parts, particularly, when the algorithm is to be used frequently and on varied types of inputs. In order to get really useful information from the average-case analysis of an algorithm, we must explicitly mention the properties of the set of inputs, obtained either through empirical evidence or on the basis of theoretical analysis. We explain the ideas through the analysis of the following algorithm, that, for given an element K and an array A[1..n], returns the index i in the array A[1..n], if A[i] = K otherwise returns 0.

```
Algorithm Sequential_ Search ( A [ 1.. n ], K)
  begin
       i ← 1
       while (i < n    and  A [ i ] ≠ K)    do
                i ←i + 1
       If   i < n        return i
            else    return   0
   end;
```

Some of the assumptions, that may be made, in the case of the above algorithm, for the purpose of average-case analysis, are

(i) for some number p, $0 \le p \le 1$, p is the probability of successful search and

(ii) in the case of successful search, the probability that first time K occurs in ith position in A[1..n], is the same for all i= 1, 2, …n.

With these two assumptions, we make *average-case analysis* of the Algorithm Sequential Search given above as follows:

From (i) above, Probability of unsuccessful search = (1– p).
In view of assumption (ii) above, in the case of successful search,
Probability of K occurring for the first time in the ith position in A[1..n] = p/n
for i= 1,2,..,n.

Therefore, if C_{avg} (n), the average complexity for an input array of n elements, is given by

$$\mathbf{C_{avg}(n) = [1.(p/n) + 2.(p/n) + \ldots + i.(p/n) + \ldots + n.(p/n)] + n(1-p),}$$

where the term i . (*p/n*) is the contribution of i comparisons that have been made when executing while-loop i times such that i is the least index with A[i] = K after which while-loop terminates.

Also, the last term (*n* . (*1* − *p*)) is the contribution in which while-loop is executed n times and after which it is found that A [i] ≠ K for i = 1, 2, …., n.

Simplifying R.H.S of the above equation, we get

$$C_{avg}(n) = \left(\frac{p}{n}\right)\ [\,1 + 2 + \ldots + i + .. + n] + n\,(\,1 - p)$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n. \quad (1-p)$$

$$= \frac{p\,(n+1)}{2} + n\ \ (1-p)$$

As can be seen from the above discussion, the average-case analysis is more difficult than the best-case and worst-case analyses.
Through the above *example* we have obtained an idea of how we may proceed to find average-case complexity of an algorithm. *Next we outline the process of finding average-case complexity of any algorithm, as follows*:

(i) First categories all possible input instances into classes in such a way that inputs in the same class require or are expected to require the execution of the same number of the basic operation(s) of the algorithm.

(ii) Next, the probability distribution of the inputs for different class as, is obtained empirically or assumed on some theoretical grounds.

(iii) Using the process as discussed in the case of Sequential-Search above, we compute the average-case complexity of the algorithm.

It is worth mentioning explicitly that average-case complexity need not be the average of the worst-case complexity and best-case complexity of an algorithm, though in some cases, the two may coincide.
Further, the effort required for computing the average-case, is worth in view of its contribution in the sense, that in some cases, the average-case complexity is much better than the Worst-Case Complexity. For example, in the case of Quicksort algorithm, which we study later, for sorting an array of elements, the *Worst-Case complexity* is a quadratic function whereas the *average-case complexity* is bounded by some constant multiple of n log n. For large values of n, *a constant multiple of (n log n)* is much smaller than *a quadratic function of n*. Thus, without average-case analysis, we may miss many on-the-average good algorithms.

3.8.2 Amortized Analysis

In the previous sections, we observed that

(i) *worst-case* and *best-case* analyses of an algorithm may not give a good idea about the behaviour of the algorithm on a *typical or random* input.

(ii) validity of the conclusions derived from *average-case* analysis depends on the quality of the assumptions about probability distribution of the inputs of a given size.

Another important fact that needs our attention is the fact that most of the operations, including the most time-consuming operations, on a data structure (used for solving a problem) do not occur in isolation, but different operations, with different time complexities, occur as a part of a *sequence* of operations. *Occurrences of a particular operation in a sequence of operations are dependent on the occurrences of*

other operations in the sequence. As a consequence, it may happen that the most time consuming operation can occur but only rarely or the operation only rarely consumes its theoretically determined maximum time. *We will support our claim later through an example*. But, we continue with our argument in support of the need for another type of analysis, viz., ***amortized analysis*****,** for better evaluation of the behaviour of an algorithm. However, this fact of *dependence* of both the occurrences and complexity of an *operation* on the occurrences of other operations, is not taken into consideration in the earlier mentioned analyses. As a consequence, the complexity of an algorithm is generally over-evaluated.

Next, we give an example in support of our claim that the frequencies of occurrences of operations and their complexities, are generally interdependent and hence the impact of the most time-consuming operation may not be as bad as it is assumed or appears to be.

Example 3.8.2.1:

We define a new data structure say *MSTACK*, which like the data structure STACK has the usual operations of PUSH and POP. In addition there is an operation MPOP(S, k), where S is a given stack and k is a non-negative integer. Then MPOP(S, k) removes top k elements of the stack S, if S has at least k elements in the stack. Otherwise it removes all the elements of S. *MPOP (S, k) may be formally defined as*

Procedure MPOP (S, k);

```
begin
While (not Empty (S) and k ≠ 0) do
   POP (S)
   k ← k−1
Endwhile;
EndMPOP.
```

For example, If, at some stage the Stack S has the elements

```
35    40    27    18    6    11
↑                            ↑
TOP                        BOTTOM
```

Then after MPOP (S, 4) we have

```
6     11
↑     ↑
TOP   BOTTOM
```

Further another application of MPOP (S, 3) gives empty stack.

Continuing with our example of MSTACK, next, we make the following assumptions and observations:

(i) Cost of each PUSH and POP is assumed to be 1 and if $m \geq 0$ is the number of elements in the Stack S when an MPOP (S, k) is issued, then

$$\text{Cost (MPOP(S, k))} = \begin{cases} k & \text{if } k \leq m \\ m & \text{otherwise} \end{cases}$$

(ii) If we start with an empty stack S, then at any stage, the number of elements that can be POPed off the stack either through a POP or MPOP, can not exceed the total number of preceding PUSHes.

The above statement can be further strengthened as follows:

(ii a) If we start with an empty stack S, then, **at any stage**, the number of elements that can be popped off the stack **through all** the POPs and MPOPs can not exceed the number of all the earlier PUSHes.

For Example, if S_i denotes *ith* PUSH and M_j denote *jth* POP/MPOP and if we have a sequence of PUSH/POP/MPOP as (say)

$$S_1\ S_2\ S_3\ M_1\ S_4\ S_5\ M_2\ S_6\ S_7\ M_3\ S_8\ S_9\ S_{10}\ S_{11}\ M_4$$

Then in view of (i) above

$$\text{Cost}(M_1) \leq \text{Cost}(S_1\ S_2\ S_3) = 3$$
$$\text{Cost}(M_1) + \text{Cost}\ M_2 \leq \text{Cost}(S_1\ S_2\ S_3) + \text{Cost}(S_4\ S_5) = 5$$
$$\text{Cost}(M_1) + \text{Cost}(M_2) + \text{Cost}(M_3) \leq \text{Cost}(S_1\ S_2\ S_3) + \text{Cost}(S_4\ S_5) + \text{Cost}(S_6\ S_7) = 7$$

In general if we have a sequence of PUSH/POP/MPOP, **total n in number**, then for a sequence.

$$S_{11}\ S_{12} \ldots S_{1i_1}\ M_1\ S_{21}\ S_{22} \ldots S_{2i_2}\ M_2 \ldots S_{t1}\ S_{t2} \ldots St\,i_t\ M_t$$

Where Mj is either a POP or MPOP and (3.8.2.1)

$$(i_1 + 1) + (i_2 + 1) + \ldots\ldots + (i_t + 1) = n. \quad (3.8.2.2)$$
$$\Rightarrow i_1 + i_2 + \ldots\ldots + i_t \leq n \quad (3.8.2.3)$$
i.e, cost of all PUSHes $\leq n$. (3.8.2.4)
$$\text{Cost}(M_1) \leq \text{Cost}(S_{11}\ S_{12} \ldots\ldots\ S_{1i_1}) = i_1$$

$$\text{Cost}(M_2) \leq \text{Cost}(S_{21}\ S_{22} \ldots S_{2i_2}) = i_2$$
$$\text{Cost}(M_t) \leq \text{Cost}(S_{t1}\ S_{t2} \ldots S_{ti_t}) = i_t$$

$\therefore \text{Cost}(M_1) + \text{Cost}(M_2) + \ldots\ldots + \text{Cost}(M_t) \leq$ sum of costs of all Pushes
$= i_1 + i_2 + \ldots + i_t \leq n$ (from (3.8.2.3))

Therefore total cost sequence of n PUSHES/POPs/MPOPs in (3.8.2.1) is $\leq n + n = 2n$

Thus, we conclude that

Total cost a sequence of n operations in MSTACK $\leq 2n$, (3.8.2.5)
whatever may be the frequency of MPOPs in the sequence.

However, if we go by the worst case analysis, then in a sequence of n operations

(i) all the operations may be assumed to be MPOPs

(ii) the worst-case cost of each MPOP may be assumed as (n−1), because, theoretically, it can be assumed for worst cas analysis purpose, that before each MPOP all the operations, which can be at most (n−1), were pushes.

Thus, the worst-case cost of a sequence of n operations of PUSHes, and MPOPs = $n\,.\,(n-1) = n^2 - n$, which is quadratic in n. (3.8.2. 6)

Thus, further, we conclude that **though**

(i) the *worst-case* cost in this case as given by (3.8.2.6) is **quadratic**, **yet**

(ii) because of interdependence of MPOPs on preceding PUSHes, the *actual cost* in this case, as given by (3.8.2.5) which is only **linear**.

Thus, we observe that operations are not considered in isolation but as a part of a sequence of operations, and, because of interactions among the operations, highly-costly operations may either not be attained or may be distributed over the less costly operations.

The above discussion motivates the concept and study of AMORTIZED ANALYSIS.

3.9 SUMMARY

In this unit, the emphasis is on the *analysis* of algorithms, though in the process, we have also *defined* a number of algorithms. Analysis of an algorithm generally leads to computational complexity/efficiency of the algorithm.

It is shown that general analysis of algorithms may not be satisfactory for all types of situations and problems, specially, in view of the fact that the same algorithm may have vastly differing complexities for different instances, though, of the same size. This leads to the discussion of worst-case and best-case analyses in Section 3.4 and of average-case analysis and amortized analysis in Section 3.8.

In, Section 3.2, we discuss two simple examples of algorithms for solving the same problem, to illustrate some simple aspects of design and analysis of algorithms. Specially, it is shown here, how a minor modification in an algorithm may lead to major efficiency gain.

In Section 3.3, the following sorting algorithms are defined and illustrated with suitable examples:

(i) Insertion Sort

(ii) Bubble Sort

(iii) Selection Sort

(iv) Shell Sort

(v) Heap Sort

(vi) Merge Sort

(vii) Quick Sort

Though these algorithms are not analyzed here, yet a summary of the complexities of these algorithms is also included in this section.

Next, the process of analyzing an algorithm and computing complexity of an algorithm in terms of basic instructions and basic constructs, is discussed in Section 3.5 and 3.6.

The Section 3.5 deals with the analysis in terms of non-recursive control structures in and the Section 3.6 deals with the analysis in terms of recursive control structures.

3.10 SOLUTIONS/ANSWERS

Ex. 1) List to be sorted: 15, 10, 13, 9, 12, 17 by Insertion Sort.

Let the given sequence of numbers be stored in A[1..6] and let m be a variable used as temporary storage for exchange purposes.

Iteration (i): For placing A[2] at its correct relative position w.r.t A[1] in the finally sorted array, we need the following operations:

(i) As A[2] = 10 < 15 = A [1], therefore, we need following additional operations
(ii) 10= A[2] is copied in m, s.t A[1] = 15, A[2] = 10, m = 10
(iii) 15 = A [1] is copied in A [2] s.t A[1] = 15, A[2] = 15, m = 10
(iv) 10 = m is copied in A[1], so that A[1] = 10, A[2] = 15.

Thus, we made one comparison and 3 assignments in this iterations
Iteration (ii): *At this stage A [1] = 15, A [2] = 10 and A [3] = 13*

Also, for correct place for A [3] = 13 w.r.t A[1] and A[2], the following operations are performed

(i) 13 = A[3] *is compared* with A[2] = 15

As A[3] < A[2], therefore, the algorithm further performs the following operations

(ii) 13 = A[3] copied to m so that m = 13

(iii) A [2] is copied in A [3] s.t. A[3] = 15 = A[2]

(iv) Then 13 = m is *compared with*

A [1] = 10 which is less than m.

Therefore A[2] is the correct location for 13

(v) 13 = m is copied in A[2] *s.t., at this stage*

A[1] = 10, A[2] = 13, A[3] = 15

And m = 13

In this iteration, the algorithm made 2 comparisons and 3 assignments

Iteration III: For correct place for A[4] = 9 w.r.t A[1], A[2] and A[3], the following operations are performed:

(i) A[4] = 9 *is compared with* A[3] = 15.
As A[4] = 9 < 15 = A[3], hence the algorithm.
(ii) 9 = A[4] is copied to m so that m = 9
(iii) 15 = A [3] is copied to A [4] s.t A[4] = 15 = A[3]
(iv) m is *compared with* A[2] = 13, as m < A [2], therefore, further the following operations are performed.
(v) 13 = A[2] is copied to A[3] s.t A[3] = 13
(vi) m = 9 is *compared with A[1]* = 10, as performs the following additional operations.
(vii) 10 = A[1] is copied to A[2]
(viii) finally 9 = m is copied to A[1]

In this iteration 3 comparisons and 5 assignments were performed

So that at this stage we have

A[1] = 9, A[2] = 10, A[3] = 13, A[4] = 15

Iteration IV: For correct place for A[5] = 12, the following operations are performed.

(i) 12 = A [5] is compared with A[4] = 15

In view of the earlier discussion, and in view of the fact that the number 12 (contents of A[5]) occurs between A[2] = 10 and A[3] = 13, the algorithm need to perform, in all, the following operations.

(a) Copy 12 = A[5] to m so that m becomes 12 **(one assignment)**
(b) *Three Comparisons* of A[5] = 12 are with A[4] = 15, A[3] = 13 and A[2] = 10. The algorithm stops comparisons on reaching a value less than value 12 of current cell
(c) The following THREE assignments: A[4] = 15 to A[5], A[3] = 13 to A[4] and m = 12 to A[3]

Thus in this iteration 4 assignments and 3 comparisons were made. And also, at this stage A[1] = 9, A[2] = 10, A[3] = 12, A[4] = 13, A[5] = 15 and m = 12.

Iteration V: The correct place for A[6] = 17, after sorting, w.r.t the elements of A[1..5] is A[6] itself. In order to determine that A[6] is the correct final position for 17, we perform the following operations.

(i) 17 = A[6] is copied to m **(one assignment)**
(ii) m is compared with A[5] = 15 and as A[5] = 15 < 17 = m, therefore, no more comparisons and no copying of elements A[1] to A[5] to the locations respectively on their right.
(iii) (*though this step appears to be redundant yet*) *the algorithm executes it*
17 = m is copied to A[6]

In this iteration 2 assignments and one comparison were performed.

To summerize:

In	I iteration,	1 comparison and	3 assignment were performed
In	II iteration,	2 comparison and	3 assignment were performed
In	III iteration,	3 comparison and	5 assignment were performed
In	IV iteration,	3 comparison and	4 assignment were performed
In	V iteration,	1 comparison and	3 assignment were performed

Thus, in all, 10 comparisons and 18 assignments were performed to sort the given array of 15,10,13,9,12 and 17.

Ex. 2) **List to be sorted: 15, 10, 13, 9, 12, 17 by Bubble Sort.**

A temporary variable m is used for exchanging values**. An exchange A [i]↔ A [j] takes 3 assignments viz**

m ← A [i]
A [i] ← A [j]
A [j] ← m

There are (6 – 1) = 5 iterations. In each iteration, whole list is scanned once, comparing pairs of neighbouring elements and exchanging the elements, if out of order. Next, we consider the different iterations.

In the following, the numbers in the bold are compared, and if required, exchanged.

Iteration I:

15	**10**	13	9	12	17
10	**15**	**13**	9	12	17
10	13	**15**	**9**	12	17
10	13	9	**15**	**12**	17
10	13	9	12	**15**	**17**
10	13	9	12	15	17

In this iteration, 5 comparisons and 5 exchanges i.e., 15 assignments, were performed

Iteration II: The last element 17 is dropped from further consideration

10	**13**	9	12	15
10	**13**	**9**	12	15
10	9	**13**	**12**	15
10	9	12	**13**	**15**
10	9	12	13	15

In this iteration 4 comparisons, 2 exchanges i.e., 6 assignments, were performed.

Iteration III: The last element 15 is dropped from further consideration

10	**9**	12	13
9	**10**	**12**	13
9	10	**12**	**13**
9	10	12	13

In this iteration 3 comparisons and 1 exchange i.e., 3 assignments were performed.

Iteration IV: The last element 13 is dropped from further consideration

9	**10**	12
9	**10**	**12**
9	10	12

In this iteration, 2 comparisons and 0 exchanges, i.e., 0 assignments were performed

Iteration V: The last element 12 is dropped from further consideration

9	**10**
9	**10**

In this iteration, 1 comparison and 0 exchange and hence 0 assignments were performed.

Thus, the Bubble sort algorithm performed (5+4+3+2+1) = 15 comparisons and 24 assignments in all.

Ex. 3) List to be sorted: 15, 10, 13, 9, 12, 17 by Selection Sort.

There will be five iterations in all. *In each of the five iterations at least, the following operations are performed:*

- 2 initial assignments to MAX and MAX-POS and

- 2 final assignments for exchanging values of A [MAX_POS] and A [n – i+1], the last element of the sublist under consideration.

Next we explain the various iterations and for each iteration, count the operations in addition to these 4 assignments.

Iteration 1: MAX ← 15
MAX_ POS ← 1
MAX is compared with successively 10, 13, 9, 12, and 17, one at a time i.e, 5 comparisons are performed.

Also MAX = 15 < 17 ∴ MAX ← 17 and MAX_ POS ← 6

Thus in addition to the 4 assignments mentioned above, 2 more assignments and 5 comparisons are performed:

Iteration 2: Now the list under consideration is

15, 10, 13, 9, 12

Again MAX ← 15, MAX_POS ← 1
Further MAX is compared successively with 10, 13, 9, 12 one at a time and as none is more than 15, therefore, no change in the value of MAX and MAX_POS. Ultimately 12 and 15 are exchanged to get the list 12, the last element of the sublist under consideration 10, 13, 9, 15.

Thus, this iteration performs 4 comparisons and no additional assignments besides the 4 mentioned earlier.

Iteration 3: The last element 15, is dropped from further consideration. The list to be consideration in this iteration is 12, 10, 13, 9

Initially MAX ← 12 and MAX_POS ← 1. MAX is compared with 10 and then with 13. As 13 > 12 = current MAX. Therefore, MAX ← 13, and MAX_POS ← 3 *(2 additional assignments).* Then MAX is compared with 9 and then 13 and 9 are exchanged we get the list: 12, 10, 9, 13 .

Thus in this iteration, 3 comparisons and six assignments are performed, in addition to the usual 4.

Iteration 4: The last element 13, is dropped from further consideration. The list to be sorted is 12, 10, 9
Again MAX ← 12, MAX_POS ← 1

The list after the iteration is 10, 9, 12. Again 2 comparisons of 12 with 10 and 9 are performed. No additional assignments are made, in addition to normal 4.

Iteration 5: The last element 12, is dropped from further consideration. The list to be sorted is 10, 9. MAX ← 10 and MAX_POS ← 1. One comparison of MAX = 10 with 9. No additional assignment over and above the normal 4.

Finally, the list to be sorted is: 9 and the process of sorting terminates.

The number of operations performed iteration-wise, is given below:

In Iteration I	:	5 comparisons and 6 assignments were performed
In Iteration II	:	4 comparisons and 4 assignments were performed
In Iteration III	:	3 comparisons and 6 assignments were performed
In Iteration IV	:	2 comparisons and 4 assignments were performed
In Iteration V	:	1 comparison and 4 assignments were performed

Hence total 15 comparisons and 24 assignments were performed to sort the list.

Ex.4) The array to be sorted is A [1 .. 6] = {15, 10, 13, 9, 12, 17}

(i) To begin with , increments in indexes to make sublists to be sorted, with value 3 from INC [1] and value 1 from INC [2], are read.

Thus two READ statements are executed for the purpose.

(ii) For selection of sublists A [1] = 15, A [2] = 9 **and** A [2] = 10, A [5] = 12 **and** A[3] = 13 A [6] = 17, the following operations are performed:

for INC [1] = 3
j ← INC [1] = 3
r ← [n/j] = [6/3] = 2

Two assignments are and one division performed. (A)

Further for each individual sublist, the following *type* of operations are performed.

t ← 1 (*one assignment*)

The comparison 6 = n < 2+3+1 is performed which returns true, hence

s ← r – 1 = 2 – 1 = 1 is performed (*one subtraction and one assignment*)

Next, to calculate the position of A [t+3 × s], one multiplication and one addition is performed. Thus, for selection of a particular sublist, 2 assignments, one comparison, one subtraction, one addition and one multiplication is performed

Thus, just for the selection of all the three sublists, the following operations are performed:

6 Assignments
3 Comparisons
3 Subtractions
3 additions
3 multiplications

(iii) (a) For sorting, the sublist

A [1] = 15, A [2] = 9 using Insertion sort,

first, the comparison 9 = A [2] < A [1] = 15 is performed which is true, hence

$i \leftarrow 1$
$m \leftarrow A[2] = 9$ **(two assignments performed)**

Next, the comparisons m = 9 < A [1] =15 and 1 = i < 0 are performed, both of which are true. **(two comparisons)**

15 = A [1] is copied to A [2]
and $i \leftarrow 1 - 1 = 0$ **(assignment)**

Next again the one comparisons is performed viz i > 0 which is false and hence 9 = m < A [0] is not performed

Then 9 = m is copied to A [1] **(one assignment)**

(iii) (b) For sorting the next sublist 10, 12

As A [2] = 12 < 10 = A [1] is not true hence no other operation is performed for this sublist

(iii) (c) For sorting the next sublist 13, 17

only one comparison of A[2] = 17< 13 = A [1], which is not true, is performed.

We can count all the operations mentioned above for the final answer.

Ex.5) To sort the list 15, 10, 13, 9, 12, 17 stored in A[1..6], **using Heap Sort** first build a heap for the list and then recursively delete the root and restore the heap.

Step I

(i) the five operations of assignments of values from 2 to 6 to j the outermost loop of variable, are made. Further, five assignments of the form: location $\leftarrow$ j are made one for each value of j.

Thus 10 assignments are made so for.

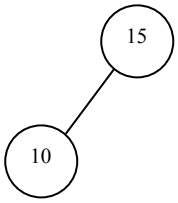
(ii) (a) Next, enter the while loop

For j = 2
2 = location > 1 is tested, which is true. **(one comparison)**

Hence
parent $\leftarrow \left(\frac{\text{location}}{2}\right) = 1,$ is performed. **(one comparison)**

A [location] = A [2] = 10 < 15 = A [1] = A [parent]
is tested which is true. Hence no exchanges of values. **(one comparison)**

The heap at this stage is



(ii) (b) For j = 3

3 = location >1 is tested **(one comparison)**

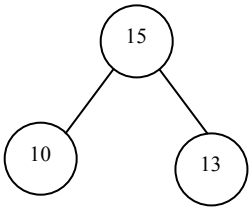
which is true. Therefore,

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = 1$ is performed **(assignments)**

A [location] = A [3] = 13 < 15 = A [1] = A [parent] is tested **(one comparison)**

As the last inequality is true, Hence no more operation in this case.

The heap at this stage is



(ii) (c) For j $\leftarrow$ 4

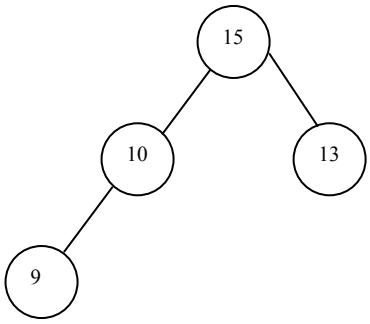
Location = 4>1 is tested **(one comparison)**

which is true. Therefore

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = 2$ is performed **(one assignment)**

A [location] = A [4] = 9 < 10 = A [2] is performed. **(one comparison)**

As the above inequality is true, no more operations in this case. The heap at this stage is



(ii) (d) For j $\leftarrow$ 5 is tested which

The Comparison

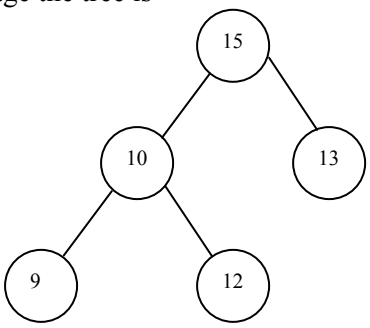
Location = 5 > 1 is performed, **(one comparison)**

which is true. Therefore,

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$

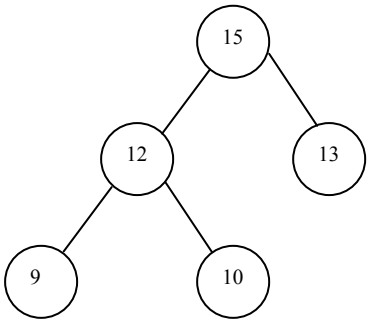
is performed **(one comparison)**

At this stage the tree is



A [location] = A [5] = 12 < A [Parent] = A [2] = 10 is
Performed. **(one comparison)**

which is not true. Hence the following additional operations are performed:
A [2] and A [5] are exchanged **(3 assignments)** so that the tree becomes



Also, the operations
location $\leftarrow$ parent = 2 and parent $\leftarrow$ (location/2) = 1 are performed
(two assignment)

Further 2 = location >1 is tested, **(one comparison)**

which is true. Therefore,
A[location] = A [2] $\leq$ A [1] = A [parent] is tested **(one comparison)**

which is true. Hence no more operations.

(ii) (e) j $\leftarrow$ 6

The comparison
Location = 6>1 is performed **(one comparison)**
Therefore, parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = 3$ is performed **(one assignment)**
A [location] = A [6] = 17 < 9 = A[3] is performed **(one comparison)**

which is not true. Hence, the following additional operations are performed

A [3] and A[6] are exchanged **(3 assignment)**

Next
location $\leftarrow$ 3 **(one assignment)**
(location>1) is performed **(one comparison)**

and

parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = 1$ **(one assignment)**

is performed
Further

A [location] = 17 < 15 = A [parent] is performed, **(one comparison)**

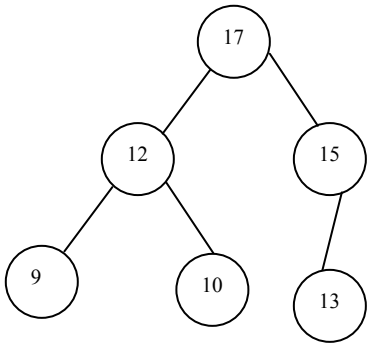
which is false.

Hence, the following operations are further performed

A [1] and A [3] are exchanged **(3 assignments)**

And A [location] ← A [parent] = 1 is performed **(one assignments)**
Also
(1 = location >1) is performed **(one comparison)**

which is not true. Hence the process is completed and we get the heap

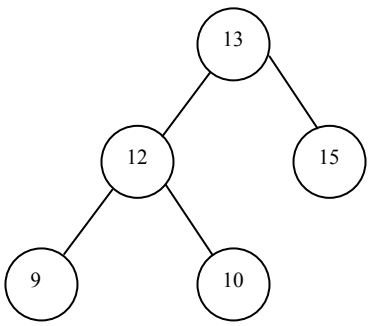


Step II: The following three substeps are iterated 1 repeated 5 times:

a. To delete the root
b. to move the value of the last node into the root and the last node is removed from further consideration
(iii) Convert the tree into a Heap.

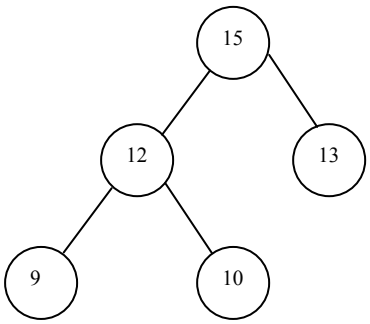
The sub steps (i) and (ii) are performed 5 times each, which contribute to 10 assignments

Iteration (i): after first two sub steps the *heap* becomes the *tree*



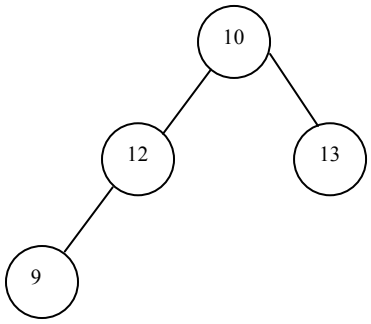
Root node is compared, one by one, with the values of its children **(2 comparisons)**
The variable MAX stores 15 and MAX_POS stores the index of the right child **(Two assignment)**

Then 15 is exchanged with 13 **(3 assignments)**
is performed, to get the heap



Thus in this iteration, 2 comparisons and 5 assignments were performed

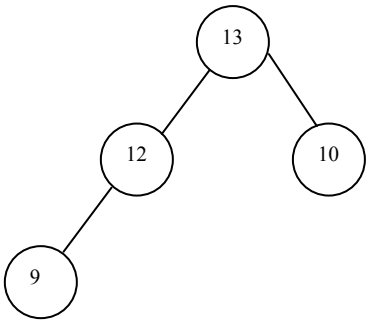
Iteration (ii): 15 of the root is removed and 10, the value of last node, is moved to the root to get the tree



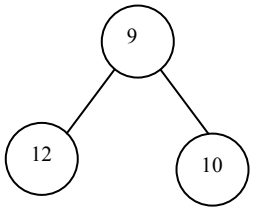
Again, the value 10 of the root is compared with the children
(2 comparison)

MAX first stores value 12 and then 13, and MAX_POS stores first the index of the left child and then index of the right child **(4 assignments)**

Then 13 and 10 are exchanged so that we get the Heap

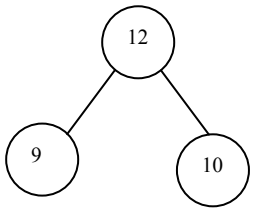


Iteration (iii): Again 13 of the root mode is removed and 9 of the last node is copied in the root to get the tree

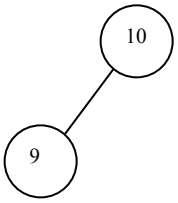


2 comparisons are made of value 9 with 12 and 10. Also **two assignments** for MAX ← 12 and MAX_POS ← index of the Left –child, are made. Also 12 and 9 are exchanged requiring **3 additional assignments.**

Thus in this iteration, 2 comparisons and 5 assignments are performed to get the heap



Iteration (iv): 12 is removed and the value 10 in the last node is copied in the root to get the tree



10 is compared with 9 **(only one comparison)**
and no assignments

In this iteration only one comparison is made

Iteration (v) (i): 10 is deleted from the root and 9 is copied in the root. As the root is the only node in the tree, the sorting process terminates.

Finally, by adding the various operations of all the iterations, we get the required numbers of operations.

Ex. 6) **List to be sorted: 15, 10, 13, 9, 12, 17 by Merge Sort.**

Chop the given list into two sublists

((15, 10, 13) (9, 12, 17))
Further chopping the sublists we get

((15) (10, 13)), ((9) (12, 17))
Further chopping the sublists we get sublists each of one element

(((15), ((10), (13))), ((9), ((12), (17)))

merging after sorting, in reverse order of chopping, we get

((15), (10, 13)) ((9), (12, 17))

Again merging, we get

((10, 13, 15) (9,12, 17))

Again merging, we get

(9,10, 12, 13, 15, 17)
This completes the sorting of the given sequence.

Ex. 7)

The sequence
15, 10, 13, 9, 12, 17,

to be sorted is stored in A [1.. 6]
We take A [1] = 15 as pivot
i is assigned values 2, 3, 4 etc. to get the first value from the left such that A[i] > pivot.

Similarly, j is moved backward from last index to get first j so that A[j] < pivot.

The index i = 6, is the first index s.t 17 = A[i] > pivot = 15.
Also j = 5 i.e. A [5] = 12, is the first value from the right such that A [j] < pivot.

As j < i, therefore
A[j] = A [5] = 12 is exchanged with pivot = 15 so that we get the array

12, 10, 13, 9, 15, 17

Next the two sublists viz 12, 10, 13, 9 and 17 separated by the pivot value 15, are sorted separately. However the relative positions of the sublists w.r.t 15 are maintained so we write the lists as

(12, 10, 13, 9), 15, (17).

The right hand sublist having only one element viz 17 is already sorted. So we sort only left-hand sublist but continue writing the whole list. Pivot for the left sublist is 12 and i = 3 and j = 4 are such that A[i] = 13 is the left most entry more than the pivot and A[j] =9 is the rightmost value, which is less than the pivot = 12. After exchange of A[i] and A[j], we get the list (12, 10, 9, 13), 15, (17). Again moving i to the right and j to the left, we get, i = 4 and j = 3. As j < i, therefore, the iteration is complete and A[j] = 9 and pivot = 12 are exchanged so that we get the list **((9, 10) 12 (13)) 15 (17)**Only remaining sublist to be sorted is (9, 10). Again pivot is 9, I = 2 and j = 1, so that A [i] is the left most value greater than the pivot and A[j] is the right most value less than or equal to pivot. As j < i, we should exchange A [j] = A[1] with pivot. But pivot also equals A [1]. Hence no exchange. Next, sublist left to sorted is {10} which being a single element is already sorted. The sublists were formed such that any element in a sublist on the left is less than any element of the sublist on the right, merging does not require.

3.11 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).

2. *ALGORITMICS: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).

3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).

4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).

5. *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni: (Galgotia Publications)

6. *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).

7. *Discrete Mathematics and Its Applications,* K.N. Rosen: (*Fifth Edition*) Tata McGraw-Hill (2003).

8. *Introduction to Alogrithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice – Hall of India (2002).

UNIT 1 DIVIDE-AND-CONQUER

Structure **Page Nos.**

1.0 INTRODUCTION

We have already mentioned that solving (a general) problem, with or without computers, is quite a complex and difficult task. We also mentioned a large number of problems, which we may encounter, even in a formal discipline like Mathematics, may not have any algorithmic/computer solutions. Out of the problems, which theoretically can be solved algorithmically, designing a solution for such a problem is, in general, quite difficult. In view of this difficulty, a number of standard techniques, which are found to be helpful in solving problems, have become popular in computer science. Out of these techniques Divide-and-Conquer is probably the most well-known one.

The general plan for Divide-and-Conquer technique has the following three major steps:

Step 1: An instance of the problem to be solved, is divided into a number of smaller instances of the (same) problem, generally of equal sizes. Any sub-instance may be further divided into its sub-instances. A stage reaches when either a direct solution of a sub-instance at some stage is available or it is not further sub-divisible. In the latter case, when no further sub-division is possible, we attempt a direct solution for the sub-instance.

Step 2: Such smaller instances are solved.

Step 3: Combine the solutions so obtained of the smaller instances to get the solution of the original instance of the problem.

In this unit, we will study the technique, by applying it in solving a number of problems.

1.1 OBJECTIVES

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

- explain the essential idea behind the Divide-and-Conquer strategy for solving problems with the help of a computer, and
- use Divide-and-Conquer strategy for solving problems.

1.2 GENERAL ISSUES IN DIVIDE-AND-CONQUER

Recalling from the introduction, Divide-and-Conquer is a technique of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved, split this into more than one sub-instances (*of the given problem*). If possible, divide each of the sub-instances into smaller instances, till a sub-instnace has a direct solution available or no further subdivision is possible. Then independently solve each of the sub-instances and then combine solutions of the sub-instances so as to yield a solution for the original instance.

The methods by which sub-instances are to be independently solved play an important role in the overall efficiency of the algorithm.

Example 1.2.1:

We have an algorithm, *alpha* say, which is known to solve all instances of size *n*, of a given problem, in at most c n^2 steps (where c is some constant). We then discover an algorithm, *beta* say, which solves the same problem by:

- Dividing an instance into 3 sub-instances of size n/2.
- Solve these 3 sub-instances.
- Combines the three sub-solutions taking *d n* steps in combining.

Suppose our original algorithm *alpha* is used to carry out the Step 2, viz., 'solve these sub-instances'. Let

T (alpha) (n) = Running time of alpha on an instance of size n.
T (beta) (n) = Running time of beta on an instance of size n.

Then,

T (alpha) (n) = c n^2 (by definition of alpha)

But

T (beta) (n) = 3 T (alpha) (n/2) + d n
= (3/4) (cn^2) + dn

So if dn < (cn^2)/4 (i.e., d/4c < n) then beta is faster than alpha.

In particular, for all large enough n's, (viz., for n > 4d/c = Constant), beta is faster than alpha.

The algorithm beta improves upon the algorithm *alpha* by just a constant factor. But if the problem size *n* is large enough such that for some i > 1, we have

n > 4d/c and also
n/2 > 4d/c and even
$n/2^i$ > 4d/c

which suggests that using *beta* instead of *alpha* for the Step 2 repeatedly until the sub-sub-sub…sub-instances are of size n_0 < = (4d/c), will yield a still faster algorithm.

So consider the following new algorithm for instances of size n

Procedure gamma (n : problem size),

If n < = n_0 then

```
            Solve problem using Algorithm alpha;
        else
            Split the problem instance into 3 sub-instances of size n/2;
            Use gamma to solve each sub-instance;
            Combine the 3 sub-solutions;
        end if ;
end  gamma;
```

Let *T* (gamma) (n) denote the running time of this algorithm. Then

$$T(\text{gamma})(n) = \begin{cases} cn^2 & \text{if } n <= n_0 \\ 3T(\text{gamma})\ (n/2) + dn, & \text{otherwise} \end{cases}$$

we shall show how relations of this form can be estimated. Later in the course, with these methods it can be shown that

$T(gamma)(n) = O(n^{(log3)}) = O(n^{1.59})$

This is a significant *improvement* upon algorithms *alpha* and *beta*, in view of the fact that as n becomes larger the differences in the values of $n^{1.59}$ and n^2 becomes larger and larger.

The improvement that results from applying algorithm *gamma* is due to the fact that it maximizes the savings achieved through *beta*. The (relatively) inefficient method *alpha* is applied only to "*small* " problem sizes.

The precise form of a divide-and-conquer algorithm is characterised by:

(i) The *threshold* input size, n_0, below which the problem size is not further sub-divided.

(ii) The *size* of sub-instances into which an instance is split.

(iii) The *number* of such sub-instances.

(iv) The method for solving instances of size $n <= n_0$.

(iv) The algorithm used to combine sub-solutions.

In (ii), it is more usual to consider the *ratio* of initial problem size to sub-instance size. In our example, the ration was 2. The *threshold* in (i) is sometimes called the (*recursive) base value*. In summary, the generic form of a divide-and-conquer algorithm is:

```
Procedure  D-and-C   (n : input size);
  begin
     read (n0)          ;        read the threshold value.
     if  n < = n0 then
       solve problem without further sub-division;
    else
       Split into sub-instances each of size n/k;
       for each of the r sub-instances do
              D-and-C (n/k);
       Combine the resulting  sub-solutions to produce the solution to the original
       problem;
end if;
end  D-and-C;
```

Such algorithms are naturally and easily realised as recursive procedures in (suitable) high-level programming languages.

1.3 INTEGER MULTIPLICATION

The following problem is a classic example of the application of Divide-and-Conquer technique in the field of Computer Science.

Input: Two n-digit decimal numbers $\underline{x}$ and $\underline{y}$ represented as

$\underline{x} = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ and

$\underline{y} = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

where x_i, y_i are decimal digits.

Output: The (*2n*)-digit decimal representation of the product x * y.

$\underline{z} = z_{2n-1}\ z_{2n-2}\ z_{2n-3} \ldots z_1\, z_0$

Note: *The algorithm given below works for any number base, e.g., binary, decimal, hexadecimal, etc. We use decimal simply for convenience.*

The classical algorithm for multiplication requires $O(n^2)$ steps to multiply two n-digit numbers.

A step is regarded as a single operation involving two single digit numbers, e.g., 5+6, 3 4, etc.*

In 1962, A. A. Karatsuba discovered an *asymptotically faster* algorithm for multiplying two numbers by using a divide-and-conquer approach.

The *values* x and y of the numbers with representations

$\underline{x} = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ and

$\underline{y} = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

are clearly given by,

$$x = \sum_{i=0}^{n-1} (x_i) * 10^i \ ; \text{ and}$$

$$y = \sum_{i=0}^{n-1} (y_i) \ * \ 10^i.$$

Then, the resultant value z = x * y

with representation

$\underline{z} = z_{2n-1}\ z_{2n-2}\ z_{2n-3} \ldots z_1\, z_0$

is given by

$$z = \sum_{i=0}^{2n-1} (z_i) * 10^i = \left(\sum_{i=0}^{n-1} x_i * 10^i \right) * \left(\sum_{i=0}^{n-1} y_i * 10^i \right).$$

For example:

581 = $5 * 10^2 + 8 * 10^1 + 1 * 10^0$
602 = $6 * 10^2 + 0 * 10^1 + 2 * 10^0$
581*602 = $349762 = 3 * 10^5 + 4 * 10^4 + 9 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 2 \times 10^0$

Let us denote

$\underline{a}$ = $x_{n-1} x_{n-2} \ldots x_{[n/2]+1} x_{[n/2]}$

$\underline{b}$ = $x_{[n/2]-1} x_{[n/2]-2} \ldots x_1 . x_0$

$\underline{c}$ = $y_{n-1} y_{n-2} \ldots y_{[n/2]+1} y_{[n/2]}$

$\underline{d}$ = $y_{[n/2]-1} y_{[n/2]-2} \ldots y_1 y_0$

Where [n/2] = largest integer less than or equal to n/2.

Then, if a, b, c, and d are the numbers whose decimal representations are $\underline{a}$, $\underline{b}$, $\underline{c}$ and $\underline{d}$ then

$x = a * 10^{[n/2]} + b \quad ; \quad y = c * 10^{[n/2]} + d$

For example, if n = 4, x = 1026 and y = 7329 then a = 10, b = 26, c = 73 and d = 29, and,

x = 1026 = $10 * 10^2 + 26$ = $a * 10^2 + b$
y = 7329 = $73 * 10^2 + 29$ = $c * 10^2 + d$

From this we also know that the result of multiplying x and y (i.e., z) is

$z = x * y = (a * 10^{[n/2]} + b) * (c * 10^{[n/2]} + d)$
$= (a * c) * 10^{2[n/2]} + (a * d + b * c) * 10^{[n/2]} + (b * d)$

where

$$2[n/2] = \begin{cases} n, & \text{if } n \text{ is even} \\ n+1 & \text{if } n \text{ is odd} \end{cases}$$

e.g,, 1026 * 7329 is

$= (10 * 73) * 10^4 + (10 * 29 + 26 * 73) * 10^2 + (26 * 29)$

$= 730 * 10^4 + 2188 * 10^2 + 754 = 7{,}519{,}554$

Each of the terms ($a * c$), ($a * d$), ($b * c$) and ($b * d$) is a product of **two [n/2]-digit numbers.**

Thus the expression for the multiplication of x and y in terms of the numbers a, b, c and d tells us that:

1. Two single digit numbers can be multiplied immediately. (**Recursive base: step 1**)
2. If n > *1* then the product of two n-digit numbers can be expressed in terms of *4 products of two numbers*[*] (**Divide-and-conquer stage**)

[*] For a given n-digit number, whenever we divides the sequence of digits into two subsequences, one of which has [n/] digits, the other subsequence has n – [n/2] digits, which is (n/2) digits if n is even and $\left(\frac{n+1}{2} \right)$ if n is odd. However, because of the convenience we may call both as (n/2) – digit sequences/numbers.

3. Given the four returned products, the calculation of the result of multiplying x and y involves only *additions* (can be done in O(n) steps) and multiplications by a *power of 10* (also can be done in O(n) steps, since it only requires placing the appropriate number of 0s at the end of the number). (**Combine stage**).

Steps 1–3, therefore, describe a Divide-and-Conquer algorithm for multiplying two n-digit numbers represented in decimal. However, Karatsuba discovered how the product of two n-digit numbers could be expressed in terms of **three** products each of two (n/2)-digit numbers, instead of the **four** products that a conventional implementation of the Divide-and-Conquer schema, as above, uses.

This saving is accomplished at the expense of a slightly more number of steps taken in the 'combine stage' (Step 3) (although, this will still uses O(n) operations).

We continue with the earlier notations in which z is the product of two numbers x and y having respectively the decimal representations

$\underline{x} = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$

$\underline{y} = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

Further a,b,c,d are the numbers whose decimal representations are given by

$\underline{a} = x_{n-1}\, x_{n-2} \ldots x_{[n/2]+1}\, x_{[n/2]}$

$\underline{b} = x_{[n/2]-1}\, x_{[n/2]-2} \ldots x_1 . x_0$

$\underline{c} = y_{n-1}\, y_{n-2} \ldots y_{[n/2]+1}\, y_{[n/2]}$

$\underline{d} = y_{[n/2]-1}\, y_{[n/2]-2} \ldots y_1\, y_0$

Let us compute the following 3 products (of two (n/2)-digit numbers):

U = a * c
V = b * d
W = (a + b) * (c + d)

Then

W = a * c + a * d + b * c + b* d
= U + a * d + b * c + V

Therefore,

a * d + b * c = W – U– V.

Therefore,

z = x * y
= $(a * 10^{[n/2]} + b) * (c * 10^{[n/2]} + d)$
= $(a * c) * 10^{2[n/2]} + (a * d + b * c) * 10^{[n/2]} + b * d$
= $U * 10^{2[n/2]} + (W–U–V) * 10^{[n/2]} + V$.

This algorithm is formalised through the following function.

function Karatsuba (xunder, yunder : **n-digit integer;** n : integer)
a, b, c, d: (n/2)–digit integer
U, V, W: n–digit integer;

begin
if n = 1 then
return x_0 * y_0;

else

a := $x_{(n-1)} \ldots x_{[n/2]}$;
b := $x_{[n/2]-1} \ldots x_0$;
c := $y_{(n-1)} \ldots y_{[n/2]}$;
d := $y_{[n/2]-1} \ldots y_0$;

U := Karatsuba (a, c, [n/2]);
V := Karatsuba (b, d, [n/2]);
W := Karatsuba (a+b, c+d, [n/2]);

Return U*10^ 2[n/2] + (W—U—V) * 10^ [n/2] + V;
; where 10^m stands for 10 raised to power m.
end if;
end Karatsuba;

Performance Analysis

One of the reasons why we study analysis of algorithms, is that **if** there are more than one algorithms that solve a given problem **then,** through analysis, we can find the running times of various available algorithms. And then we may choose the one be which takes least/lesser running time.

This is useful in allowing comparisons between the performances of two algorithms to be made.

For Divide-and-Conquer algorithms the running time is mainly affected by 3 criteria:

- The **number of sub-instances** (*let us call the number as* $\propto$) into which a problem is split.
- The **ratio of initial problem size to sub-problem size.** (*let us call the ration as* β*)*
- The **number of steps** required to **divide** the initial instance into substances and to **combine** sub-solutions, expressed as a function of the input size, n.

Suppose, *P*, is a divide-and-conquer algorithm that instantiates *alpha* sub-instances, each of size n/beta.

Let TP (n) denote the number of steps taken by P on instances of size n. Then

T (P (n_0)) = Constant (*Recursive-base*)
T (P (n)) = $\propto$ T (P (n | β) + gamma (n)

In the case when $\propto$ and β are both constant (as mentioned earlier, in all the examples we have given, there is a general method that can be used to solve such *recurrence relations* in order to obtain an asymptotic bound for the running time *Tp* (n). These methods were discussed in Block 1.

In general:

T (n) = $\propto$ T (n/β) + O (n ^ gamma)

$$T(n) = \begin{cases} O(n \text{ ^ gamma}) & \text{if alpha} < \text{beta ^gamma} \\ O(n \text{ ^ gamma log n}) & \text{if alpha} = \text{beta ^gamma} \\ O(n \text{ ^ log^-beta (alpha)}) & \text{if alpha} > \text{beta ^gamma} \end{cases}$$

In general:

T (n) = $\propto$ T (n/β) + O (n ^ gamma)

(where gamma is constant) has the solution

		O(n ^ gamma)	if	alpha	<	beta ^gamma
T(n)	=	O (n ^ gamma log n)	if	alpha	=	beta ^gamma
		O (n ^ log^-beta (alpha))if	if	alpha	>	beta ^gamma

Ex. 1) Using Karatsuba's Method, find the value of the product 1026732 × 732912

1.4 BINARY SEARCH

Binary Search algorithm searches a given value or element in an *already sorted* array by repeatedly dividing the search interval in half. It first compares the value to be searched with the item in the middle of the array. If there is a match, then search is successful and we can return the required result immediately. But if the value does not match with the item in middle of the array, then it finds whether the given value is less than or greater than the value in the middle of array. If the given value is less than the middle value, then the value of the item sought must lie in the lower half of the array. However, if the given value is greater than the item sought, must lie in the upper half of the array. So we repeat the procedure on the lower or upper half of the array according as the given value is respectively less than or greater than the middle value. The following C++ function gives the Binary Search Algorithm.

int Binary Seach (int * A, int low, int high, int value)

```
{       int  mid:
        while (low < high)
        {           mid = (low + high) / 2;
                     if (value = = A [mid])
                              return mid;
                    else if (value < A [mid])
                              high = mid – 1;
                    else low = mid + 1;
        }
        return – 1;

 }
```

Explanation of the Binary Search Algorithm

It takes as parameter the array A, in which the value is to be searched. It also takes the lower and upper bounds of the array as parameters viz., *low* and *high* respectively. At each step of the interation of the while loop, the algorithm reduces the number of elements of the array to be searched by half. If the value is found then its index is returned. However, if the value is not found by the algorithm, then the loop terminates if the value of the low exceeds the value of high, there will be no more items to be searched and hence the function returns a negative value to indicate that item is not found.

Analysis

As mentioned earlier, each step of the algorithm divides the block of items being searched in half. The presence or absence of an item in an array of n elements, can be established in at most lg n steps.

Thus the running time of a binary search is proportional to **lg *n*** and we say this is a **O(lg n)** algorithm.

Ex. 2) Explain how Binary Search method finds or fails to find in the given sorted array:

8 12 75 26 35 48 57 78 86 93 97 108 135 168 201

the following values

(i) 15

(ii) 93

(iii) 43

1.5 SORTING

We have already discussed the two sorting algorithms viz., Merge Sort and Quick Sort. The purpose of repeating the algorithm is mainly to discuss, not the design but, the analysis part.

1.5.1 Merge Sort

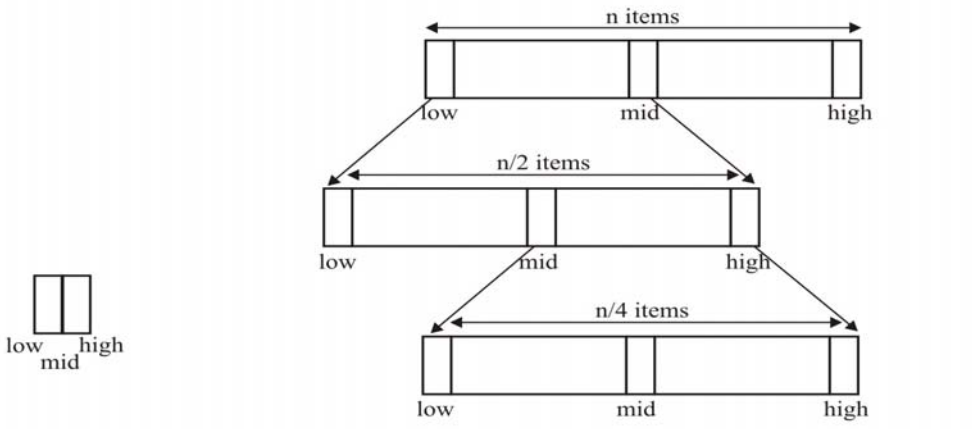
As discussed in Block 1, Merge Sort is a sorting algorithm which is based on the divide-and-conquer technique or paradigm. The Merge-Sort can be described in general terms as consisting of 3 major steps namely, a divide step, recursive step and merge step. These steps are described below in more detail.

Divide Step: If given array A has zero or 1 element then return the array A, as it is trivially sorted. Otherwise, chop the given array A in almost the middle to give two subarrays A_1 and A_2, each of which containing about half of the elements in A.

Recursive Step: Recursively sort arry A1 and A2.

Merge Step: Though recursive application of we reach a stage when subarrays of sizes 2 and then of sizes 1 are obtained. At this stage two sublists of sizes 1 each are combined by placing the elements of the lists in sorted order. The process of this type of combinations of sublists is repeated on lists of larger and large sizes. To accomplish this step we will define a C++ function void merge (int A []. int p, int r).

The recursion stops when the subarray has just only 1 element, so that it is trivially sorted. Below is the Merge Sort function in C++.



```
void merge_sort (int A[], int p, int r)
    {

        if (p < r)          //Check for base case
        {
                int q = (p + r)/2;          //Divide step
                merge_sort (A, p,q);              //Conquer step
                merge_sort (A, q + 1, r);//Conquer step
                merge (A, p, q, r);     }         //Combine step
```

Next, we define merge function which is called by the program merge-sort At this stage, we have an Array *A* and indices *p,q,r* such that $p < q < r$. Subarray *A*[p .. q] is sorted and subarray *A* [q + 1 . . r] is sorted and by the restrictions on *p, q, r,* neither subarray is empty. We want that the two subarrays are merged into a single sorted subarray in *A*[*p .. r*]. We will implement it so that it takes O(n) time, where n = $r - p + 1$ = the number of elements being merged.

Let us consider two piles of cards. Each pile is sorted and placed face-up on a table with the smallest card on top of each pile. We will merge these into a single sorted pile, face-down on the table. A basic step will be to choose the smaller of the two top cards, remove it from its pile, thereby exposing a new top card and then placing the chosen card face-down onto the output pile. We will repeatedly perform these basic steps until one input becomes empty. Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile. Each basic step should take constant time, since we check just the two top cards and there are 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. We don't actually need to check whether a pile is empty before each basic step. Instead we will put on the bottom of each input pile a special ***sentinel*** card. It contains a special value that we use to simplify the code. We know in advance that there are exactly r – p + 1 non sentinel cards. We will stop once we have performed r – p + 1 basic step. Below is the function merge which runs in O(n) time.

```
Void merge (int A[]. int p. int q. int r)
    {

        int n 1 = q – p + 1
        int n 2 = r – q:
        int* L = new int[n1 + 1];
        int * R = new int [ n2 + 1];
        for (int i = 1; i < = n1; i ++)
                L [i] = A [ p + i – 1];

        for (int j = 1; j < = n2; j++)
                R[ j] = A [q + j];
        L[0] = INT_MIN;          //negative infinity
        R[0] = INT_MIN;          //negative infinity
        L[n1 + 1] = INT_MAX;          // positive infinity
        R[n2 + 1] = INT_MAX;// positive infinity

        i = j = 1;
for (int k = p;k <=r; k++)
‘
‘
        if (L[i] < = R [j])

                ‘
                ‘
                        A[k] = L[i];
```

```
i + = 1;
        ‘
        ‘
        else
        ‘
        ‘
                A[k] = R[j];
                j + = 1;
        ‘
        ‘               }
```

Analysing merge sort

For simplicity, we will assume that n is a power of 2. Each divide step yields two subproblems, both of size exactly n/2. The base case occurs when n = 1. When n ≥ 2, time for merge sort steps are given below:

Divide: Just compute q as the average of p and r. D(n) = O(1).

Conquer: Recursively solve 2 subproblems, each of size n/2. Therefore, total time is 2T (n/2).

Combine: MERGE on an n-element subarray takes O(n) time. Therefore, C(n) = O(n). Since D(n) = O(1) and C(n) = O(n), summed together they give a function that is linear in n: O(n). Recurrence for merge sort running time is

T(n) = O(1) if n = 1,
T(n) = 2T (n/2) + O(n) if n ≥ 2.

Solving the merge-sort recurrence: By the master theorem, this recurrence has the solution T(n) = O(n lg n). Compared to insertion sort ($O(n^2)$ worst-case time), merge sort is faster. Trading a factor of n for a factor of lg n is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

1.5.2 Quick Sort

As mentioned earlier, the purpose of discussing the Quick Sort Algorithm is to discuss its analysis

- In the previous section, we discussed how the divide-and-conquer technique can be used to design sorting **algorithm Merge-sort**, as summarized below:
    - Partition *n* elements array *A* into two subarrays of *n/2* elements each
    - Sort the two subarrays recursively
    - Merge the two subarrays

Running time: *T*(n) = 2T (n/2) + θ (*n* log *n*).

The Quick-Sort algorithm is obtained by exploring another possibility of dividing the elements such that there is no need of merging, that is

Partition *a* [1…*n*] into subarrays A′ = A [1..q] and A′′ = A[q + 1…n] such that all elements in A′′ are larger than all elements in A′.
Recursively sort A′ and A′′.

(*nothing to combine/merge. A is already sorted after sorting A′ and A′′)*

Pseudo code for QUICKSORT:

```
QUICKSORT (A, p, r)
If p < r THEN
        q = PARTITION (A, p, r)
        QUICKSORT (A, p, q − 1)
        QUICKSORT (A, q + 1, r)
end if
```

The algorithm PARTITION, which is called by QUICKSORT, is defined after a short while.

Then, in order to sort an array A of n elements, we call QUICKSORT with the three parameters A, 1 and n QUICKSORT (*A, 1, n*).

If q = n/2 and is θ(n) time, we again get the recurrence. If T(n) denotes the time taken by QUICKSORT in sorting an array of n elements.

$T(n) = 2T(n/2) + \theta(n)$. Then after solving the recurrence we get the running time as $\Rightarrow T(n) = \theta(n \log n)$

The problem is that it is hard to develop partition algorithm which always divides *A* in two halves.

```
PARTITION (A, p, r)
x = A [ r]
i = p − 1
FOR j = p TO  r − 1 DO
        IF A [J] ≤ r THEN
                i = i + 1
                Exchange A [ i] and A[j]
        end if

end Do
Exchange A[I + 1] and A [r]
RETURN i + 1
```

QUICKSORT correctness:

- Easy to show inductively, if PARTITION works correctly

Example:

|2	8	7	1	3	5	6	4	i = 0, j = 1
2 |	8	7	1	3	5	6	4	i = 1, j = 2
2 |	8 |	7	1	3	5	6	4	i = 1, j = 3
2 |	8	7 |	1	3	5	6	4	i = 1, j = 4
2	1 |	7	8 |	3	5	6	4	i = 2, j = 5
2	1	3 |	8	7 |	5	6	4	i = 3, j = 6
2	1	3 |	8	7	5 |	6	4	i = 3, j = 7
2	1	3 |	8	7	5	6	4	i = 3, j = 8
2	1	3	4 |	7 |	5	6	8	q = 4

Average running time

The natural question: what is the average case running time of QUICKSORT? Is it close to worst case ($\theta(n^2)$), or to the best case θ (*n lg n)* ? Average time

depends on the distribution of inputs for which we take the average.

- If we run QUICKSORT on a set of inputs that are already sorted, the average running time will be close to the worst-case.
- Similarly, if we run QUICKSORT on a set of inputs that give good splits, the average running time will be close to the best-case.
- If we run QUICKSORT on a set of inputs which are picked uniformly at random from the space of all possible input permutations, then the average case will also be close to the best-case. Why? Intuitively, if any input ordering is equally likely, then we expect at least as many good splits as bad splits, therefore on the average a bad split will be followed by a good split, and it gets "absorbed" in the good split.

So, under the assumption that all input permutations are equally likely, the average time of QUICKSORT is θ (*n lg n*) (intuitively). Is this assumption realistic?

- Not really. In many cases the input is almost sorted: think of rebuilding indexes in a database etc.

The question is: how can we make QUICKSORT have a good average time irrespective of the input distribution?

- Using randomization.

1.6 RANDOMIZATION QUICK SORT*

Next, we consider what we call *randomized algorithms*, that is, algorithms that make some random choices during their execution.

Running time of normal *deterministic* algorithm only depend on the input.

Running time of a randomized algorithm depends not only on input but also on the random choices made by the algorithm.

Running time of a randomized algorithm is not fixed for a given input!

Randomized algorithms have best-case and worst-case running times, but the inputs for which these are achieved are not known, they can be any of the inputs.
We are normally interested in analyzing the *expected* running time of a randomized algorithm, that is the expected (average) running time for all inputs of size *n*

$$T_c(\mathrm{n}) = E_{|x|=n} \left| T(X) \right|$$

Randomized Quicksort

We can enforce that all *n*! permutations are equally likely by randomly permuting the input before the algorithm.

- Most computers have pseudo-random number generator *random* (1, n) returning "random" number between 1 and *n*.
- Using pseudo-random number generator we can generate a random permutation (such that all n! permutations equally likely) in O(*n*) time:

Choose element in *A*[1] randomly among elements in *A*[1..n],
choose element in *A*[2] randomly among elements in *A* [2..n], choose element in *A*[3] randomly among elements in *A* [3..n] and so on.

* This section may be omitted after one reading.

- Alternatively we can modify PARTITION slightly and exchange last element in *A* with random element in *A* before partitioning.

```
RAND PARTITION  (A. p. r)
i = RANDOM (p, r)
Exchange A|r| and A [i ]
RETURN PARTITION (A, p, r)
```

```
RANDQUICKSORT (A, p, r)
IF p < r  THEN
        q = RANDPARTITION (A,p,r)
   RANDQUICKSORT (A, p, q − 1, r)

END IF
```

Expected Running Time of Randomized Quicksort

Let $T(n)$ be the running of RANDQUICKSORT for an input of size n.

- Running time of RANDQUICKSORT is the total running time spent in all PARTITION calls.
- PARTITION is called n times
  - The pivot element r is not included in any recursive calls.

One call of PARTITION takes $O(1)$ time plus time proportional to the number of iterations FOR-loop.

  - In each iteration of FOR-loop we compare an element with the pivot element.

If X is the number of comparisons $A[j] \le r$ performed in PARTITION over the entire execution of RANDQUICKSORT then the running time is
$O(n + X)$.

$E[T(n)] = E[O(n + X)] = n + E[X]$
To analyse the expected running time we need to compute $E[X]$
To compute X we use $z_1, z_2 \ldots z_n$ to denote the elements in A where z_i is the ith smallest element. We also use Z_{ij} to denote $\{z_i, z_{i+1} \ldots z_j\}$.

Each pair of elements z_i and z_j are compared at most once (when either of them is the pivot)

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \quad \text{where}$$

$$X_{ij} = \begin{cases} 1 & \text{If } z_i \text{ compared to } z_i \\ 0 & \text{If } z_i \text{ not compared to } z_i \end{cases}$$

$$E|X| = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\left[z_i \text{ compared to } z_j \right]$$

To compute *Pr* [z_i compared to z_j] it is useful to consider when two elements are *not* compared.

Example: Consider an input consisting of numbers 1 through *n*.

Assume first pivot it 7 $\Rightarrow$ first partition separates the numbers into sets

{1, 2, 3, 4, 5, 6} and {8, 9, 10}.

In partitioning. 7 is compared to all numbers. No number from the first set will ever be compared to a number from the second set.

In general once a pivot *r*, $z_i < r < z_j$, is chosen we know that z_i and z_j cannot later be compared.

On the other hand if z_i is chosen as pivot before any other element in Z_{ij} then it is compared to each element in Z_{ij}. Similar for z_j.

In example 7 and 9 are compared because 7 is first item from $Z_{7,9}$ to be chosen as pivot and 2 and 9 are not compared because the first pivot in $Z_{2,9}$ is 7.

Prior to an element in Z_{ij} being chosen as pivot the set Z_{ij} is together in the same partition $\Rightarrow$ any element in Z_{ij} is equally likely to be first element chosen as pivot $\Rightarrow$ the probability that z_i or z_j is chosen first in Z_{ij} is $\frac{1}{j-i+1}$

$$Pr\,[z_i \text{ compared to } z_j] = \frac{2}{j-i+1}$$

- We now have:

$$E|X| = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\left[z_i \text{ compared to } z_j \right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$

Since best case is 0(n lg n) $\Rightarrow E\,|X| = \theta\,(n \lg n)$ and therefore $E\,|T(n)| = \theta\,(n \lg n)$.

Next time we will see how to make quicksort run in worst-case O(n log n) time.

1.7 FINDING THE MEDIAN

The problem of finding the median of n elements is a particular case of the more general selection problem. The selection problem can be stated as finding the ith order statistic in a set of n numbers or in other words the ith smallest element in a set of n element. The minimum is thus the 1st order statistic, maximum is the nth order

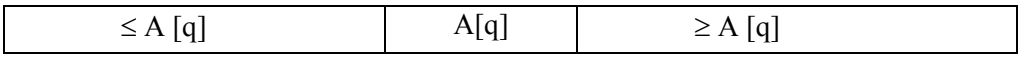
statistic and median is the $n/2^{th}$ order statistic. Also note that if n is even then there will be two medians.

We will give here two algorithms for the solution of above problem. One is practical randomized algorithm with O(n) expected running time. Another algorithm which is more of theoretical interest only, with O(n) worst case running time.

Randomized Selection

The key idea is to use the algorithm partition () from quicksort but we only need to examine one subarray and this saving also shows up in the running time O(n). We will use the Randomized Partition (A, p,r) which randomly partitions the Array A around an element A[q] such that all elements from A[p] to A[q−1] are less than A[q] and all elements A[q+1] to A[r] are greater than A[q]. This is shown in a diagram below.

q = Randomized Partition (A, p, r)



p q r

We can now give the pseudo code for Randomized Select (A, p, r, i). This procedure selects the i^{th} order statistic in the Array A [p ..r].

Randomized Select (A, p, r, i)

```
if  (p = = r) then return A [p];
q = Randomized Partition (A, p, r)
k = q – p + 1;
if (i ==k) then return A [q]  |q |;
if (i < k) then
        return Randomized Select (A, p, q-1, i);
else
        return Randomized Select (A, q+1, r, i-k);
```

Analyzing **Randomized Select ()**

Worst case: The partition will always occur in 0:n-1 fashion. Therefore, the time required by the Randomized Select can be described by a recurrence given below:

$T(n) = T(n-1) + O(n)$
$= O(n^2)$ (arithmetic series)

This running time is no better than sorting.
"Best" case: suppose a 9:1 partition occurs. Then the running time recurrence would be

$T(n) = T(9n/10) + O(n)$
$= O(n)$ (Master Theorem, case 3)

Average case: Let us now analyse the average case running time of Randomized Select.

For upper bound, assume *i*th element always occurs in the larger side of partition:

$$T(n) \leq \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

Let's show that T(n) = O(n) by substitution.

Assume T(n) ≤ cn for sufficiently large c:

T(n)	≤	$\frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$	*The recurrence we started with*
	≤	$\frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$	*Substitute T(n) ≤ cn for T(k)*
	=	$\frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$	*Split" the recurrence*
	=	$\frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$	*Expand arithmetic series*
	=	$c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$	*Multiply it out*
T(n)	≤	$c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$	*The recurrence so far*
	=	$cn - c - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$	*Subtract c/2*
	=	$cn - (\frac{cn}{4} - \frac{c}{2} + \Theta(n))$	*Rearrange the arithmetic*
	≤	cn (if c is big enough)	*What we set out to prove*

Worst-Case Linear Time Selection

Randomized algorithm works well in practice. But there exists an algorithm which has a worst-case O(n) time complexity for find the ^{ith} order statistic but which is only of theoretical significance. The basis idea of worst case linear time selection is to generate a good partitioning element. We will call this element x.

The algorithm in words:

1. Divide n elements into groups of 5
2. Find median of each group
3. Use Select () recursively to find median x of the [n/5] medians
4. Partition the n elements around x. Let *k* = rank (x)
5. **if** (i = = k) **then** return x
6. **if** (i < K) **then** use Select () recursively to find *i*th smallest element in first partition
7. **else** (i > k) use Select () recursively to find (*i*–*k*) th smallest element in last partition.

There are at least ½ of the 5-element medians which are ≤ x which equal to $\lfloor\lfloor n/5\rfloor/2\rfloor = \lfloor n/10\rfloor$ and also there are at least $3\lfloor n/10\rfloor$ elements which are ≤ x. Now, for large n, $3\lfloor n/10\rfloor \geq n/4$. So at least n/4 elements are ≤ x and similarly n/4 elements are ≥ *x*. Thus after partitioning around *x*, step 5 will call Select () on at most 3*n*/4 elements. The recurrence is therefore.

$T(n)$	$\leq$	$T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n)$	
	$\leq$	$T(n/5) + T(3n/4) + \Theta(n)$	$\lfloor n/5 \rfloor \leq n/5$
	$\leq$	$cn/5 + 3cn/\Theta(n))$	*Substitute T(n) = cn*
	=	$19cn/20 + \Theta(n)$	*Combine fractions*
	=	$cn - (cn/20 - \Theta(n))$	*Express in desired form*
	$\leq$	cn if c is big enough	*What we set out to prove*

1.8 MATRIX MULTIPLICATION

In number of problems, matrix multiplications form core of algorithms to solve the problems under consideration. Any saving in time complexity may lead to significant improvement in the overall algorithm. The conventional algorithm makes $O(n^3)$ integer multiplications. Next, we discuss Strassen's Algorithm which makes only $O(n^{2.8})$ integer multiplications for multiplying 2 $n \times n$ matrices.

Strassen's Algorithm

Strassen's recursive algorithm for multiplying $n \times n$ matrices runs in $\theta(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large value of n, therefore, it outperforms the $\theta(n^3)$ matrix-multiplication algorithm.

The idea behind the Strassen's algorithm is to multiply 2×2 matrices with only 7 scalar multiplications (instead of 8). Consider the matrices

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \left(\begin{array}{c|c} e & g \\ f & h \end{array} \right)$$

The seven submatrix products used are

$P_1 = a \,.\, (g - h)$
$P_2 = (a + b) \,.\, h$
$P_3 = (c + d) \,.\, e$
$P_4 = d \,.\, (f - e)$
$P_5 = (a + d) \,.\, (e + h)$
$P_6 = (b - d) \,.\, (f + h)$
$P_7 = (a - c) \,.\, (e + g)$

Using these submatrix products the matrix products are obtained by

$r = P_5 + P_4 - P_2 + P_6$
$s = P_1 + P_2$
$t = P_3 + P_4$
$u = P_5 + P_1 - P_3 - P_7$

This method works as it can be easily seen that $s = (ag - ah) + (ah + bh) = ag + bh$. In this method there are 7 multiplications and 18 additions. For $(n \times n)$ matrices, it can be worth replacing one multiplication by 18 additions, since multiplication costs are much more than addition costs.

The recursive algorithm for multiplying $n \times n$ matrices is given below:

1. Partition the two matrices A, B into $n/2 \times n/2$ matrices.
2. Conquer: Perform 7 multiplications recursively.
3. Combine: Form using + and −.

The running time of above recurrence is given by the recurrence given below:

$T(n) = 7T(n/2) + \theta(n^2)$
$= \theta(n^{\lg 7})$
$= O(n^{2.81})$

The current best upper bound for multiplying matrices is approximately $O(n^{2.376})$

Limitations of Strassen's Algorithm

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for the following four reasons:

1. The constant factor hidden in the running time of Strassen's algorithm is larger than the constant factor in the naïve $\theta(n^3)$ method.
2. When the matrices are sparse, methods tailored for sparse matrices are faster.
3. Strassen's algorithm is not quite as numerically stable as the naïve method.
4. The sub matrices formed at the levels of recursion consume space.

Ex. 3) Multiply the following two matrices using Strassen's algorithm

$$\begin{bmatrix} 5 & 6 \\ -4 & 3 \end{bmatrix} \text{ and } \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

1.9 EXPONENTIATION

Exponentiating by Squaring is an algorithm used for the fast computation of large powers of number *x*. It is also known as the **square-and-multiply** algorithm or **binary exponentiation**. It implicitly uses the binary expansion of the exponent. It is of quite general use, for example, in modular-arithmetic.

Squaring Algorithm

The following recursive algorithm computes x^n, for a positive integer n:

$$\text{Power}(x, n) = \begin{cases} x, & \text{if } n = 1 \\ \text{Power}(x^2, n/2), & \text{if } n \text{ is even} \\ x \cdot (\text{Power}(x^2, (n-1)/2)), & \text{if } n > 2 \text{ is odd} \end{cases}$$

Compared to the ordinary method of multiplying x with itself $n - 1$ times, this algorithm uses only $O(\log n)$ multiplications and therefore speeds up the computation of x^n tremendously.

Further Applications

The same idea allows fast computation of large exponents modulo a number. Especially in cryptography, it is useful to compute powers in a ring of integers modulo q. It can also be used to compute integer powers in a group, using the rule

$$\text{Power}(x, -n) = (\text{Power}(x, n))^{-1}.$$

The method works in every semigroup and is often used to compute powers of matrices.

Examples 1.9.1:

$$13789^{722341} \pmod{2345}$$

would take a very long time and lots of storage space if the native method is used: compute 13789^{72234} then take the remainder when divided by 2345. Even using a more effective method will take a long time: square 13789, take the remainder when divided by 2345, multiply the result by 13789, and so on. This will take 722340 modular multiplications. The square-and-multiply algorithm is based on the observation that $13789^{722341} = 13789\,(13789^2)^{361170}$. So if we computed 13789^2, then the full computation would only take 361170 modular multiplications. This is a gain

of a factor of two. But since the new problem is of the same type, we can apply the same observation *again*, once more approximately having the size.

The repeated application this algorithm is equivalent to decomposing the exponent (by a base conversion to binary) into a sequence of squares and products: for example,

$x^7 = x^4 x^2 x^1$
$= (x^2)^2 * x^2 * x$
$= (x^2 * x)^2 * x$ *algorithm needs only 4 multiplications instead of* $7 - 1 = 6$
where $7 = (111)_2 = 2^2 + 2^1 + 2^0$

Some more examples:

$x^{10} = ((x^2)^2 * x)^2$ because $10 = (.1010)_2 = 2^3 + 2^1$, algorithm needs 4 multiplications instead of 9
$x^{100} = (((((x^2 * x)^2)^2)^2 * x)^2)^2$ because $100 = (1\ 100\ 100)2 = 2^6 + 2^5 + 2^2$, algorithm needs 8 multiplications instead of 99
$x^{1,000} = (((((((x^2 * x)^2 * x)^2 * x)^2)^2 * x)^2)^2)^2$ because $10^3 = (1\ 111\ 101\ 000)_2$, algorithm needs 14 multiplications instead of 999
$x^{1,000,.000} = ((((((((((((((((x^2*x)^2*x)^2)^2*x)^2)^2)^2)^2)^2*x)^2)^2)^2)^2)^2)^2$ because $10^6 = (\ 11\ \ 110\ 100\ \ 001\ 001\ 000\ \ 000)_2$, algorithm needs 25 multiplications
$x^{1.000.000.000} = ((((((((((((((((((((((((((x^2 * x)^2 * x)^2)^2 * x)^2\ )^2)^2)^2 *x)^2)^2 * x)^2)^2 * x)^2* x)^2)^2)^2\ x)^2x)^2)^2*x)^2)^2)^2)^2)^2)^2)^2)^2$ because $10^9 = (111\ \ 0011\ \ 100\ 101\ 101\ 100\ 101\ \ 000\ 000\ 000)_2$, algorithm needs 41 multiplications

Addition Chain

In mathematics, an **addition chain** is a sequence $a_0, a_1, a_2, a_3\ldots$ that satisfies

$a_0 = 1$, and
for each $k > 0$:
$a_k = a_i + a_j$ for some $i < j < k$

For example: 1, 2, 3, 6, 12, 24, 30, 31 is an addition chain for 31, of length 7, since

$2 = 1 + 1$
$3 = 2 + 1$
$6 = 3 + 3$
$12 = 6 + 6$
$24 = 12 + 12$
$30 = 24 + 6$
$31 = 30 + 1$

Addition chains can be used for exponentiation: thus, for example, we only need 7 multiplications to calculate 5^{31}:

$5^2 = 5^1 \times 5^1$
$5^2 = 5^2 \times 5^1$
$5^6 = 5^3 \times 5^3$
$5^{12} = 5^6 \times 5^6$
$5^{24} = 5^{12} \times 5^{12}$
$5^{30} = 5^{24} \times 5^6$
$5^{31} = 5^{30} \times 5^1$

Addition chain exponentiation

In mathematics, **addition chain exponentiation** is a fast method of exponentiation. It works by creating a minimal-length addition chain that generates the desired exponent. Each exponentiation in the chain can be evaluated by multiplying two of the earlier exponentiation results.

This algorithm works better than binary exponentiation for high exponents. However, it trades off space for speed, so it may not be good on over-worked systems.

1.10 SUMMARY

The unit discusses various issues in respect of the technique viz., *Divide and Conquer* for designing and analysing algorithms for solving problems. First, the general plan of the *Divide and conquer* technique is explained and then an outline of a formal *Divide-and-conquer* procedure is defined. The issue of whether at some stage to solve a problem directly or whether to further subdivide it, is discussed in terms of the relative efficiencies in the two alternative cases.

The technique is illustrated with examples of its applications to solving problems of (large) integer multiplication, Binary search, Sorting, of finding median of a given data, of matrix multiplication and computing exponents of a given number. Under sorting, the well-known techniques viz., Merge-sort and quick-sort are discussed in detail.

1.11 SOLUTIONS/ANSWERS

Ex.1) 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

$$\begin{aligned} x \times y &= (1026 \times 0732)\ 10^{2\times 3} + 732 \times 912 \\ &\quad + [(1026 + 732) \times (732 + 912) \\ &\quad - (1026 \times 0732) - (732 \times 912)]10^3 \\ &= (1026 \times 0732)\ 10^6 + 732 \times 912 + \\ &\quad [(1758 \times 1644) - (1026 \times 0732) - (732 \times 912)]10^3 \end{aligned}$$

… (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.

$U = 1026 \times 732$
$V = 732 \times 912$
$P = 1758 \times 1644$

Let us consider only the product 1026 × 732 and other involved products may be computed similarly and substituted in (A).

Let us write

$$\begin{aligned} U &= 1026 \times 732 = (10 \times 10^2 + 26)\ (07 \times 10^2 + 32) \\ &= (10 \times 7)\ 10^4 + 26 \times 32 + [(10 + 7)\ (26 + 32) \\ &\quad - 10 \times 7 - 26 \times 32)]\ 10^2 \end{aligned}$$

$$= 17 \times 10^4 + 26 \times 32 + (17 \times 58 - 70 - 26 \times 32)\, 10^2$$

At this stage, we do not apply Karatsuba's algorithm and compute the products of 2-digit numbers by conventional method.

Ex. 2) The number of elements in the given list is 15. Let us store these in an array say A[1..15]. Thus, initially low = 1 and high = 15 and, hence, mid = (1+15)/2 = 8.

In the first iteration, the search algorithm compares the value to be searched with A[8] = 78

Part (i): The value to be searched = 15
As 15 < A[8] = 78, the algorithm iterates itself once more. In the second iteration, the new values become

low = 1 high = mid − 1 = 7

and hence (new) mid = $\left\lfloor \frac{1+7}{2} \right\rfloor = 4$

As 15 < A[4] = 26, the algorithm iterates once more. In the third iteration, the new values become

low = 1, high 4 − 1 = 3

Therefore, mid = $\left(\frac{1+3}{2} \right) = 2$

As 15 > A[2] = 12, the algorithm iterates itself once more. In fourth iteration, new values become

low = mid + 1 = 2+ 1 = 3, high = 4

Therefore

(new) mid = $\left\lfloor \frac{3+4}{2} \right\rfloor = 3$

As A[3] = 15 (*the value to be searched*)
Hence, the algorithm terminates and returns the index value 3 as output.

Part (ii): The value to be searched = 93

As the first iteration is common for all values to be searched, therefore, in the first iteration

low = 1, high = 15 and mid = 8
As 93 > A[8] = 78,

therefore, the algorithm iterates once more. In the second iteration, new values are

low = mid + 1 = 9, high = 15

and (new) mid = $\frac{9+15}{2} = 12$, where A[12] = 108

As 93 < A [12] = 108,

Therefore, the algorithm iterates once more. For the third

iteration

low = 9 ; high = mid − 1 = 12 − 1 = 11

and (new) mid = $\frac{9+11}{2}$ = 10 with A[10] = 93

As A[10] = 93, therefore, the algorithm terminates and returns the index value 10 of the given array as output.

Part (iii): The value to be searched is 43. As explained earlier, in the first iteration

low = 1, high = 15 and mid = 8

As 43 < A[8] = 78, therefore, as in part (i)

low = 1, high = 8 − 1 = 7 and mid = 4

As 43 > A[4] = 26, the algorithm makes another iteration in which

low = mid + 1 = 5 high = 7 and (new) mid = (5 + 7)/2 = 6

Next, as 43 < A[6] = 48, the algorithm makes another iteration, in which

low = 5 high = 6 − 1 = 5
hence mid = 5, and A[5] = 35
As 43 > A[5], hence value ≠ A[5].

But, at this stage, low is not less than high and hence the algorithm returns − 1, indicating failure to find the given value in the array.

Ex. 3) Let us denote

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

and

$$\begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

Then

$P_1 = a \,.\, (g - h)$
$= 5\,(6 - 9) = -15$
$P_2 = (a + b) \,.\, h = (5 + 6) \,.\, 9 = 99,$
$P_3 = (c + d) \,.\, e = (-4 + 3) \,.\, (-7) = 7$
$P_4 = d \,.\, (f - e) = 3.\, (5 - (-7)) = 36;$
$P_5 = (a + d)\,(e + h) = (5 + 3)\,(-7 + 9) = 16$
$P_6 = (\,b - d)\,(f + h) = (6 - 3) \,.\, (5 + 9) = 42$
$P_7 = (a - c)\,(e + g) = (5 - (-4))\,(-7 + 6) = -9$

Then, the product matrix is

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

where

$r = P_5 + P_4 - P_2 + P_6$
$= 16 + 36 - 99 + 42 = -5$
$s = P_1 + P_2 = -15 + 99 = 84$
$t = P_3 + P_4 = 7 + 36 = 43$
$u = P_5 + P_1 - P_3 - P_7$
$= 16 + (-15) - 7 - (-9)$
$= 16 - 15 - 7 + 9$
$= 3$

1.12 FURTHER READINGS

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

UNIT 2 GRAPH ALGORITHMS

Structure **Page Nos.**

2.0 INTRODUCTION

A number of problems and games like chess, tic-tac-toe etc. can be formulated and solved with the help of graphical notations. The wide variety of problems that can be solved by using graphs range from searching particular information to finding a good or bad move in a game. In this Unit, we discuss a number of problem-solving techniques based on graphic notations, including the ones involving searches of graphs and application of these techniques in solving game and sorting problems.

2.1 OBJECTIVES

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

- explain and apply various graph search techniques, viz Depth-First Search (DFS), Breadth-First-Search (BFS), Best-First Search, and Minimax Principle;
- discuss relative merits and demerits of these search techniques, and
- apply graph-based problem-solving techniques to solve sorting problems and to games.

2.2 EXAMPLES

To begin with, we discuss the applicability of graphs to a popular game known as NIM.

2.2.1 NIM/Marienbad Game

The game of nim is very simple to play and has an interesting mathematical structure.

Nim is a game for 2 players, in which the players take turns alternately. Initially the players are given a position consisting of several piles, each pile having a finite number of tokens. On each turn, a player chooses one of the piles and then removes at least one token from that pile. The player who picks up the last token wins.

Assuming your opponent plays optimally, there may be some positions/situations in which the player having the current move cannot win. Such positions are called

"**losing** positions" (for the player whose turn it is to move next). The positions which are not losing ones are called **winning**.

Marienbad is a variant of a **nim** game and it is played with matches. The rules of this game are similar to the nim and are given below:

(1) It is a two-player game

(2) It starts with n matches (n must be greater or equal to 2 i.e., n >= 2)

(3) The winner of the game is one who takes the last match, whosoever is left with no sticks, loses the game.

(4) On the very first turn, up to n –1 matches can be taken by the player having the very first move.

(5) On the subsequent turns, one must remove at least one match and at most twice the number of matches picked up by the opponent in the last move.

Before going into detailed discussion through an example, let us explain what may be the possible states which may indicate different stages in the game. At any stage, the following two numbers are significant:

(i) The total number of match sticks available, after picking up by the players so far.
(ii) The number of match sticks that the player having the move can pick up.

We call the ordered pair a **state** in the game, where

i	:	the number of sticks available
j	:	the number of sticks that can be picked, by the player having the move, according to the rules.

For example:

(i) Initially if n is the number of sticks, then the state is (n, n−1), because the players must leave at least one stick.

(ii) While in the state (i, j), if the player having the move picks up k sticks then the state after this move, is (i − k, min (2k, i −k)),
which means

(a) the total number of available sticks is (i − k)

(b) the player, next to pick up, can not pick up more than the double of the number of sticks picked up in the previous move by the opponent and also clearly the player can not pick up more than the number of sticks available, i.e., (i −k)

(iii) We can not have the choice of picking up zero match sticks, unless no match stick is available for picking. Therefore the state (i, 0) implies the state (0,0),

After discussing some of possible states, we elaborate the game described above through the following example.

Example 2.2.1:

Let the initial number of matches be 6, and let player A take the chance to move first. What should be A's strategy to win, for his first move. Generally, A will consider all possible moves and choose the best one as follow:

- if A takes 5 matches, that leaves just one for B, then B will take it and win the game;
- if A takes 4 matches, that leaves 2 for B, then B will take it and win;
- if A takes 3 matches, that leaves 3 for B, then B will take it and win;

- if A takes 2 match, that leaves 4 for B, then B will take it and win;
- if A takes 1 match, that leaves 5 for B. In the next step, B can take 1 or 2(*recall that B* can *take at most the twice of the number what A just took*) and B will go to either of the states (4,2) or (3,3) both of which are winning moves for A, B's move will lead to because A can take all the available stick and hence there will be not more sticks for B to pick up. Taking a look at this reasoning process, it is for sure that the best move for A is just taking one match stick.

The above process can be expressed by a directed graph, where each node corresponds to a *position* (*state*) and each edge corresponds a *move* between two positions, with each node expressed by a pair of numbers $< i,j >$, $0 <= j <= i$, and
i: the number of the matches left;
j : the upper limit of number of matches which can be removed in the next move, that is, any number of matches between 1 and *j* can be taken in the next move.

As mentioned earlier, we have
The initial node: $< n,n-1 >$.
Edges leaving the node $< i,j >$ can lead to the node $< i-k, \min(2k, j-k) >$,
with $0 < k \leq i$.

In the directed graph shown below, *rectangular* nodes denote *losing* nodes and *oval* nodes denote *winning* nodes:

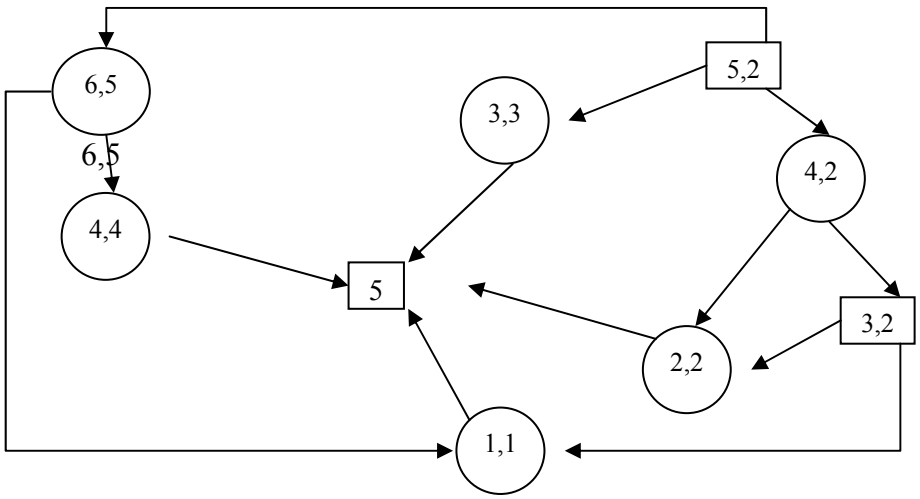


Figure 1

- a terminal node $< 0, 0 >$, from which there is no legal move. It is a *losing* position.
- a nonterminal node is a winning node (denoted by a circle), if **at least one** of its successors is a losing node, because the player currently having the move is can leave his opponent in losing position.
- a nonterminal node is a losing node (denoted by a square) if **all** of its successors are wining nodes. Again, because the player currently having the move cannot **avoid** leaving his opponent in one of these winning positions.

How to determine the wining nodes and losing nodes in a directed graph?
Intuitively, we can starting at the losing node $< 0, 0 >$, and work back according to the definition of the winning node and losing node. A node is a **losing node,** for the current player, if the move takes to a state such that the opponent can make at least one move which forces the current player to lose. On the other hand, a node is a **winning node,** if after making the move, current player will leave the opponent in a state, from which opponent can not win. For instance, in any of nodes $< 1, 1 >$,

< 2, 2 >, < 3, 3 > and < 4, 4 >, a player can make a move and leave his opponent to be in the position < 0, 0 >, thus these 4 nodes are wining nodes. From position < 3, 2 >, two moves are possible but both these moves take the opponent to a winning position so it is a losing node. The initial position < 6, 5 > has one move which takes the opponent to a losing position so it is a winning node. Keeping the process of going in the backward direction, we can mark the types of nodes in a graph. A recursive C program for the purpose, can be implemented as follows:

2.2.2 Function for Computing Winning Nodes

function *recwin*(*i, j*)
{ Return *true* if and only if node < *i, j* > is winning,
we assume 0 <= *j*< = *i* }
for *k* = 1 to *j* do
{ if not *recwin*(*i* - *k*, min(2*k*, *i* − *k*))
then return *true*
}
return *false*

Ex.1) Draw a directed graph for a game of Marienbad when the number of match sticks, initially, is 5.

2.3 TRAVERSING TREES

Traversing a tree means exploring all the nodes in the tree, starting wsith the root and exploring the nodes in some order. We are already aware that in the case of binary trees, three well-known tree-traversal techniques used are preorder, postorder and inorder. In **preorder** traversal, we first visit the node, then all the nodes in its left subtree and then all nodes in the right subtree. In **postorder** traversal, we first visit the left subtree, then all the nodes in the right subtree and the root is traversed in the last. In i**norder** traversal, the order of traversal is to first visit all the nodes in the leftsubtree, then to visit the node and then all the nodes in its right subtree. Postorder and preorder can be generalized to nonbinary trees. These three techniques explore the nodes in the tree from left to right.

Preconditioning

Consider a scenario in which problem might have many similar situations or instances, which are required to be solved. In such a situation, it it might be useful to spend some time and energy in calculating the auxiliary solutions (i.e., attaching some extra information to the problem space) that can be used afterwards to fasten the process of finding the solution of each of these situations. This is known as **preconditioning**. Although some time has to be spent in calculating / finding the auxiliary solutions yet it has been seen that in the final tradeoff, the benefit achieved in terms of speeding up of the process of finding the solution of the problem will be much more than the additional cost incurred in finding auxiliary/additional information.

In other words, let x be the time taken to solve the problem without preconditioning, y be the time taken to solve the problem with the help of some auxiliary results (i.e., after preconditioning) and let t be the time taken in preconditioning the problem space i.e., time taken in calculating the additional/auxiliary information. **Then to solve n typical instances, provided that y < x , preconditioning will be beneficial only when** ,

$$nx > t + ny$$
$$\text{i.e., } nx - ny > t$$
$$\text{or } n > t / (x - y)$$

Preconditioning is also useful when only a few instances of a problem need to be solved. Suppose we need a solution to a particular instance of a problem, and we need it in quick time. One way is to solve all the relevant instances in advance and store their solutions so that they can be provided quickly whenever needed. But this is a very inefficient and impractical approach,– i.e., to find solutions of all instances when solution of only one is needed. On the other hand, a popular alternative could be to calculate and attach some additional information to the problem space which will be useful to speedup the process of finding the solution of any given instance that is encountered.

For an **example**, let us consider the problem of finding the ancestor of any given node in a rooted tree (which may be a binary or a general tree).

In any rooted tree, node u will be an **ancestor** of node v, if node u lies on the path from root to v. Also we must note that every node is an ancestor of itself and root is an ancestor of all nodes in the tree including itself. Let us suppose, we are given a pair of nodes (u,v) and we are to find whether u is an ancestor or v or not. If the tree contains n nodes, then any given instance can take $\Omega(n)$ time in the worst case. But, if we attach some relevant information to each of the nodes of the tree, then after spending $\Omega(n)$ time in preconditioning, we can find the ancestor of any given node in constant time.

Now to precondition the tree, we first traverse the tree in preorder and calculate the precedence of each node in this order, similarly, we traverse the tree in postorder and calculate the precedence of each node. For a node u, let **precedepre[u]** be its precedence in preorder and let **precedepost[u]** be its precedence in postorder.

Let u and v be the two given nodes. Then according to the rules of preorder and postorder traversal, we can see that :

In ***preorder traversal,*** as the root is visited first before the left subtree and the right subtree, so,

If precedepre[u] <= precedepre[v], then
u is an ancestor of v or u is to the left of v in the tree.

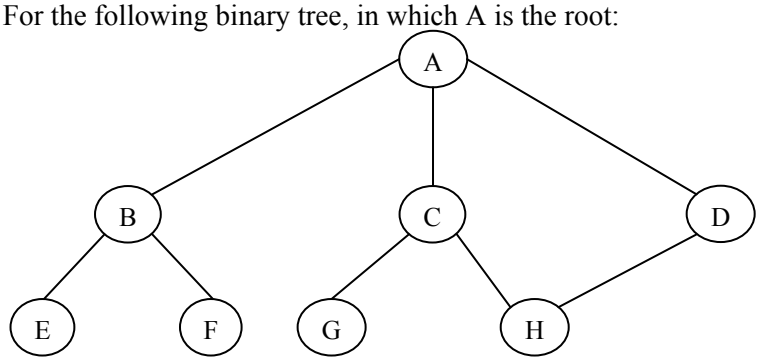
In ***postorder traversal,*** as the root is visited last, because, first we visit leftsubtree, then right subtree and in the last we visit root so,

If precedepost[u] >= precedepost[v], then
u is an ancestor of v or u is to the right of v in the tree.

So for u to be an ancestor of v, both the following conditions have to be satisfied:
precedepre[u] <= precede[v] and precedepost[u] >= precedepost[v].

Thus, we can see that after spending some time in calculating preorder and postorder precedence of each node in the tree, the ancestor of any node can be found in constant time.

Ex. 2) For the following binary tree, in which A is the root:



2.4 DEPTH-FIRST SEARCH

The depth-first search is a search strategy in which the examination of a given vertex u, is delayed when a new vertex say v is reached and examination of v is delayed when new vertex say w is reached and so on. When a leaf is reached (i.e., a node which does not have a successor node), the examination of the leaf is carried out. And then the immediate ancestor of the leaf is examined. The process of examination is carried out in reverse order of reaching the nodes.

In depth first search for any given vertex u, we find or explore or discover the first adjacent vertex v (in its adjacency list), not already discovered. Then, in stead of exploring other nodes adjacent to u, the search starts from vertex v which finds its first adjacent vertex not already known or discovered. The whole process is repeat for each newly discovered node. When a vertex adjacent to v is explored down to the leaf, we back track to explore the remaining adjacent vertices of v. So we search farther or deeper in the graph whenever possible. This process continues until we discover all the vertices reachable from the given source vertex. If still any undiscovered vertices remain then a next source is selected and the same search process is repeated. This whole process goes on until all the vertices of the graph are discovered.

The vertices have three adjacent different statuses during the process of traversal or searching, the status being: *unknown*, *discovered and visited*. Initially all the vertices have their status termed as '*unknown*', after being explored the status of the vertex is changed to '*discovered*' and after all vertices adjacent to a given vertex are discovered its status is termed as '*visited*'. This technique ensures that in the depth first forest, at a time each vertex belong to only one depth-first tree so these trees are disjoint.

Because we leave partially visited vertices and move ahead, to backtrack later, stack will be required as the underlying data structure to hold vertices. In the recursive version of the algorithm given below, the stack will be implemented implicitly, however, if we write a non-recursive version of the algorithm, the stack operation have to be specified explicitly.

In the algorithm, we assume that the graph is represented using adjacency list representation. To store the parent or predecessor of a vertex in the depth-first search, we use an array parent[]. Status of a 'vertex' i.e., unknown, discovered, or visited is stored in the array status. The variable *time* is taken as a global variable. V is the vertex set of the graph G.

In depth-first search algorithm, we also timestamp each vertex. So the vertex u has two times associated with it, the discovering time *d[u]* and the termination time *t[u]*. The *discovery time* corresponds to the status change of a vertex from unknown to discovered, and *termination time* corresponds to status change from discovered to visited. For the initial input graph when all vertices are unknown, time is initialized to 0. When we start from the source vertex, time is taken as 1 and with each new discovery or termination of a vertex, the time is incremented by 1. Although DFS algorithm can be written without time stamping the vertices, time stamping of vertices helps us in a better understanding of this algorithm. However, one drawback of time stamping is that the storage requirement increases.

Also in the algorithm we can see that for any given node u, its discovering time will be less than its termination time i.e., d[u] < t[u].

The algorithm is:

Program

DFS(G)
//This fragment of algorithm performs initializing
//and starts the depth first search process

```
1 for all vertices u ∈ V
2   {     status[u] = unknown;
3         parent[u] = NULL;

4          time = 0   }
5 for each vertex u ∈ V
6         {if status[u] == unknown
7                   VISIT(u)
```

```
VISIT(U)
       1 status[u] = discovered;
2 time = time + 1;
3 d[u] =time;}
4 for each Vertex v ∈  V adjacent to u
5         {if status[v] == unknown
6                   parent[v] = u;
7                   VISIT(V);

      8 time = time + 1;
      9 t[u] = time;
     10 status[u] = visited;}
```

In the procedure DFS, the first *for-loop* initializes the status of each vertex to unknown and parent or predecessor vertex to NULL. Then it creates a global variable *time* and initializes it to 0. In the second *for-loop* belonging to this procedure, for each node in the graph if that node is still unknown, the VISIT(u) procedure is called. Now we can see that every time the VISIT (u) procedure will be called, the vertex u it will become the root of a new tree in the forest of depth first search.

Whenever the procedure VISIT(u) will be called with parameter u, the vertex u will be unknown. So in the procedure VISIT(u), first the status of vertex u is changed to '*discovered*', time is incremented by 1 and it is stored as discovery time of vertex u in d[u].

When the VISIT procedure will be called for the first time, d[u] will be 1. In the *for-loop* for each given vertex u, every unknown vertex adjacent to u is visited recursively and the parent[] array is updated. When the for-loop concludes, i.e., when every vertex adjacent to u is discovered, the time is increment by 1 and is stored as the termination time of u i.e. t[u] and the status of vertex u is changed to '*visited*'.

Analysis of Depth-First Search

In procedure DFS(), each for loop takes time O($|V|$), where $|V|$ is the number of vertices in V. The procedure VISIT is called once for every vertex of the graph. In the procedure visit for each of the *for-loop* is executed equal to the number of edges emerging from that node and yet not traversed. Considering the adjacency list of all nodes to total number of edges traversed are O($|E|$), where $|E|$, is the number of edges in E. So the running time of DFS is, therefore, O ($|V|+|E|$).

Example 2.4.1:

For the graph given in Figure 2.4.1.1. Use DFS to visit various vertices. The vertex D is taken as the starting vertex and, if there are more than one vertices adjacent to a vertex, then the adjacent vertices are visited in lexicographic order.

In the following,

(i) the label i/ indicates that the corresponding vertex is the ith discovered vertex.

(ii) the label i/j indicates that the corresponding vertex is the ith discovered vertex and jth in the combined sequence of discovered and visited.

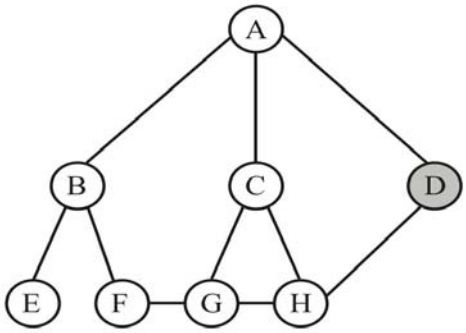


Figure 2.4.1.1: Status of D changes to discovered, d[D] = 1

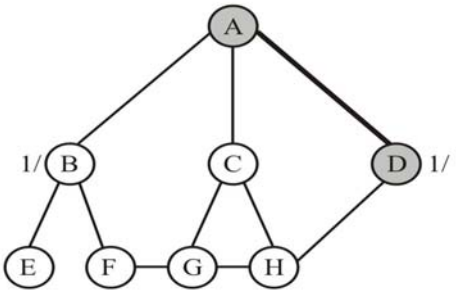


Figure 2.4.1.2: D has two neighbors by convention A is visited first i.e., the status of A changes to discovered, d[A] = 2

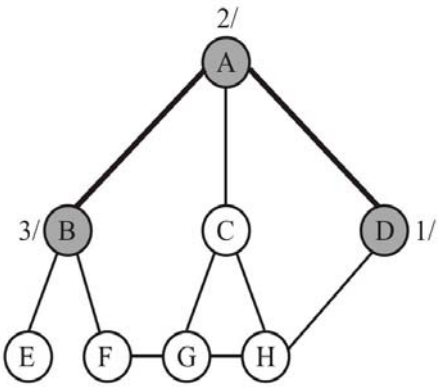


Figure 2.4.1.3: A has two unknown neighbors B and C, so status of B changes to 'discovered', i.e., d[B] = 3

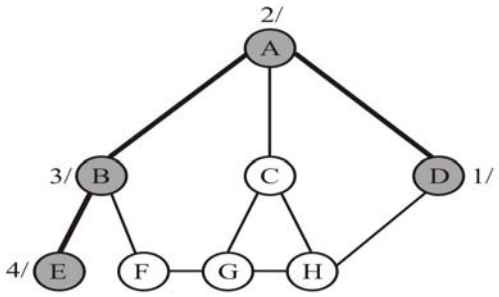


Figure 2.4.1.4: Similarly vertex E is discovered and d[E] = 4

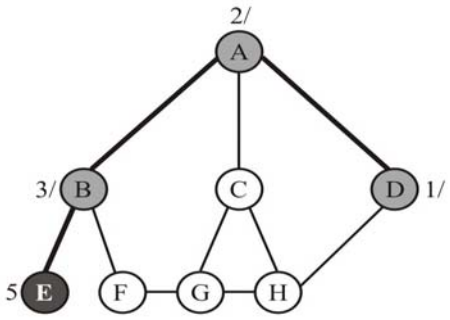


Figure 2.4.1.5: All of E's neighbors are discovered so status of vertex E is changed to 'visited' and t[E] = 5

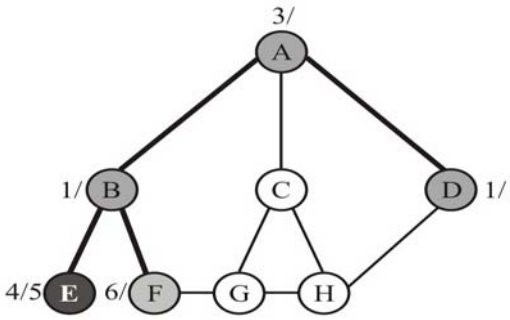


Figure 2.4.1.6: The nearest unknown neighbor ok B is F, so we change status of F to 'discovered', d[F] = 6

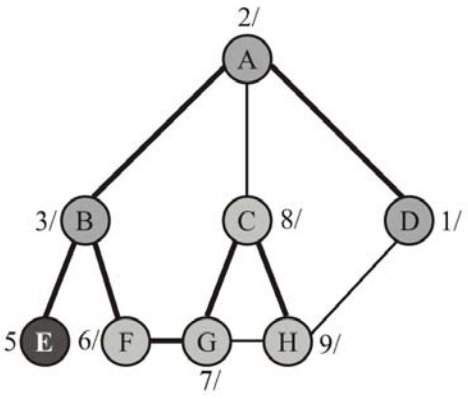


Figure 2.4.1.7: Similarly vertices G, E and H are discovered respectively with d[G] = 7, d[C] = 8 and d[H] = 9

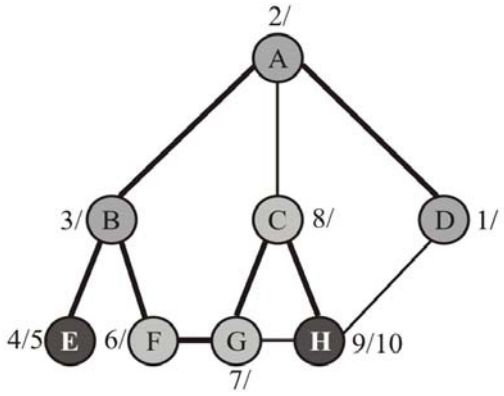


Figure 2.4.1.8: Now as all the neighbors of H are already discovered we backtrack, to C and stores its termination time as t[H] = 10

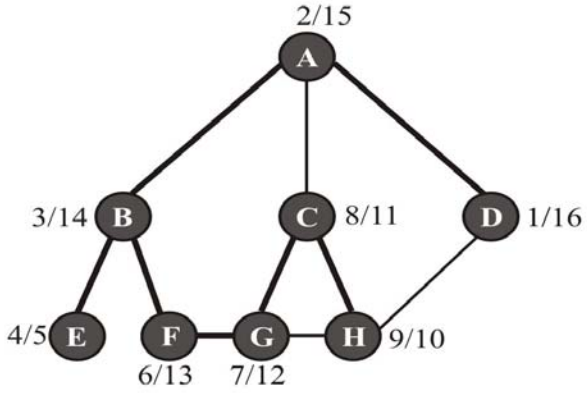


Figure 2.4.1.9: We find the termination time of remaining nodes in reverse order, backtracking along the original path ending with D.

The resultant parent pointer tree has its root at D, since this is the first node visited. Each new node visited becomes the child of the most recently visited node. Also we can see that while D is the first node to be '*discovered*', it is the last node *terminated.* This is due to recursion because each of D's neighbors must be discovered and terminated before D can be terminated. Also, all the edges of the graph, which are not used in the traversal, are between a node and its ancestor. This property of depth-first search differentiates it from breadth-first search tree.

Also we can see that the maximum termination time for any vertex is 16, which is twice the number of vertices in the graph because time is incremented only when a vertex is discovered or terminated and each vertex is discovered once and terminated once.

Properties of Depth-first search Algorithm

(1) **Parenthesis Structure**

In a graph G, if u and v are two vertices such that u is discovered before v then the following cases may occur:

(a) If v is discovered before u is terminated, then v will be finished before u (i.e., the vertex which is being discovered later will be terminated first). This property exists because of the recursion stack, as the vertex v which is discovered after u will be pushed on the stack at a time when u is already on the stack, so v will be popped out of the stack first i.e., v will be terminated first.

Also the interval [d[v],t[v]] is contained in the interval [d[u],t[u]], we Say the v is the proper descendant of u.

(b) If u is terminated before v is discovered then in this case t[u] < d[v] so the two intervals are disjoint.

In this case t[u] < t[v] so the two intervals are disjoint.

Note: We should remember that in depth-first search the third case of overlapping intervals is not possible i.e., situation given below is not, possible because of recursion.

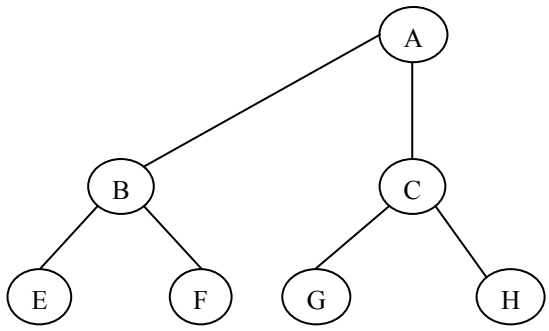
(2) Another important property of depth-first search (sometimes called white path property) is that v is a descendant of u if and only if at the time of discovery of u, there is at least one path from u to v contains only unknown vertices (i.e., white vertices or vertices not yet found or discovered).

(3) *Depth-First Search can be used to find connected components in a given graph:* One useful aspect of depth first search algorithm is that it traverses connected component one at a time and then it can be used to identify the connected components in a given graph.

(4) *Depth-first search can also be used to find cycles in an undirected graph:* we know that an undirected graph has a cycle if and only if at some particular point during the traversal, when u is already discovered, one of the neighbors v of u is also already discovered and is not parent or predecessor of u.

We can prove this property by the argument that if we discover v and find that u is already discovered but u is not parent of v then u must be an ancestor of v and since we traveled u to v via a different route, there is a cycle in the graph.

Ex.3) Trace how DFS traverses (i.e., discover and visits) the graph given below when starting node/vertex is B.



Depth First Search in *Directed* Graphs

The earlier discussion of the Depth-First search was with respect to undirected graphs. Next, we discuss Depth-First strategy with respect to Directed Graph. In a directed graph the relation of '*binary adjacent to*' is not symmetric, where the relation of '*being adjacent to*' is symmetric for undirected graphs.

To perform depth first search in *directed* graphs, the algorithm given above can be used with minor modifications. The main difference exists in the interpretation of an "adjacent vertex". In a directed graph vertex v is adjacent to vertex u if there is a directed edge from u to v. If a directed edge exists from u to v but not from v to u, then v is adjacent to u but u is not adjacent to v.

Because of this change, the algorithm behaves differently. Some of the previously given properties may no longer be necessarily applicable in this new situation.

Edge Classification

Another interesting property of depth first search is that search can be used to classify different type of edges of the directed graph G(V,E). This edge classification gives us some more information about the graph.

The different edges are:

(a) **Tree Edge:** An edge to a still 'unknown' vertex i.e., edge (u,v) is a *tree edge* if it is used to discover v for the first time.

(b) **Back edge:** An edge to an already 'discovered' or ancestor vertex i.e., edge (u,v) is a back edge if it connects to a vertex v which is already discovered which means v is an ancestor of u.

(c) **Forward edge:** An edge to an already 'visited' descendant (not possible in undirected graph) i.e., edge (u, v) is a forward edge if v is a descendant of u in the depth first tree. Also, we can see that d[u] < d[v].

(d) **Cross edge:** An edge to an already 'visited' neighbor, which is not a descendant. As long as one vertex is not descendant of the other, cross edge can go between vertices in the same depth first tree or between vertices in different depth first trees.

Note: In an undirected graph, every edge is either a tree edge or back edge, i.e., forward edges or cross edges are not possible.

Example 2.4.2:

In the following directed graph, we consider the adjacent nodes in the increasing alphabetic order and let starting vertex be.

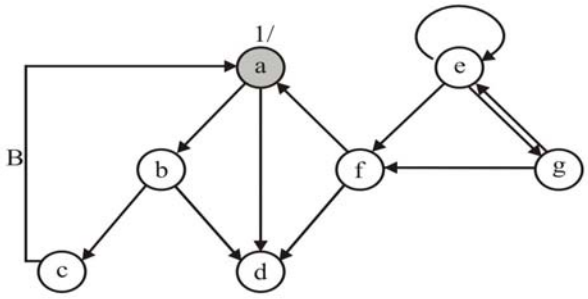


Figure 2.4.2.1: Status of a changed to discovered, d[a] = 1

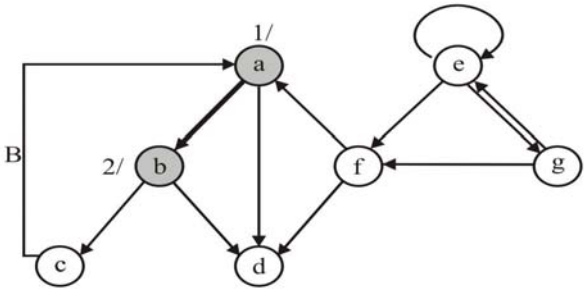


Figure 2.4.2.2: a has unknown two neighbors a and d, by convention b is visited first, i.e the status of b changes to discovered, d[a] =2

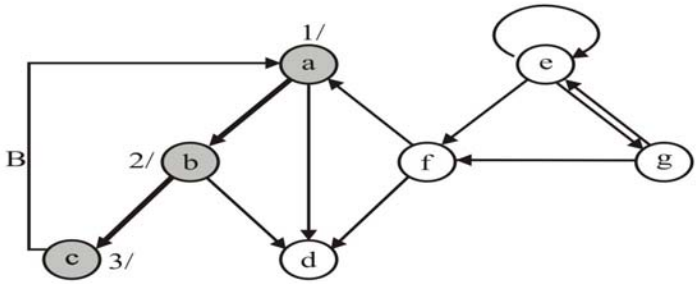


Figure 2.4.2.3: b has two unknown neighbors c and d, by convention c is discovered first i.e., d[c] = 3

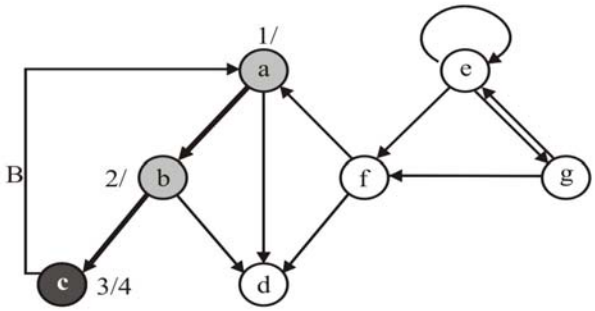


Figure 2.4.2.4: c has only a single neighbor a which is already discovered so c is terminated i.e., t[c] = 5

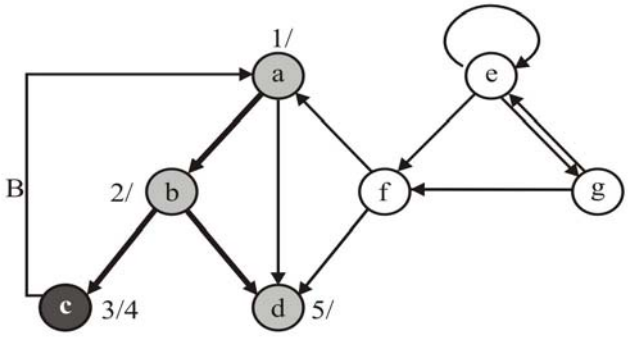


Figure 2.4.2.5: The algorithm backtracks recursively to b, the next unknown neighbor is d, whose status is change to discovered i.e., d[d] = 5

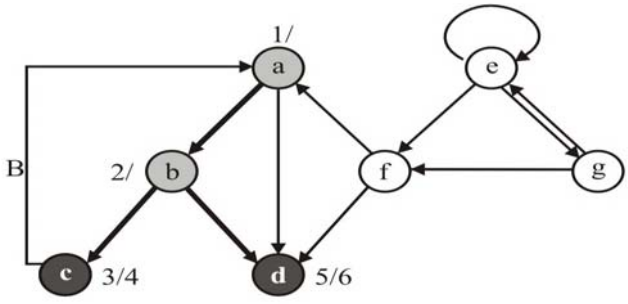


Figure 2.4.2.6: d has no neighbor, so d terminates, t[d] = 6

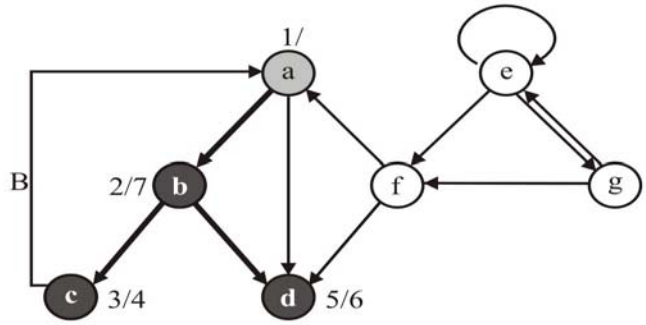


Figure 2.4.2.7: The algorithm backtracks recursively to b, which has no unknown neighbors, so b(terminated) is visited i.e., t[b] = 7

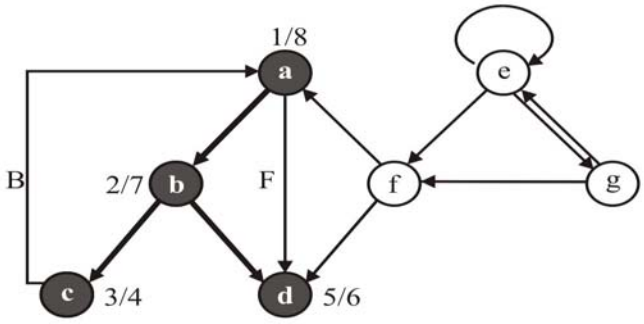


Figure 2.4.2. 8: The algorithm backtracks to a which has no unknown neighbors so a is visited i.e., t[a] = 8.

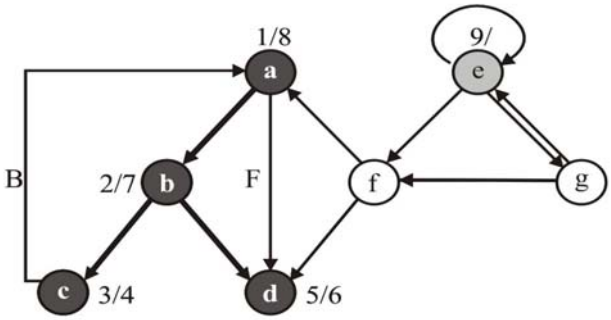


Figure 2.4.2. 9: The connected component is visited so the algorithm moves to next component starting from e (because we are moving in increasing alphabetic order) so e is 'discovered' i.e. , d[e] = 9

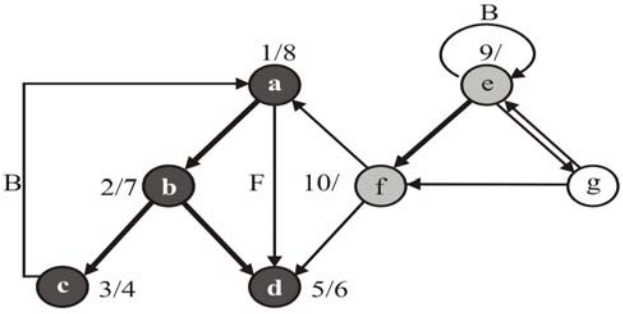


Figure 2.4.2. 10: e has two unknown neighbors f and g, by convention we discover f i.e., d[f] = 10

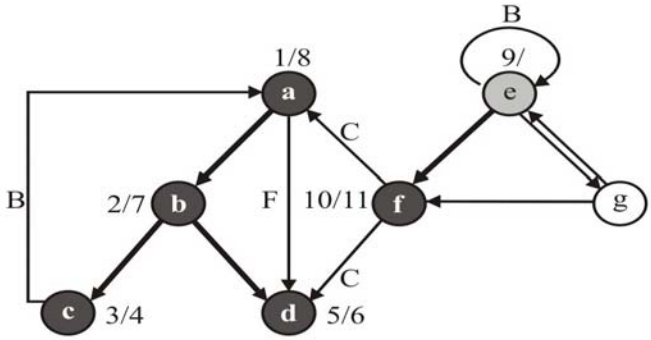


Figure 2.4.2.11: f has no unknown neighbors so f (terminates) is 'visited' i.e., t[f] = 11

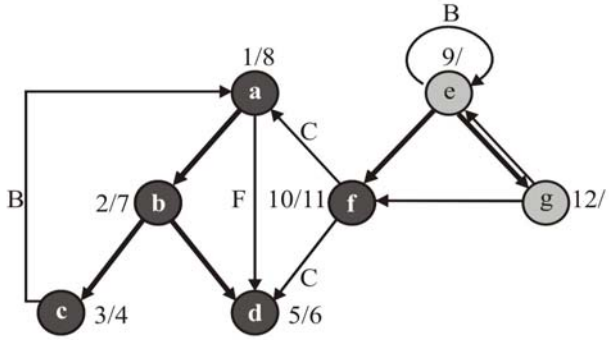


Figure 2.4.2. 12: The algorithm backtracks to e, which has g as the next 'unknown' neighbor, g is 'discovered' i.e., d[g] = 12

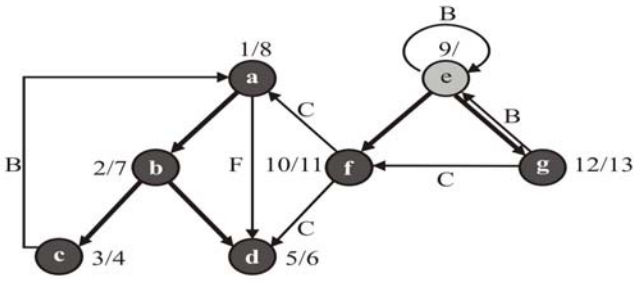


Figure 2.4.2. 13: The only neighbor of g is e, which is already discovered, so g(terminates) is 'visited' i.e., t[g] = 12

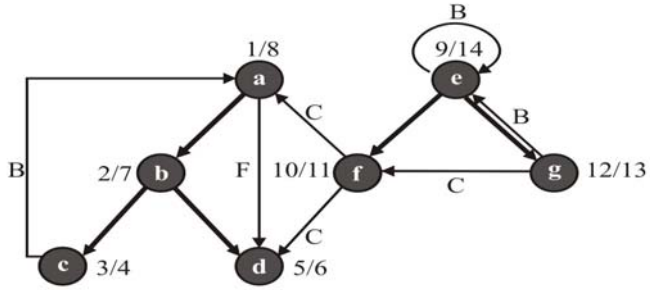


Figure 2.4.2. 14: The algorithm backtracks to e, which has no unknown neighbors left so e (terminates) is visit i.e., t[e] = 14

Some more properties of Depth first search (in directed graph)

(1) Given a directed graph, depth first search can be used to determine whether it contains cycle.

(2) Cross-edges go from a vertex of higher discovery time to a vertex of lower discovery time. Also a forward edge goes from a vertex of lower discovery time to a vertex of higher discovery time.

(3) Tree edges, forward edges and cross edges all go from a vertex of higher termination time to a vertex of lower finish time whereas back edges go from a vertex of lower termination time to a vertex of higher termination time.

(4) A graph is acyclic if and only if f any depth first search forest of graph G yields no back edges. This fact can be realized from property 3 explained above, that if there are no back edges then all edges will go from a vertex of higher termination time to a vertex of lower termination time. So there will be no cycles. So the property which checks cycles in a directed graph can be verified by ensuring there are no back edges.

2.5 BREADTH-FIRST SEARCH

Breadth first search as the name suggests first discovers all vertices adjacent to a given vertex before moving to the vertices far ahead in the search graph. If G(V,E) is a graph having vertex set V and edge set E and a particular source vertex s, breadth first search find or discovers every vertex that is reachable from s. First it discovers every vertex adjacent to s, then systematically for each of those vertices it finds the all the vertices adjacent to them and so on. In doing so, it computes the distance and the shortest path in terms of fewest numbers of edges from the source node s to each of the reachable vertex. Breadth-first Search also produces a breadth-first tree with root vertex s in the process of searching or traversing the graph.

For recording the status of each vertex, whether it is still unknown, whether it has been discovered (or found) and whether all of its adjacent vertices have also been discovered. The vertices are termed as **unknown**, **discovered** and **visited** respectively. So if $(u,v) \in E$ and u is visited then v will be either discovered or visited i.e., either v has just been discovered or vertices adjacent to v have also been found or visited.

As breadth first search forms a breadth first tree, so if in the edge (u,v) vertex v is discovered in adjacency list of an already discovered vertex u then we say that u is the **parent or predecessor** vertex of V. Each vertex is discovered once only.

The data structure we use in this algorithm is a queue to hold vertices. In this algorithm we assume that the graph is represented using adjacency list representation. *front[u]* is used to represent the element at the front of the queue. *Empty()* procedure returns true if queue is empty otherwise it returns false. Queue is represented as Q. Procedure *enqueue()* and *dequeue()* are used to insert and delete an element from the queue respectively. The data structure Status[]is used to store the status of each vertex as unknown or discovered or visited.

2.5.1 Algorithm of Breadth First Search

```
1 for each vertex u ∈ V – {s}
2         status[u] = unknown
3 status[s]=discovered
4 enqueue(Q,s)
5 while(empty[Q] != false)
6         u = front[Q]
7         for each vertex v ∈ Adjacent to u
8                   if status[v] = unknown
```

```
9                       status[v] = discovered
10                      parent (v) = u
11                     end for
12                      enqueue(Q,v);
13      dequeue(Q)
14      status[u] = visited
15      print "u is visited"
16      end while
```

The algorithm works as follows. Lines 1-2 initialize each vertex to 'unknown'. Because we have to start searching from vertex s, line 3 gives the status 'discovered' to vertex s. Line 4 inserts the initial vertex s in the queue. The while loop contains statements from line 5 to end of the algorithm. The while loop runs as long as there remains 'discovered' vertices in the queue. And we can see that queue will only contain 'discovered' vertices. Line 6 takes an element u at the front of the queue and in lines 7 to 10 12 the adjacency list of vertex u is traversed and each unknown vertex v in the adjacency list of u, its status is marked as discovered, its parent is marked as u and then it is inserted in the queue. In the line 13, vertex u is removed from the queue. In line 14-15, when there are no more elements in adjacency list of u, vertex u is removed from the queue its status is changed to 'visited' and is also printed as visited.

The algorithm given above can also be improved by storing the distance of each vertex u from the source vertex s using an array distance[] and also by permanently recording the predecessor or parent of each discovered vertex in the array parent[]. In fact, the distance of each reachable vertex from the source vertex as calculated by the BFS is the shortest distance in terms of the number of edges traversed. So next we present the modified algorithm for breadth first search.

2.5.2 Modified Algorithm

Program BFS(G,s)

```
1 for each vertex u ∈ s v – {s}
2       status[u] = unknown
3       parent[u] = NULL
4       distance[u] = infinity
5 status[s] = discovered
6 distance[s] = 0
7 parent[s] = NULL
8 enqueue(Q,s)
9 while empty(Q) != false
10      u = front[Q]
11      for each vertex v adjacent to u
12              if status[v] = unknown
13                      status[v] = discovered
14                      parent[v] = u
15                      distance[v]  = distance[u] + 1
16                      enqueue(Q,v)
17      dequeue(Q)
18      status[u] = visited
19      print "u is visited"
```

In the above algorithm the newly inserted line 3 initializes the parent of each vertex to NULL, line 4 initializes the distance of each vertex from the source vertex to infinity, line 6 initializes the distance of source vertex s to 0, line 7 initializes the parent of source vertex s NULL, line 14 records the parent of v as u, line 15 calculates the shortest distance of v from the source vertex s, as distance of u plus 1.

Example 2.5.3:

In the figure given below, we can see the graph given initially, in which only source s is discovered.

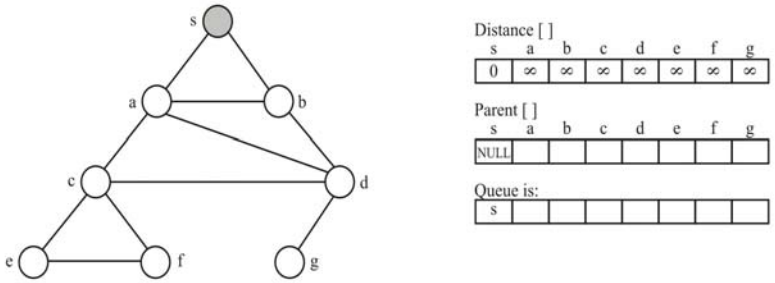


Figure 2.5.1.1: Initial Input Graph

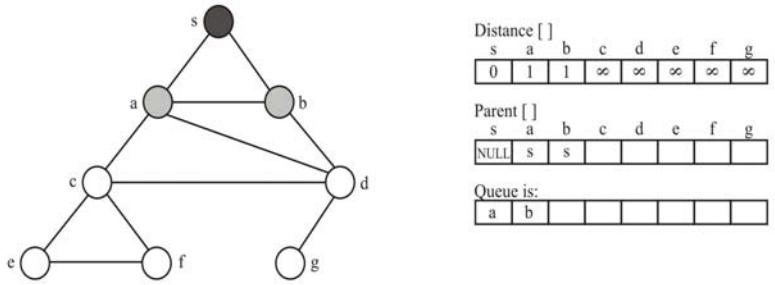


Figure 2.5.1.2: After we visit s

We take unknown (i.e., undiscovered) adjacent vertex of s and insert them in queue, first a and then b. The values of the data structures are modified as given below:

Next, after completing the visit of a we get the figure and the data structures as given below:

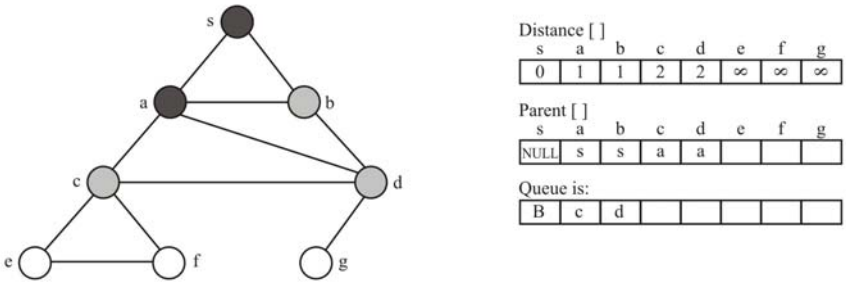


Figure 2.5.1.3: After we visit a

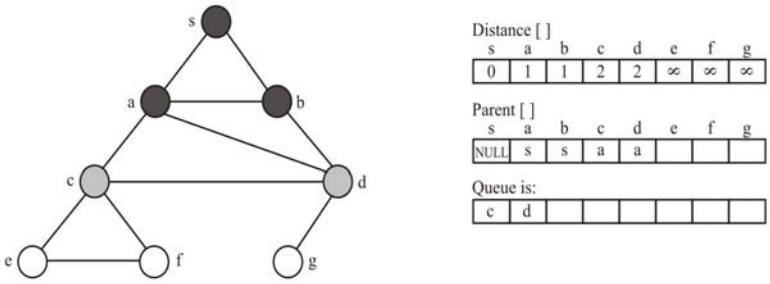


Figure 2.5.1.4: After we visit b

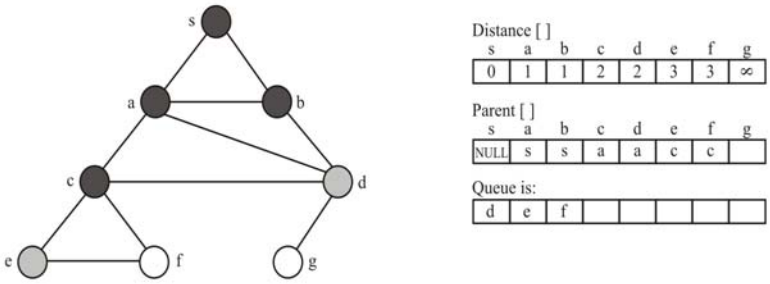


Figure 2.5.1.5: After we visit c

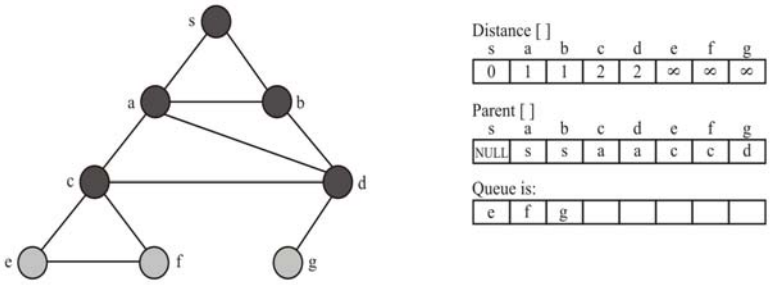


Figure 2.5.1.6: After we visit d

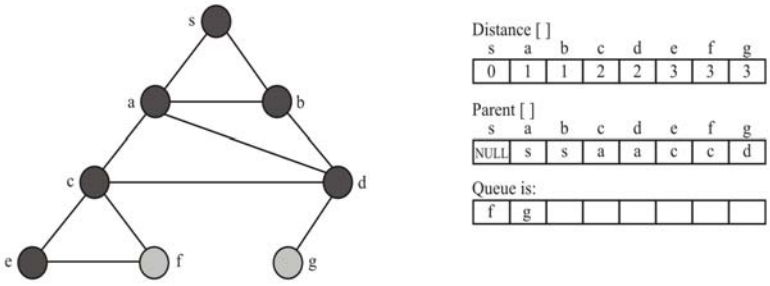


Figure 2.5.1.7: After we visit e

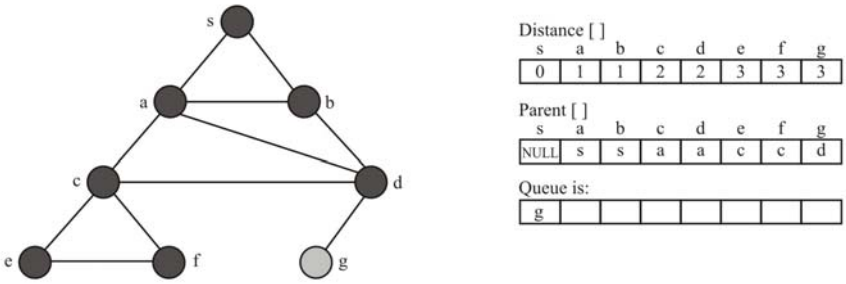


Figure 2.5.1.8: After we visit f

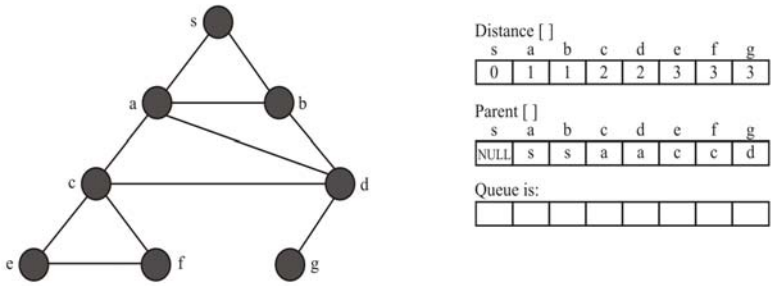


Figure 2.5.1.9: After we visit g

Figure 1: Initial Input Graph

Figure 2: We take unknown (i.e., undiscovered) adjacent vertices of s and insert them in the queue.

Figure 3: Now the gray vertices in the adjacency list of u are b, c and d, and we can visit any of them depending upon which vertex is inserted in the queue first. As in this example, we have inserted b first which is now at the front of the queue, so next we will visit b.

Figure 4: As there is no undiscovered vertex adjacent to b so no new vertex will be inserted in the queue, only the vertex b will be removed from the queue.

Figure 5: Vertices e and f are discovered as adjacent vertices of c, so they are inserted in the queue and then c is removed from the queue and is visited.

Figure 6: Vertex g is discovered as the adjacent vertex of d an after that d is removed from the queue and its status is changed to visited.

Figure 7: No undiscovered vertex adjacent to e is found so e is removed from the queue and its status is changed to visited.

Figure 8: No undiscovered vertex adjacent to f is found so f is removed from the queue and its status is changed to visited.

Figure 9: No undiscovered vertex adjacent to g is found so g is removed from the queue and its status is changed to visited. Now as queue becomes empty so the while loop stops.

2.6 BEST FIRST SEARCH & MINIMAX PRINCIPLE

Best First Search

In the two basic search algorithms we have studied before i.e., depth first search and breadth first search we proceed in a systematic way by discovering/finding/exploring nodes in a predetermined order. In these algorithms at each step during the process of searching there is no assessment of which way to go because the method of moving is fixed at the outset.

The best first search belongs to a branch of search algorithms known as **heuristic search algorithms**. The basic idea of heuristic search is that, rather than trying all possible search paths at each step, we try to find which paths seem to be getting us nearer to our goal state. Of course, we can't be sure that we are really near to our goal state. It could be that we are really near to our goal state. It could be that we have to take some really complicated and circuitous sequence of steps to get there. But we might be able to make a good guess. Heuristics are used to help us make that guess.

To use any heuristic search we need an evaluation function that scores a node in the search tree according to how close to the goal or target node it seems to be. It will just be an estimate but it should still be useful. But the estimate should always be on the lower side to find the optimal or the lowest cost path. For example, to find the optimal path/route between Delhi and Jaipur, an estimate could be straight arial distance between the two cities.

There are a whole batch of heuristic search algorithms e.g., Hill Climbing, best first search, A* ,AO* etc. But here we will be focussing on best first search.

Best First Search combines the benefits of both depth first and breadth first search by moving along a single path at a time but change paths whenever some other path looks more promising than the current path.

At each step in the depth first search, we first generate the successors of the current node and then apply a heuristic function to find the most promising child/successor. We then expand/visit (i.e., find its successors) the chosen successor i.e., find its unknown successors. If one of the successors is a goal node we stop. If not then all these nodes are added to the list of nodes generated or discovered so fart. During this process of generating successors a bit of depth search is performed but ultimately if the solution i.e., goal node is not found then at some point the newly found/discovered/generated node will have a less promising heuristic value than one of the top level nodes which were ignored previously. If this is the case then we backtrack to the previously ignored but currently the most promising node and we expand /visit that node. But when we back track, we do not forget the older branch from where we have come. Its last node remains in the list of nodes which have been discovered but not yet expanded/ visited . The search can always return to it if at some stage during the search process it again becomes the most promising node to move ahead.

Choosing the most appropriate heuristic function for a particular search problem is not easy and it also incurs some cost. One of the simplest heuristic functions is an estimate of the cost of getting to a solution from a given node this cost could be in terms of the number of expected edges or hops to be traversed to reach the goal node.

We should always remember that in best first search although one path might be selected at a time but others are not thrown so that they can be revisited in future if the selected path becomes less promising.

Although the example we have given below shows the best first search of a tree, it is sometimes important to search a graph instead of a tree so we have to take care that the duplicate paths are not pursued. To perform this job, an algorithm will work by searching a directed graph in which a node represents a point in the problem space. Each node, in addition to describing the problem space and the heuristic value associated with it, will also contain a link or pointer to its best parent and points to its successor node. Once the goal node is found, the parent link will allow us to trace the path from source node to the goal node. The list of successors will allow it to pass the improvement down to its successors if any of them are already existing.

In the algorithm given below, we assume two different list of nodes:

- **OPEN list →** is the list of nodes which have been found but yet not expanded i.e., the nodes which have been discovered /generated but whose children/successors are yet not discovered. Open list can be implemented in the form of a queue in which the nodes will be arranged in the order of decreasing prioity from the front i.e., the node with the most promising heuristic value (i.e., the highest priority node) will be at the first place in the list.
- **CLOSED list →** contains expanded/visited nodes i.e., the anodes whose successors are also genereated. We require to keep the nodes in memory I we want to search a graph rather than a tree, since whenever a new node is generated we need to check if it has been generated before.

The algorithm can be written as:

Best First Search

1. Place the start node on the OPEN list.
2. Create a list called CLOSED i.e., initially empty.
3. If the OPEN list is empty search ends unsuccessfully.
4. Remove the first node on OPEN list and put this node on CLOSED list.
5. If this is a goal node, search ends successfully.
6. Generate successors of this node:
   For each successor :
   (a) If it has bot been discovered / generated before i.e., it is not on OPEN, evaluate this node by applying the heuristic function, add it to the OPEN and record its parent.
   (b) If it has been discovered / generated before, change the parent if the new path is better than the previous one. In that case update the cost of getting to this node and to any successors that this node may already have.
7. Reorder the list OPEN, according to the heuristic merit.
8. Go to step 3.

Example

In this example, each node has a heuristic value showing the estimated cost of getting to a solution from this node. The example shows part of the search process using best first search.

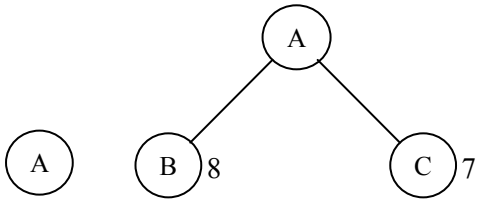


Figure 1

Figure 2

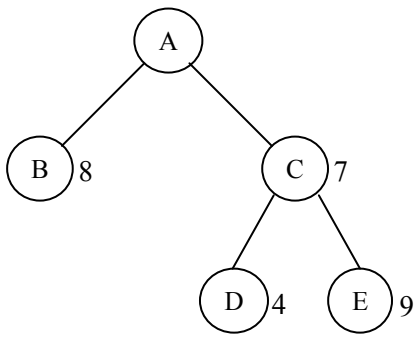


Figure 3

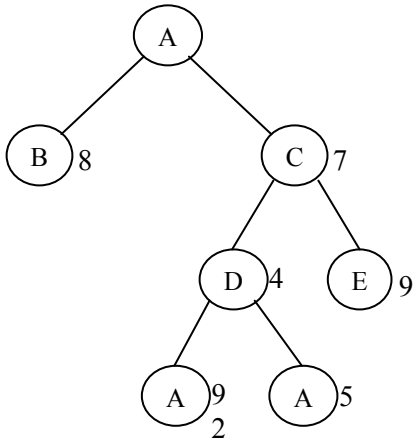


Figure 4

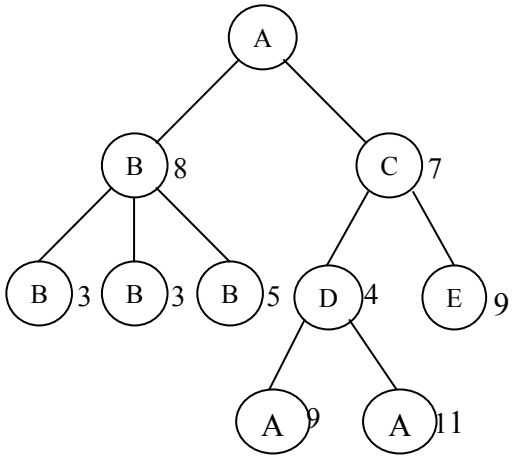


Figure 5

Figure 1: A is the starting node

Figure 2: Generate its successors B and C

Figure 3: As the estimated goal distance of C is less so expand C to find its successors d and e.

Figure 4: Now D has lesser estimated goal distance i.e., 4 , so expand D to generate F and G with distance 9 and 11 respectively.

Figure 5: Now among all the nodes which have been discovered but yet not expanded B has the smallest estimated goal distance i.e., 8, so now backtrack and expand B and so on.

Best first searches will always find good paths to a goal node if there is any. But it requires a good heuristic function for better estimation of the distance to a goal node.

The Minimax Principle

Whichever search technique we may use, it can be seen that many graph problems including game problems, complete searching of the associated graph is not possible. The alternative is to perform a partial search or what we call a limited horizon search from the current position. This is the principle behind the minimax procedure.

Minimax is a method in decision theory for minimizing the expected maximum loss. It is applied in two players games such as tic-tac-toe, or chess where the two players take alternate moves. It has also been extended to more complex games which require general decision making in the presence of increased uncertainty. All these games have a common property that they are logic games. This means that these games can be described by a set of rules and premises. So it is possible to know at a given point of time, what are the next available moves. We can also call them full information games as each player has complete knowledge about possible moves of the adversary.

In the subsequent discussion of games, the two players are named as MAX and MIN. We are using an assumption that MAX moves first and after that the two players will move alternatively. The extent of search before each move will depend o the play depth – the amount of lookahead measured in terms of pairs of alternating moves for MAX and MIN.

As I have already specified, complete search of most game graphs is computationally infeasible. It can be seen that for a game like chess it might take centuries to generate the complete search graph even in an environment where a successor could be

generated in a few nanoseconds. Therefore, for many complex games, we must accept the fact that search to termination is impossible instead we must use partial searching techniques.

For searching we can use either breadth first, depth first or heuristic methods except that the termination conditions must now be specified. Several artificial termination conditions can be specified based on factors such as time limit, storage space and the depth of the deepest node in the search tree.

In a two player game, the first step is to define a **static evaluation function efun()**, which attaches a value to each position or state of the game. This value indicates how good it would be for a player to reach that position. So after the search terminates, we must extract from the search tree an estimate of the best first move by applying a static evaluation function efun() to the leaf nodes of the search tree. The evaluation function measures the worth of the leaf node position. For example, in chess a simple static evaluation function might attach one point for each pawn, four points for each rook and eight points for queen and so on. But this static evaluation is too easy to be of any real use. Sometimes we might have to sacrifice queen to prevent the opponent from a winning move and to gain advantage in future so the key lies in the amount of lookahead. The more number of moves we are able to lookahead before evaluating a move, the better will be the choice.

In analyzing game trees, we follow a convention that the value of the evaluation function will increase as the position becomes favourable to player MAX , so **the positive values will indicate position that favours MAX** whereas for the **positions favourable to player MIN are represented by the static evaluation function having negative values** and values near zero correspond to game positions not favourable to either MAX or MIN. In a terminal position, the static evaluation function returns either positive infinity or negative infinity where as positive infinity represents a win for player MAX and negative infinity represents a win for the player MIN and a value zero represents a draw.

In the algorithm, we give ahead, the search tree is generated starting with the current game position upto the end game position or lookahead limit is reached. Increasing the lookahead limit increases search time but results in better choice. The final game position is evaluated from the MAX's point of view. The nodes that belong to the player MAX receive the maximum value of its children. The nodes for the player MIN will select the minimum value of its children.

In the algorithm, lookahead limit represents the lookahead factor in terms of number of steps, u and v represent game states or nodes, maxmove() and minmove() are functions to describe the steps taken by player MAX or player MIN to choose a move, efun() is the static evaluation function which attaches a positive or negative integer value to a node (i.e., a game state), value is a simple variable.

Now to move number of steps equal to the lookahead limit from a given game state u, MAX should move to the game state v given by the following code :

```
maxval = -
    for each game state w that is a successor of u
      val = minmove(w,lookaheadlimit)
        if (val >= maxval)
            maxval = val
            v = w              // move to the state v
```

The minmove() function is as follows :

```
minmove(w, lookaheadlimit)
{
  if(lookaheadlimit = = 0 or w has no successor)
```

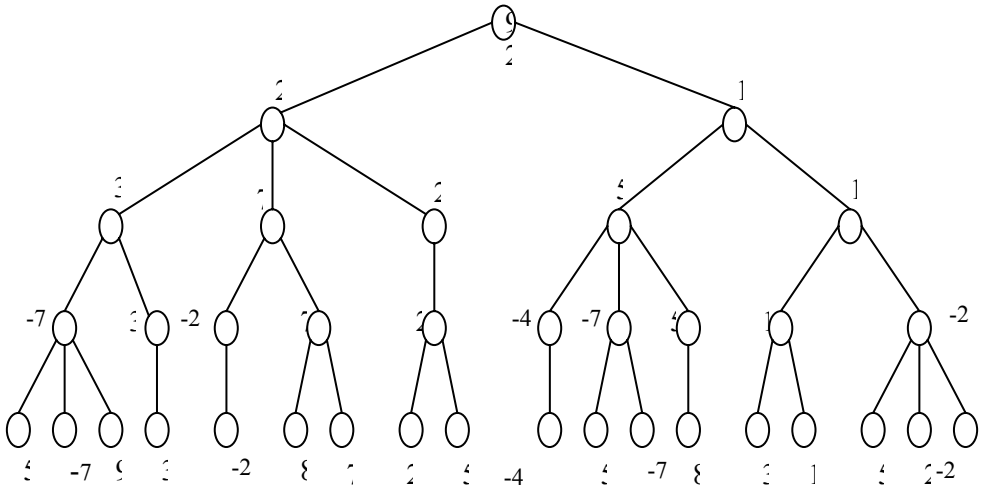
```
    return efun(w)
  else
      minval = +
      for each successor x of w
            val = maxmove(x,lookaheadlimit – 1)
            if (minval > val)
                  minval = val
      return(minval)
}
```

The maxmove() function is as follows :

```
maxmove(w, lookaheadlimit)
{
   if (lookaheadlimit = =  0 or w has no successor)
         return efun(w)
   else
         maxval = -
         for each successor x of w
               val = minmove(x,lookaheadlimit – 1)
               if (maxval < val)
                     maxval = val
         return(maxval)
}
```

We can see that in the minimax technique, player MIN tries to minimize te advantage he allows to player MAX, and on the other hand player MAX tries to maximize the advantage he obtains after each move.

Let us suppose the graph given below shows part of the game. The values of leaf nodes are given using efun() procedure for a particular game then the value of nodes above can be calculated using minimax principle.Suppose the lookahead limit is 4 and it is MAX's turn.

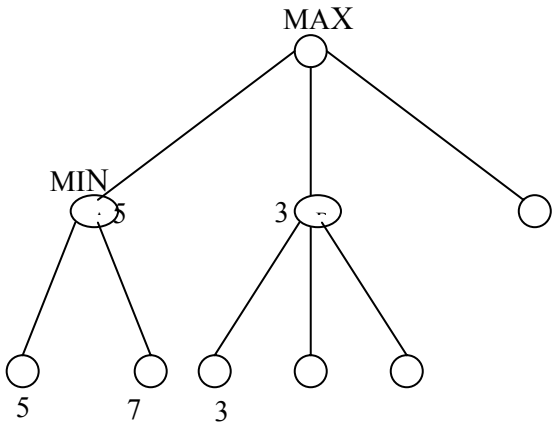


Speeding up the minimax algorithm using Alpha-Beta Pruning

We can take few steps to reduce the search time in minimax procedure. In the figure given below, the value for node A is 5 and the first found value for the subtree starting at node B is 3. So since the B node is at the MIN player level, we know that the selected value for the B node must be less than or equal to 3. But we also know that the A node has the value 5 and both A and B nodes share the same parent at the MAX level immediately above. This means that the game path starting at the B node can never be selected because 5 is better than 3 for the MAX node. So it is not worth

spending time to search for children of the B node and so we can safely ignore all the remaining children of B.

This shows that **the search on same paths can sometimes be aborted** (i.e., it is not required to explore all paths) because we find out that the search subtree will not take us to any viable answer.



This optimization is known as alpha beta pruning/procedure and the values, below which search need not be carried out are known as alpha beta cutoffs.

A general algorithm for alpha beta procedure is as follows:

1. Have two values passed around the tree nodes:

   The alpha value - which holds best MAX value found at the MAX level
   The beta value - which holds best Min value found at the MIN level

2. At MAX player level, before evaluating each child path, compare the returned value of the previous path with the beta value. If the returned value is greater then abort the search for the current node.

3. At Min player level,before evaluating eachh child path, compare the returned value of the previouss path with the alpha value. If the value is lesser then abort the searh for the current node.

We should note that:

- The alpha values of MAX nodes (including the start value) can never decrease.
- The beta value of MIN nodes can never increase.

So we can see that remarkable reductions in the amount of search needed to evaluate a good move are possible by using alpha beta pruning / procedure.

Analysis of BFS Algorithm

In the algorithm BFS, let us analyse the running time taken by the algorithm on a graph G. We can see that each vertex is inserted n the queue exactly once and also deleted from the queue exactly once. So for each the insertion and deletion from the queue costs O(1) time therefore for all vertices queue insertion and deletion would cost O(V) time. Because graph is represented using adjacency list and adjacency list of each vertex is scanned at most once. We can see that the total length of all adjacency list is no. of edge E in the graph G. So a total of O(E) time to spent in scanning all adjacency lists. The initialization portion costs O(V). So the total running time of BFS is O(V+E).

2.7 TOPOLOGICAL SORT

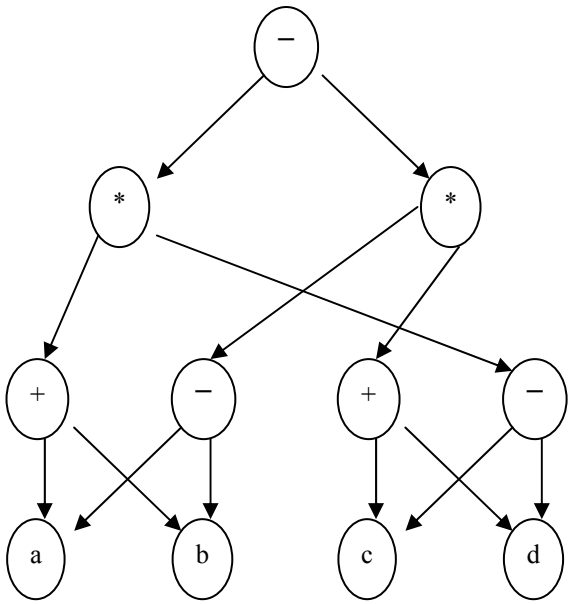
In many applications we are required to indicate precedences or dependencies among various events. Using directed graphs we can easily represent these dependencies. Let a directed graph G with vertex set V and edge set E. An edge from a vertex u to vertex v in the directed graph will then mean that v is dependent on u or v precedes v. Also there cannot be any cycles in these dependency graphs as it can be seen from the following simple argument. Suppose that u is vertex and there is an edge from u to u, i.e., there is a single node cycle. But the graph is dependency graph; this would mean that vertex u is dependent on vertex u which means u must be processed before u which is impossible.

A directed graph that does not have any cycles is known as directed acyclic graph. Hence, dependencies or precedences among events can be represented by using directed acyclic graphs.

There are many problems in which we can easily tell which event follows or precedes a given event, but we can't easily work out in which order all the events are held. For example, it is easy to specify/look up prerequisite relationships between modules in a course, but it may be hard to find an order to take all the modules so that all prerequisite material is covered before the modules that depend on it. Same is the case with a compiler evaluating sub-expressions of an expression like the following:

$$(a + b)(c - d) - (a - b)(c + d)$$

Both of these problems are essentially equivalent. The data of both problems can be represented by directed acyclic graph (See figure below). In the first each node is a module; in the second example each node is an operator or an operand. Directed edges occur when one node depends on the other, because of prerequisite relationships among courses or the parenthesis order of the expression. The problem in both is to find an acceptable ordering of the nodes satisfying the dependencies. This is referred to as a topological ordering. More formally it is defined below.



Directed Acyclic Graph

A **topological sort** is a linear ordering of vertices in a **directed acyclic graph** (normally called **dag)** such that, if there is path from node **u** to node **v**, then **v** appears after **u** in the ordering. Therefore, a cyclic graph cannot have a topological order. A

topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go in one direction.

The term topological sort comes from the study of partial orders and is sometimes called a *topological* order or *linear order*.

ALGORITHM FOR TOPOLOGICAL SORT

Our algorithm picks vertices from a DAG in a sequence such that there are no other vertices preceding them. That is, if a vertex has in-degree 0 then it can be next in the topological order. We remove this vertex and look for another vertex of in-degree 0 in the resulting DAG. We repeat until all vertices have been added to the topological order.

The algorithm given below assumes that the direceted acyclic graph is represented using adjacency lists. Each node of the adjacency list contains a variable indeg which stores the indegree of the given vertex. Adj is an array of |V| lists, one for each vertex in V.

```
Topological-Sort(G)
1 for each vertex u ε G
2         do indeg[u] = in-degree of vertex u
3         if indeg[u] = 0
4                   then enqueue(Q,u)
5 while Q != 0
6         do u = dequeue(Q)
7              print u
8              for each v ε Adj[u]
9                   do indeg[v] = indeg[v] – 1
10                       if indeg[v] = 0
11                            then enqueue(Q,v)
```

The for loop of lines 1-3 calculates the indegree of each node and if the indegree of any node is found to be 0, then it is immediately enqueeud. The while loop of lines 5-11 works as follows. We dequeue a vertex from the queue. Its indegree will be zero(Why?). It then output the vertex, and decrement the in degree of each vertex adjacent to u. If in the process, the in degree of any vertex adjacent to u becomes 0, then it is also enqueued.

TOPOLOGICAL SORT – ANOTHER APPROACH

We can also use Depth First Search Traversal for topologically sorting a directed acyclic graph. DFS algorithm can be slightly changed or used as it is to find the topological ordering. We simply run DFS on the input directed acyclic graph and insert the vertices of a node in a linked list or simply print the vertices in decreasing order of the termination time.

To see why this approach work, suppose that DFS is run on a given dag G = (V,E) to determine the finishing times for its vertices. Let u, v ε V, if there is an edge in G from u to v, then termination time of v will be less than termination time of u i.e., t[v] < t[u]. Since, we output the vertices in decreasing order of termination time, the vertex with least number of dependencies will be outputted first.

ALGORITHM

1. Run the DFS algorithm on graph G. In doing so compute the termination time of each vertex.
2. Whenever a vertex is terminated (i.e. visited), insert it in the front o f a list.
3. Output the list.

RUNNING TIME

Let n is the number of vertices (or nodes, or activities) and m is the number of edges (constraints). As each vertex is discovered only once, and for each vertex we loop over all its outgoing edges once. Therefore, total running time is O(n+m).

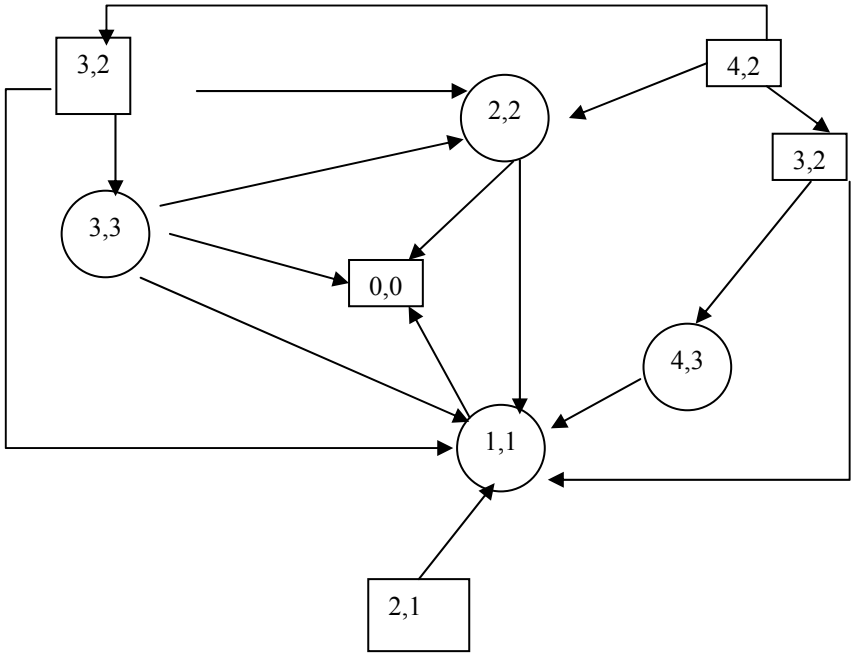
2.8 SUMMARY

This unit discusses some searching and sorting techniques for sorting those problems each of which can be efficiently represented in the form of a graph. In a graphical representation of a problem, generally, a node represents a state of the problem, and an arrow/arc represents a move between a pair of states.

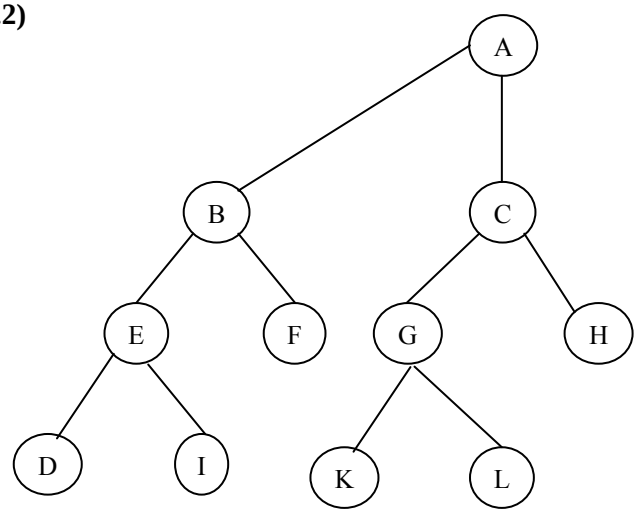
Graph representation of a problem is introduced through the example of a game of NIM/ Marienbad. Then a number of search algorithms viz., Depth-First, Breadth-First, Best-First, and Minimax principal are discussed. Next, a sorting algorithm viz., Topological sort is discussed.

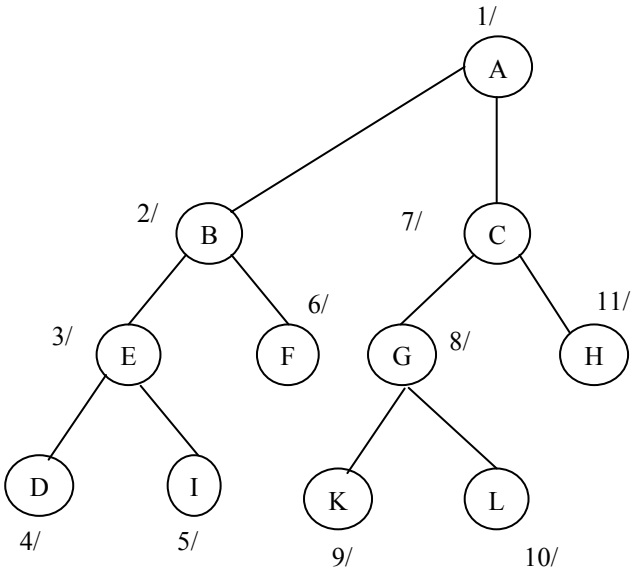
2.9 SOLUTIONS/ANSWERS

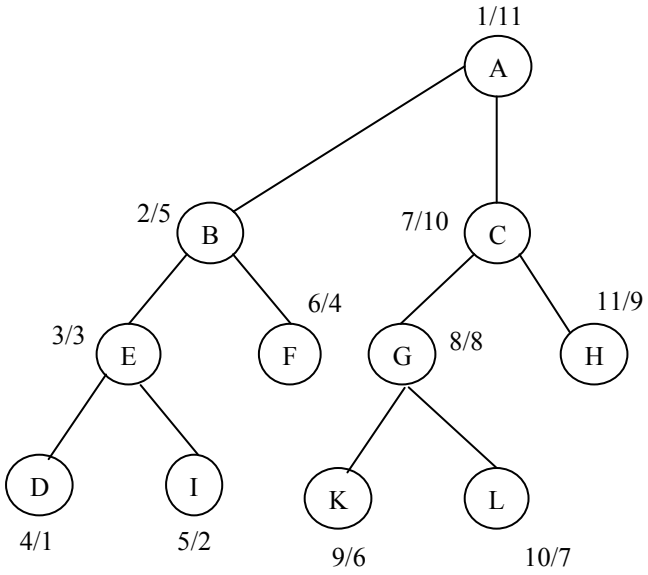
Ex. 1)



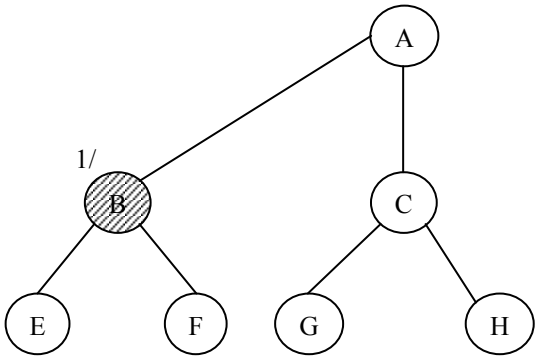
Ex.2)

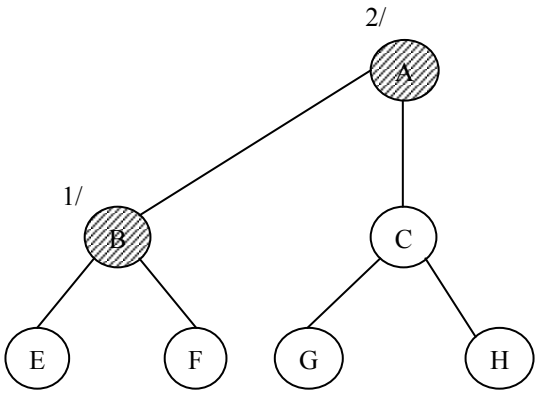


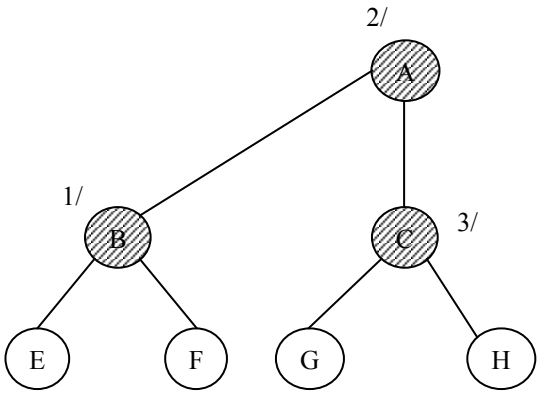

1/
A
2/
B
7/
C
3/
E
6/
F
G
8/
11/
H
D
I
K
L
4/
5/
9/
10/


1/11
A
2/5
B
7/10
C
3/3
E
6/4
F
G
8/8
11/9
H
D
I
K
L
4/1
5/2
9/6
10/7

Ex.3)


A
1/
B
C
E
F
G
H





2.10 FURTHER READINGS

1. *Discrete Mathematics and Its Applications* (*Fifth Edition*) K.N. Rosen: Tata McGraw-Hill (2003).

2. *Introduction to Algorithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice-Hall of India (2002).

UNIT 1 DYNAMIC PROGRAMMING

Structure **Page Nos.**

1.0 INTRODUCTION

In the earlier units of the course, we have discussed some well-known techniques, including the divide-and-conquer technique, for developing algorithms for algorithmically solvable problems. The divide-and-conquer technique, though quite useful in solving problems from a number of problem domains, yet in some cases, as shown below, may give quite *inefficient* algorithms to solve problems.

Example 1.0.1: Consider the problem of computing binomial coefficient $\binom{\mathbf{n}}{\mathbf{k}}$ (or in linear notation **c (n, k))** where n and k are given non-negative integers with n≥k. One way of defining and calculating the binomial coefficient is by using the following recursive formula

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = n \text{ or } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 0 & \text{otherwise} \end{cases} \qquad (1.0.1)$$

The following recursive algorithm named *Bin (n, k),* implements the above formula for computing the binomial coefficient.

Function Bin (n, k)

If k = n or k = 0 **then return** 1
else return Bin (n–1, k–1) + Bin (n–1, k)

For computing Bin (n, k) for some given values of n and k, a number of terms Bin (i, j), 1≤ i ≤ and 1 ≤ j ≤ k, particularly for smaller values of i and j, are *repeatedly* calculated. For example, to calculate Bin (7, 5), we compute Bin (6, 5) and Bin (6, 4). Now, for computing Bin (6, 5), we compute Bin (5, 4) and Bin (5, 5). But for calculating Bin (6, 4) we have to calculate Bin (5, 4) again. If the above argument is further carried out for still smaller values, the number of repetitions for Bin (i, j) increases as values of i and j decrease.

For given values of n and k, in order to compute Bin (n, k), we need to call Bin (i, j) for 1 ≤ i ≤ n − 1 and 1 ≤ j ≤ k−1 and as the values of i and j decrease, the number of times Bin (i, j) is required to be called and executed generally increases.

The above example follows the *Divide-and-Conquer* technique in the sense that the task of calculating C(n, k) is replaced by the two relatively simpler tasks, viz., calculating C(n–1, k) and C (n–1, k–1). But this technique, in this particular case,

makes large number of *avoidable repetitions* of computations. This is not an isolated instance where the Divide-and-Conquer technique leads to *inefficient* solutions. In such cases, an alternative technique, viz., *Dynamic Programming*, may prove quite useful. This unit is devoted to developing algorithms using Dynamic Programming technique. But before, we discuss the technique in more details, let us briefly discuss *underlying idea* of the technique and the *fundamental difference* between Dynamic Programming and Divide-and-Conquer technique.

Essential idea of Dynamic Programming, *being quite simple, is that we should avoid calculating the same quantity more than once, usually by keeping a table of known results for simpler instances*. These results, in stead of being calculated repeatedly, can be retrieved from the table, as and when required, after first computation.

Comparatively, ***Divide-and-Conquer*** **is conceptually a** ***top-down*** **approach** for solving problems or developing algorithms, in which the problem is attempted *initially* with *complete* instance and gradually replacing the more complex instances by simpler instances.

On the other hand, ***Dynamic Programming*** **is a** ***bottom-up*** **approach** for solving problems, in which we first attempt the *simplest* subinstances of the problem under consideration and then gradually handle more and more complex instances, using the results of earlier computed (sub) instances.

We illustrate the bottom-up nature of Dynamic Programming, by attempting solution of the problem of computing binomial coefficients. In order to calculate C(n, k), for given numbers n and k, we consider an n × k table or matrix of the form shown below.

The (i, j) th entry of the table contains the value C(i, j). We know,

C (i, 0) = 1 for all i = 0, 1, 2,…., n and
C (0, j) = 0 for j = 1, 2, …., k

Thus, initially the table looks like

	0	1	2	3	……..	k
0	1	0	0	0	……..	0
1	1					
2	1					
3	1					
.	.					
.	.					
.	.					
.	.					
n	1					

Then row C (1, j) for j = 1, 2, …. k may be filled up by using the formula

C (i, j) = C (i – 1, j – 1) + C (i – 1, j).

After filling up the entries of the first row, the table takes the following form:

	0	1	2	3	……..	k
0	1	0	0	0		0
1	1	1	0	0		0
2	1					
.	1					
.	.					
.	.					
.	.					
n	1					

From the already calculated values of a given row i, adding successive pairs of consecutive values, we get the values for (i + 1)th row. After completing the entries for row with index 4, the table may appear as follows, where the blank entries to the right of the main diagonal are all zeros.

	0	1	2	3	4	…..	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
.	.						
.	.						
.	.						
.	.						
n	1						

***We Summarize below the process followed above, for calculating C(i, j)*:**

First of all, the simplest values C(i, 0) = 1 for i = 1, 2, …., n and C(0, j) = 0 for $j \geq 1$, are obtained directly from the given formula. Next, more complex values are calculated from the already available less complex values. Obviously, the above mentioned process is a *bottom-up* one.

Though the purpose in the above discussion was to introduce and explain the Dynamic Programming technique, yet we may also consider the complexity of calculating C (n, k) using the tabular method.

Space Complexity/Requirement: In view of the following two facts, it can be easily seen that the amount of space required is not for all the n × k entries but only for k values of any row C (i, j) for j = 1, 2, ….., k independent of the value of i:

(i) The value in column 0 is always 1 and hence need not be stored.

(ii) Initially 0^{th} row is given by C(0,0) = 1 and C (0, j) = 0 for j = 1, 2, …., k. Once any value of row 1, say C(1, j) is calculated the values C(0, j–1) and C(0, j) are no more required and hence C (1, j) may be written in the space currently occupied by C(0, j) and hence no extra space is required to write C (1, j).

In general, when the value C(i, j) of the ith row is calculated the value C (I – 1, j) is no more required and hence the cell currently occupied by C (i –1, j) can be used to store

the value C(i, j). Thus, at any time, one row worth of space is enough to calculate C(n, k). Therefore, space requirement is θ (k).

Time Complexity: If we notice the process of calculating successive values of the table, we find that any value C (i, j) is obtained by adding 1's. For example, C(4, 2) = 6 is obtained by adding C(3, 1) = 3 and C(3, 2) = 3. But then C(3, 1) is obtained by adding C(2, 0) = 1 and C(2, 1) = 2. Again C(2, 1) is obtained by adding C(1, 0) = 1 and C(1, 1) = 1. Similarly, C(3, 2) and hence C(4, 2) can be shown to have been obtained by adding, directly or indirectly, a sequence of 1's. Also, the number of additions for calculating C(n, k) can not be more than all the entries in the (n−1) rows viz., 1, 2, ….. (n−1), each row containing k elements.
Thus, number of additions ≤ n k.

Therefore, the time complexity is θ (n k).

In the next sections, we shall discuss solution of some well-known problems, using Dynamic Programming technique.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the dynamic programming technique for solving optimization problems;
- apply the dynamic programming technique for solving optimization problems. specially you should be able to solve the following well-known problems using this technique:
  - Shortest paths problems
  - Knapsack Problem
  - Chained Matrix Multiplication Problem
  - Problem of making change
- Understand the principle of optimality.

1.2 THE PROBLEM OF MAKING CHANGE

First of all, we state a special case of the problem of making change, and then discuss the problem in its general form.

We, in India, have currency notes or coins of denominations of Rupees 1, 2, 5, 10, 20, 50, 100, 500 and 1000. Suppose a person has to pay an amount of Rs. 5896 after having decided to purchase an article. Then, *the **problem** is about how to pay the amount using **minimum** number of coins/notes.*

A simple, frequently and unconsciously used **algorithm** based on Greedy technique is that after having collected an amount A < 5896, choose a note of denomination D, which is s.t

(i) A + D ≤ 5896 and

(ii) D is of maximum denomination for which (i) is satisfied, i.e., if E > D then A+E > 5896.

***In general, the Change Problem may be stated as follows*:**

Let $d_1, d_2, \ldots d_k$, with $d_i > 0$ for i = 1, 2, …, k, be the only coins that are available such that each coin with denomination d_i is available in sufficient quantity for the purpose

of making payments. Further, let A, a positive integer, be the amount to be paid using the above-mentioned coins. The problem is to use the *minimum* number of coins, for the purpose.

The problem with above mentioned algorithm based on greedy technique, is that in some cases, it may either *fail* or may yield *suboptimal* solutions. In order to establish *inadequacy* of greedy technique based algorithms, we consider the following two examples.

Example 1.2.1: Let us assume a hypothetical situation in which we have supply of rupee-notes of denominations 5, 3 and 2 and we are to collect an amount of Rs. 9. Then using greedy technique, first we choose a note of Rupees 5. Next, we choose a 3-Rupee note to make a total amount of Rupees 8. But then according to greedy technique, we can not go ahead in the direction of collecting Rupees 9. The failure of greedy technique is because of the fact that there is a solution otherwise, as *it is possible* to make payment of Rupees 9 using notes of denominations of Rupees 5, Rupees 3 and Rupees 2, viz., 9 = 5+2+2.

Example 1.2.2: Next, we consider another example, in which greedy algorithm may yield a solution, but the solution may not be optimal, but only *suboptimal*. For this purpose, we consider a hypothetical situation, in which currency notes of denominations 1, 4 and 6 are available. And, we have to collect an amount of 8 = 6+1+1. But this solution uses **three currency** notes/coins, whereas another solution using only **two currency** notes/coins, viz., 8 = 4+4, is available.

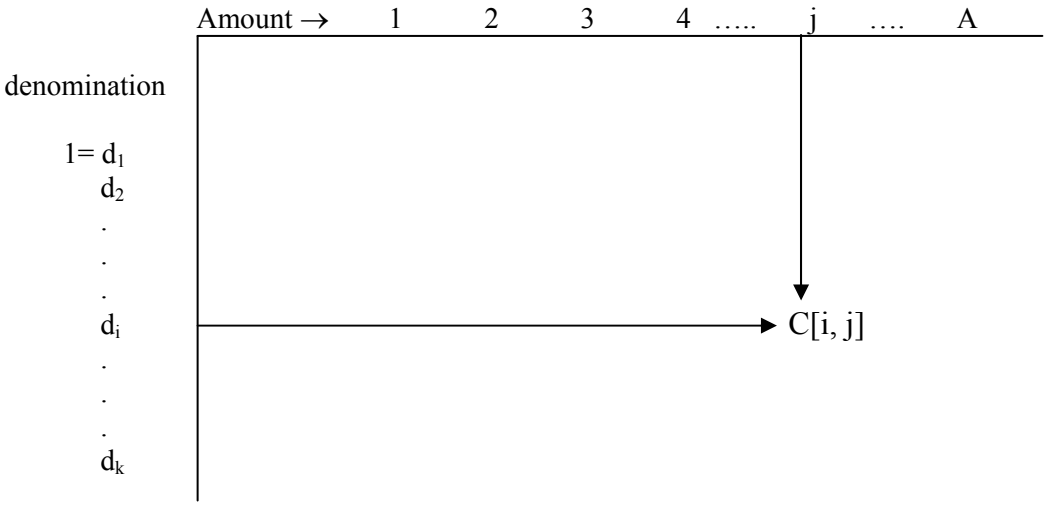
Next, we discuss how the Coin Problem is solved using Dynamic Programming technique.

As mentioned earlier, to solve the coin problem using Dynamic Programming technique, we construct a table, some of the entries in which, corresponding to simple cases of the problem, are filled up *initially* from the definition of the problem. And then recursively entries for the more complex cases, are calculated from the already known ones.

We recall the definition of the coin problem:

Pay an amount A using minimum number of coins of denominations d_i, $1 \le i \le k$. It is assumed coins of each denomination, are available in sufficient quantities.

Each of the denomination d_i, $1 \le i \le k$, is made a row label and each of the value j for $1 \le j \le A$ is made a column label of the proposed table as shown below, where A is the amount to be paid:



In the table given on page no. 9, $0 < d_1 < d_2 < \ldots < d_k$ and **C[i, j]** denotes the minimum number of coins of denominations $d_1, d_2, \ldots, d_i$ (only) that is used to make an amount j, where more than one coin of the same denomination may be used. The value C[i, j] is entered in the table with row label d_i and column label j.

Next, in respect of entries in the table, we make the following two observations:

(i) In order to collect an amount 0, we require zero number of coins, which is true whether we are allowed to choose from, say, one of the successively larger sets of denominations viz., $\{d_1\}, \{d_1, d_2\}, \{d_1, d_2, d_3\}, \ldots, \{d_1, d_2, \ldots, d_k\}$. Thus, entries in the table in the column with 0 as column label, are all 0's.

(ii) If $d_1 \neq 1$, then there may be some amounts j (including j = 1) for which, **even with dynamic programming technique, no solution may exist, i.e., there may not be any number of coins of denominations $d_1, d_2, \ldots d_k$ for which the sum is j.** ***Therefore, we assume $d_1 = 1$.*** **The case $d_1 \neq 1$ may be handled similarly**.

As $d_1 = 1$, therefore, the first row of the table in the jth column, contains j, the number of coins of denomination only $d_1 = 1$ to form value j.

Thus, the table at this stage looks like–

	0	1	2	3	4	…..	j	….	A
d_1	0	1	2	3	4	…	j	…	A
d_2	0								
d_i	0								
.	.								
.	.								
.	.								
d_k	0								

Next for $i \geq 2$ and $j \geq 1$, the value C[i , j], the minimum number of **coins of denominations upto d_i (only)** to sum up to j, can be obtained recursively through either of the following two ways:

(i) **We choose** a coin of denomination d_i the largest, available at this stage; to make up j (provided, of course, $j \geq d_i$). In this case, rest of the amount to be made out of coins of denomination $d_1, d_2, \ldots, d_i$ is $(j - d_i)$. Thus, in this case, we may have–

$$C[i, j] = 1 + C[i, j - d_i], \quad j \geq d_i \text{ if } j \geq d_i \qquad (1.2.1)$$

(ii) **We do not choose** a coin of denomination d_i even when such a coin is available. Then we make up an amount j out of coins of denominations $d_1, d_2, \ldots, d_{i-1}$ (only). Thus, in this case, we may have

$$C[i, j] = C[i-1, j] \quad \text{if } i \geq 1 \qquad (1.2.2)$$

If we choose to fill up the table row-wise, in increasing order of column numbers, we can easily see from (i) and (ii) above that both the values – $C[i, j - d_i]$ and $C[i-1, j]$ are already known for comparison, to find out better alternative for C[i, j].

By definition, C[i, j] = min { 1 + C[i, j – d_i], C [i–1, j]} and can be calculated, as the two involved values viz., C [i, j – d_i] and C [i–1, j] are already known.

Comment 1.2.3

If j < d_i in case (1.1) then the Equation (1.1) is impossible. Mathematically we can say C [i, j – d_i] = ∞ if j < d_i, because, then the case is automatically excluded from consideration for calculating C[i, j].

Similarly we take

C[i–1, j] = ∞ if i < 1

Following the above procedure, C{k, A] gives the desired number.

In order to explain the above method, let us consider the earlier example *for which greedy algorithm gave only suboptimal solution.*

Example 1.2.4: *Using Dynamic Programming technique, find out minimum number of coins required to collect Rupees 8 out of coins of denominations 1, 4, 6.*

From the earlier discussion we already know the following portion of the table to be developed using Dynamic Programming technique.

	0	1	2	3	4	5	6	7	8
d_1 = 1	0	1	2	3	4	5	6	7	8
d_2= 4	0								
d_i= 6	0								

Next, let us **calculate C[2,1],** which by definition, is given by

C[2, 1] = min {1 + c [1, 1– 4], c [1,1]}

By the comment above C [1, – 3] = ∞

∴ C [2, 1] = C [1, 1] = 1

Similarly, we can show that

C[3, 1] = C[2, 1] = 1

Next we consider

C[2, 2] = min [1 + C (2, – 2), C(1, 2)]

Again C[2, – 2] = ∞

Therefore, C[2, 2] = [1, 2] = 2

Similarly, C[3, 2] = 2

On the similar lines, we can show that

C[2, 3] = C[1, 3] = 3

C[3, 3] = C[2, 3] = 3

Next, interesting case is C[2, 4], i.e., to find out minimum number of coins to make an amount 4 out of coins of denominations 1, 4, 6. By definition,

C[2, 4] = min {1 + C (2, 4 – 4), C(1, 4)}
But C[2, 0] = 0 and C [1, 4] = 4, therefore,
= min {1 + 0, 4} = 1 C[2, 4]

By following the method explained through the above example steps, finally, we get the table as

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	**1**	2	3	4	**2**
$d_i = 6$	0	1	2	3	**1**	2	**1**	2	**2**

Let us formalize the method explained above of computing C[k, A], in the general case, in the form of the following algorithm:

Function Min-Num-Coins (A, D[1..k])

{gives the minimum number of coins to add up to A where the number k of denominations and the value A are supplied by the calling procedure/program}

```
array C [1… k,   0… A]
For i = 1 to k
Read (d [i ] )
{reads the various denominations available with each denomination coin in sufficient
numbers}.
{assuming d1 = 1, initialize the table C {  } as follows}

For i = 1 to k
        C[ i, 0] = 0
For j = 1to A
       C [ 1, j] = j

If i = 1 and j < d [ i ] then
        C [ i, j] = ∞
else
        if
        j < d [ i ] then
        C [ i, j] = C [ i – 1, j]
else

        C[ i, j] = min 1+C {i, j – d  [i] ],    C [i – 1, j]}
return C [ k, A]
```

Comments 1.2.5

Comment 1: The above algorithm *explicitly* gives *only the number* of coins, which are minimum to make a pre-assigned amount A, yet it can also be used to *determine the set of coins* of various denominations that add upto A.

This can be seen through the following argument:

By definition, C [i, j] = either 1 + C [i, j – d_i] or C [i –1, j], which means either we choose a coin of denomination d_i or we do not choose a coin of denomination d_i,

depending upon whether $1 + C [i, j - d_i] \leq C [i - 1, j]$ or not. Applying the above rule recursively for decreasing values of i and j, we know which coins are chosen for making an amount j out of the available coins.

Comment 2: Once an algorithm is designed, it is important to know its computational complexity in order to determine its utility.

In order to determine C[k, A], the above algorithm computes $k \times (A + 1)$ values through additions and comparisons. Therefore, the complexity of the algorithm is $\theta (k . A)$, the order of the size of the matrix C[k, A].

1.3 THE PRINCIPLE OF OPTIMALITY

The Principle of Optimality states that components of a globally optimum solution must themselves be optimal. Or, Optimality Principle states subsolutions of an optimal solution must themselves be optimal.

While using the Dynamic Programming technique in solving the coin problem we tacitly assumed the above mentioned principle in defining C [i, j], as minimum of $1 + C [i, j - d_i]$ and $C [i - 1, j]$. The principle is used so frequently and without our being aware of having used it.

However, we must be aware that there may be situations in which the principle may not be applicable. The principle of optimality may **not** be true, specially, in situations where resources used over the (sub) components may be more than total resources used over the whole, and where total resources are limited and fixed. A simple example may explain the point. For example, suppose a 400-meter race champion takes 42 seconds to complete the race. However, covering 100 meters in 10.5 seconds may not be the best/optimal solution for 100 meters. The champion may take less than 10 seconds to cover 100 meters. The reason being total resources (here the concentration and energy of the athlete) are limited and fixed whether distributed over 100 meters or 400 meters.

Similarly, for a vehicle with best performance over 100 miles, can not be thought of in terms of 10 times best performance over 10 miles. First of all, fuel usage after some lower threshold, increases with speed. Therefore, as the distance to be covered increases (e.g., from 10 to 100 miles) then fuel has to be used more cautiously, restraining the speed, as compared to when distance to be covered is less (e.g., 10 miles). Even if refuelling is allowed, refuelling also takes time. The drivers concentration and energy are other fixed and limited resources; which in the case of shorter distance can be used more liberally as compared to over longer distances, and in the process produce better speed over short distances as compared to over long distances. The above discussion is for the purpose of driving attention to the fact that principle of optimality is ***not*** universal, specially when the resources are limited and fixed. Further, it is to draw attention that Dynamic Programming technique assumes validity of Principle of Optimality for the problem domain. Hence, while applying Dynamic Programming technique for solving optimisation problems, in order for the validity of the solution based on the technique, we need to ensure that the Optimality Principle is valid for the problem domain.

Ex. 1) Using Dynamic Programming, solve the following problem (well known as **Knapsack Problem**). Also write the algorithm that solves the problem.

We are given *n objects* and a knapsack. For i = 1, 2, …, n, object i has a positive *weight* w_i and a positive *value* v_i. The knapsack can carry a *weight not exceeding W*. Our aim is to fill the knapsack in a way that maximizes the value of the objects included in the knapsack. Further, it is assumed that the objects *may **not** be broken*

into pieces[*]. In other words, either a whole object is to be included or it has to be excluded.

1.4 CHAINED MATRIX MULTIPLICATION

In respect of multiplication of matrices, we recall the following facts:

(i) Matrix multiplication is a binary operation, i.e., at one time only two matrices can be multiplied.

(ii) However, not every given pair of matrices may be multiplicable. If we have to find out the product M_1M_2 then the orders of the matrices must be of the from $m \times n$ and $n \times p$ for some positive integers m, n and p. Then matrices M_1 and M_2, in this order, are said to be multiplication – compatible. Number of scalar (number) multiplications required is mnp.

As can be easily seen that matrix multiplication *is not commutative*. Even $M_2 M_1$ may not be defined even if $M_1 M_2$ is defined.

(iii) Matrix multiplication is associative in the sense that if M_1 M_2 and M_3 are three matrices of order $m \times n$, $n \times p$ and $p \times q$ then the matrices

$(M_1 M_2) M_3$ and $M_1 (M_2 M_3)$ are defined,
$(M_1 M_2) M_3 = M_1 (M_2 M_3)$
and the product is an $m \times n$ matrix.

(iv) Though, for three or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may very significantly depending upon how we pair the matrices and their product matrices to get the final product.

For example, If A is 14×6 matrix, B is 6×90 matrix, and C is 90×4 matrix, then the number of scalar multiplications required for (AB)C is

	$14 \times 6 \times 90 = 7560$	(*for (AB) which of order 14×90*)
Plus	$14 \times 90 \times 4 = 5040$	(*for product of (AB) with C*)

equal to 12600 scalar multiplication.

On the other hand, number of scalar multiplications for A(BC) is

	$6 \times 90 \times 4 = 2160$	(*for (BC) which is of order 6×4*)
Plus	$14 \times 6 \times 4 = 336$	(*for product of with BC*)

equal to 2496 scalar multiplication

Summering: The product of matrices A (14×6), B(6×90) and C(90×4) takes **12600** scalar operators when first the product of A and B is computed and then product AB is multiplied with C. On the other hand, if the product BC is calculated first and then product of A with matrix BC is taken then **only 2496** scalar multiplications are required. The later number is around 20% of the former. In case when large number of matrices are to be multiplied for which the product is defined, proper parenthesizing through pairing of matrices, may cause dramatic saving in number of scalar operations.

[*] Another version allows any fraction x_i with $0 \le x_i \le 1$. However, in this problem, we assume either $x_i = 1$ or $x_i = 0$.

This raises the question of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to optimize the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

Brute-Force Method: One way for finding the optimal method (i.e., the method which uses minimum number of scalar (numerical) operations) is to parenthesize the expression $A_1A_2 \ldots A_n$ in all possible ways and calculate number of scalar multiplications required for each way. Then choose the way which requires minimum number of scalar multiplications.

However, if **T(n)** denotes the number of ways of putting parentheses for pairing the expression $A_1A_2 \ldots A_n$, T(n) is an exponentially increasing function. The rate at which values of T(n) increase may be seen from the following table of values of T(n) for n = 1, 2, …….

n :	1	2	3	4	5	…	10	…	15
T(n):	1	1	2	5	14	…	4862	…	2674440

These values of T(n) are called *Catalan numbers*.

Hence, it is almost impossible to use the method for determining how to optimize the computation of the product $A_1A_2 \ldots A_n$.

Next, we discuss how to optimize the computation using Dynamic Programming Technique.

1.5 MATRIX MULTIPLICATION USING DYNAMIC PROGRAMMING

It can be seen that if one arrangement is optimal for $A_1A_2 \ldots A_n$ then it will be optimal for any pairings of $(A_1 \ldots A_k)$ and $(A_{k+1}A_n)$. Because, if there were a better pairing for say $A_1A_2 \ldots A_k$, then we can replace the better pair $A_1A_2 \ldots A_k$ in $A_1A_2 \ldots A_k\ A_{k+1} \ldots A_n$ to get a pairing better than the initially assumed optimal pairing, leading to a contradiction. Hence the principle of optimality is satisfied.

Thus, the Dynamic Programming technique can be applied to the problem, and is discussed below:

Let us first define the problem. Let A_i, $1 \le i \le n$, be a $d_{i-1} \times d_i$ matrix. Let the vector d [0.. n] stores the dimensions of the matrices, where the dimension of A_i is $d_{i-1} \times d_i$ for i = 1, 2, …, n. By definition, any subsequence $A_j \ldots A_k$ of $A_1A_2 \ldots A_n$ for $1 \le j \le k \le n$ is a well-defined product of matrices. Let us consider a table m [1.. n, 1 .. n] in which the entries m_{ij} for $1 \le i \le j \le n$, represent **optimal (i.e., minimum)** number of operations required to compute the product matrix $(A_i \ldots A_j)$.

We fill up the table diagonal-wise, i.e., in one iteration we fill-up the table one diagonal $m_{i,\ i+s}$, at a time, for some constant $s \ge 0$. Initially we consider the biggest diagonal m_{ii} for which **s = 0**. Then next the diagonal $m_{i,\ i+s}$ for **s = 1** and so on.

First, *filling up the entries m_{ii}, i = 1, 2, ..., n*.

Now m_{ii} stands for the minimum scalar multiplications required to compute the product of single matrix A_i. But number of scalar multiplications required are zero.

Hence,

$$m_{ii} = 0 \qquad \text{for } i = 1, 2, \ldots n.$$

Filling up entries for $m_{i(i+1)}$ for i = 1, 2, …, (n – 1).

$m_{i(i+1)}$ denotes the minimum number of scalar multiplication required to find the product $A_i A_{i+1}$. As A_i is $d_{i-1} \times d_i$ matrix and A_{i+1} is $d_i \times d_{i+1}$ matrix. Hence, there is a unique number for scalar multiplication for computing $A_i A_{i+1}$ giving

$$m_{i,(i+1)} = d_{i-1} d_i d_{i+1}$$

for i = 1, 2, …, (n – 1).

The above case is also subsumed by the general case

$$m_{i(i+s)} \qquad \text{for } s \geq 1$$

For the expression

$$A_i A_{i+1} \ldots A_{i+s,}$$

let us consider top-level pairing

$$(A_i A_{i+1} \ldots A_j)(A_{J+1} \ldots A_{i+s})$$

for some k with $i \leq j \leq i + s$.

Assuming optimal number of scalar multiplication viz., $m_{i,j}$ and $m_{i+1,j}$ are already known, we can say that

$$m_{i(i+s)} = \min_{i \leq j \leq i+s} (m_{i,j} + m_{j+1,s} + d_{i-1} d_j d_{i+s})$$

for i = 1, 2, …, n – s.

When the term $d_{i-1} d_j d_{i+s}$ represents the number of scalar multiplications required to multiply the resultant matrices $(A_i \ldots A_j)$ and $(A_{j+1} \ldots A_{i+s})$

Summing up the discussion, we come the definition $m_{i,i+s}$ for i=1,2, …(n –1) as

for s = 0: $m_{i,i} = 0$ for i = 1, 2, …, n

for s = 1: $m_{i,i+1} = d_{i-1} d_i d_{i+1}$ for i = 1, 2, …, (n–1)

for $1 < s < n$:

$$m_{i,i+s} = \min_{i \leq j \leq i+s} (m_{ij} + m_{j+1,i+s} + d_{i-1} d_i d_{i+1}) \qquad \text{for i= 1, 2, …, (n–s)}$$

Then $m_{1,n}$ is the final answer

Let us illustrate the algorithm to compute $m_{j+1,i+s}$ discussed above through an example

Let the given matrices be

A1	of order	14 × 6
A2	of order	6 × 90
A3	of order	90 × 4
A4	of order	4 × 35

Thus the dimension of vector d [0..4] is given by [14, 6, 90, 4, 35]

For s = 0, we know $m_{ii} = 0$. Thus we have the Matrix

Next, consider for s = 1, the entries

$$m_{i,i+1} = d_{i-1} d_i d_{i+1}$$

$m_{12} = d_0\ d_1\ d_2 = 14 \times 6 \times 90 = 7560$
$m_{23} = d_1\ d_2\ d_3 = 6 \times 90 \times 4 = 3240$
$m_{34} = d_2\ d_3\ d_4 = 9 \times 4 \times 35 = 1260$

	1	2	3	4
1	0	7560		
2		0	3240	
3			0	1260
4				0

Next, consider for s = 2, the entries

m_{13} = min $(m_{11} + m_{23} + 14 \times 6 \times 4,\ m_{12} + m_{33} + 14 \times 90 \times 4)$
= min $(0 + 3240 + 336,\ 7560 + 0 + 5040)$
= 3576

m_{24} = min $(m_{22} + m_{34} + 6 \times 90 \times 35,\ m_{23} + m_{44} + 6 \times 4 \times 35)$
= min $(0 + 1260 + 18900,\ 3240 + 0 + 840)$
= 4080

Finally, for s = 3

m_{14} = min $(m_{11} + m_{24} + 14 \times 6 \times 35,$ {when k = 1}
$M_{12} + m_{34} + 14 \times 90 \times 35,$ {when k = 2}
$M_{13} + m_{44} + 14 \times 4 \times 35)$ {when k = 3}

= min $(0 + 4080 + 2090,\ 7560 + 3240 + 44100,\ 3576 + 0 + 1960)$
= 5536

Hence, the optimal number scalar multiplication, is 5536.

1.6 SUMMARY

(1) The Dynamic Programming is a technique for solving optimization Problems, using bottom-up approach. The underlying idea of dynamic programming is to avoid calculating the same thing twice, usually by keeping a table of known results, that fills up as substances of the problem under consideration, are solved.

(2) In order that Dynamic Programming technique is applicable in solving an optimisation problem, it is necessary that the principle of optimality is applicable to the problem domain.

(3) The principle of optimality states that for an optimal sequence of decisions or choices, each subsequence of decisions/choices must also be optimal.

(4) **The Chain Matrix Multiplication Problem:** The problem of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to minimize the number of scalar multiplications in the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

(5) **The Knapsack Problem:** We are given *n objects* and a knapsack. For i = 1, 2, …, n, object i has a positive *weight* w_i and a positive *value* v_i. The knapsack can carry a *weight not exceeding W*. The problem requires that the knapsack is filled in a way that maximizes the value of the objects included in the knapsack.

Further, a special case of knapsack problem may be obtained in which the objects *may **not** be broken into pieces*[*]. In other words, either a whole object is to be included or it has to be excluded.

1.7 SOLUTIONS/ANSWERS

Ex. 1)

First of all, it can be easily verified that Principle of Optimality is valid in the sense for an optimal solution of the overall problem each subsolution is also optimal. Because in this case, a non-optimal solution of a subproblem, when replaced by a better solution of the subproblem, would lead to a better than optimal solution of the overall problem. A contradiction.

As usual, for solving an optimization problem using Dynamic Programming technique, we set up a table V[1..n, 0..W].

In order to label the rows we first of all, order the given objects according to increasing relative values R = v/w.

Thus first object O_1 is the one with minimum relative value R_1. The object O_2 is the one with next least relative value R_2 and so on. The last object, in the sequence, is O_n, with maximum relative weight R_n.

The **ith row** of the Table Corresponds to object O_i having ith relative value, when values are arranged in increasing order. The jth column corresponds to weight j for $0 \leq j \leq W$. The entry **Knap [i, j]** denotes the maximum value that can be packed in knapsack when objects $O_1, O_2, \ldots, O_i$ *only* are used and the included objects have weight at most j.

Next, in order to fill up the entries Knap[i, j], $1 \leq i \leq n$ and $0 \leq j \leq W$, of the table, we can check, as was done in the coin problem that,

(i) Knap [i, 0] = 0 for i = 1, 2, …, n

(ii) Knap [1, j] = V, for j = 1, …, W
where V is the value of O_1

[*] Another version allows any fraction x_i with $0 \leq x_i \leq 1$. However, in this problem, we assume either $x_i = 1$ or $x_i = 0$.

	0	1	2	3	4	…	j	…	W
1	0	V	V	V	V		V		V
2	0								
.	.								
.	.								
.	.								
j	0								
.	.								
.	.								
.	.								
n	0								

Where V is the value of the object O_1.

Further, when calculating Knap [i, j], there are two possibilities: either

(i) The ith object O_i is taken into consideration, then
Knap [i, j] = Knap [i–1, j – w_i] + v_i or

(ii) The ith object O_i is not taken into consideration then
Knap [i, j] = V[i–1, j]

Thus we define

Knap [i, j] = max {Knap [i–1, j], Knap [i–1, j - w_i] + v_i}

The above equation is valid for i = 2 and j ≥ w_i. In order that the above equation may be applicable otherwise also, without violating the intended meaning we take,

(i) Knap [O, j] = O for j ≥ O and
(ii) Knap [i, j] = – ∞ for k < 0

We explain the Dynamic Programming based solution suggested above, through the following example.

We are given six objects, whose weights are respectively 1, 2, 5, 6, 7, 10 units and whose values respectively are 1, 6, 18, 22, 28, 43. Thus relative values in increasing order are respectively 1.00, 3.00, 3.60, 3.67, 4.00 and 4.30. **If we can carry a maximum weight of 12, then the table below shows that we can compose a load whose value is 49.**

Weights	0	1	2	3	4	5	6	7	8	9	10	11	12
Relative Values													
w_1= 1, v_1 = 1, R_1 = 1.00	0	1	1	1	1	1	1	1	1	1	1	1	1
w_2 = 2, v_2 = 6, R_2 = 3.00	0	1	6	7	7	7	7	7	7	7	7	7	7
w_3 = 5, v_3 = 18, R_3 = 3.60	0	1	6	7	7	18	19	24	25	25	25	25	25
W_4 = 6, v_4 = 22, R_4 =3.67	0	1	6	7	7	18	22	24	28	29	29	40	41
W_5 = 7, v_5 = 28 R_5 = 4.00	0	1	6	7	7	18	22	28	29	34	35	40	46
W_6 = 10, v_6 = 43, R_6 = 4.30	0	1	6	7	7	18	22	28	29	34	43	44	49

Algorithm for the solution of the solution explained above of the Knapsack Problem.

Function Knapsack (W, Weight [1..n], value [1..n])

{returns the maximum value corresponding to maximum allowed weight W, where we are given n objects O_i, $1 \le i \le n$, with weights Weight [i] and values Value [i]}
array R [1 ..n], Knap [1..n, 0..W]
For i = 1 to n do

begin
read (Weight [i])
read (value [i])
R [i] / weight [i]
end

For j = 1 to n do

{Find k such that R[k] is minimum of
R[j], R[j+1], …R[n]}

Begin
k = j
For t = j + 1 to n do
If R [t] < R [k] then
k = t
Exchange (R [j], R [k]);
Exchange (Weight [j], Weight [k]);
Exchange (Value [j], value [k]);
end

{At this stage R[1.. n] is a sorted array in increasing order and Weight [j] and value [j] are respectively weight and value for jth least relative value}
{Next, we complete the table knap for the problem}

For i = 1 to n do
Knap [i , 0] = 0 for i = 1, …, n
Knap [1, j] = value [1] for j = 1, …, W

{Value [1] is the value of the object with minimum relative value}

{Next values for out of the range of the table Knap}

If $i \le 0$ and $j \ge 0$ then
Knap [i , j] = 0
Else if $j < 0$ – then
Knap [i, j] = $-\infty$

For i = 2 to n do
For j = 1 to W do
Knap [i, j] = maximum {Knap [i –1, j], Knap [i – 1, j – Weight [i]]
+ value [i]

Retu rn Knap [n, W]

1.8 FURTHER READINGS

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

UNIT 2 GREEDY TECHNIQUES

Structure **Page Nos.**

2.0 INTRODUCTION

Algorithms based on Greedy technique are used for solving **optimization problems**. **An optimization problem** is one in which some value (or set of values) of interest is required to be either minimized or maximized w.r.t some given relation on the values. Such problems include maximizing profits or minimizing costs of, say, production of some goods. Other examples of optimization problems are about

- finding the *minimum number* of currency notes required for an amount, say of Rs. 289, where arbitrary number of currency notes of each denomination from Rs. 1 to Rs. 100 are available, and
- finding shortest path covering a number of given cities where distances between pair of cities are given.

As we will study later, the algorithms based on greedy technique, *if exist*, are easy to think of, implement and explain about. However, for many interesting optimization problems, no algorithm based on greedy technique may yield optimal solution. In support of this claim, let us consider the following example:

Example 1.1: Let us suppose that we have to go from city A to city E through either city B or city C or city D with costs of reaching between pairs of cities as shown below:

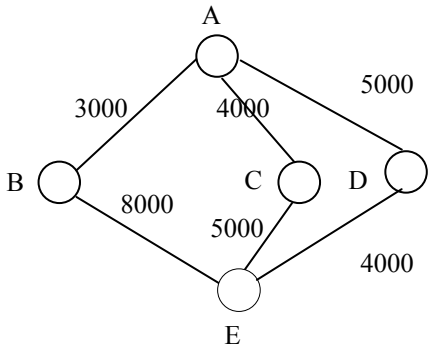


Figure 2.0.1

Then the greedy technique suggests that we take the route from A to B, the cost of which Rs.3000, is the minimum among the three costs, (viz., Rs. 3000, Rs. 4000 and Rs. 5000) of available routes.

However, at B there is only one route available to reach E. Thus, greedy algorithm suggests the route from A to B to E, which costs Rs.11000. But, the route from A to C to E, costs only Rs.9000. Also, the route from A to D to E costs also Rs.9000. Thus, locally better solution, at some stage, suggested by greedy technique yields overall (or globally) costly solution.

The essence of Greedy Technique is : In the process of solving an optimization problem, initially and at subsequent stages, we evaluate the costs/benefits of the various available alternatives for the next step. Choose the alternative which is *optimal* in the sense that either it is the least costly or it is the maximum profit yielding. In this context, it may be noted that the *overall solution*, yielded by choosing *locally optimal steps*, may *not* be optimal.

2.1 OBJECTIVES

After studying unit, you should be able to:

- explain the Greedy technique for solving optimization problems;
- apply the Greedy technique for solving optimization problems;
- apply Greedy technique for solving well-known problems including shortest path problem.

2.2 SOME EXAMPLES

In order to set the subject-matter to be discussed in proper context, we recall the general characteristics of greedy algorithms. These algorithms are:

- used to solve optimization problems,
- most straight forward to write,
- easy to invent, easy to implement, and if exist, are efficient,
- may not yield a solution for an arbitrary solvable problem,
- short-sighted making decisions on the basis of information immediately on hand, without worrying about the effect these decisions may have in future, and
- never reconsider their decisions.

In order to understand the salient features of the algorithms based on greedy technique, let us consider some examples in respect of the following **Minimum Number of Notes Problem:**

In a business transaction, we are required to make payment of some amount A (say Rs.289/-). We are given Indian currency notes of all denominations, viz., of 1,2,5,10,20,50,100, 500 and 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.

Example 2.2.1

In this example, we discuss, how intuitively we attempt to solve the Minimum Number of Notes Problem , to be specific, to make an amount of Rs.289/-.

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

i) $D \leq 289$ and

ii) *if* D_1 is another denomination of a note such that $D_1 \leq 289$, *then* $D_1 \leq D$.

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 2.2.2

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 2.2.3

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.

Ex.1) Give another example in which greedy technique fails to deliver an optimal solution.

2.3 FORMALIZATION OF GREEDY TECHNIQUE

In order to develop an *algorithm* based on *the greedy technique* to solve a *general optimization problem*, we need the following data structures and functions:

(i) **A set or list of given/candidate values** from which choices are made, to reach a solution. For example, in the case of Minimum Number of Notes problem, the list of candidate values (in rupees) of notes is {1, 2, 5, 10, 20, 50, 100, 500, 1000}. Further, the number of notes of each denomination should be clearly

mentioned. Otherwise, it is assumed that each candidate value can be used as many times as required for the solution using greedy technique. Let us call this set as

GV: Set of Given Values

(ii) **Set (*rather multi-set*) of considered and chosen values:** This structure contains those candidate values, which are considered and chosen by the algorithm based on greedy technique to reach a solution. Let us call this structure as

CV: Structure of Chosen Values

The structure is generally not a set but a multi-set in the sense that values may be repeated. For example, in the case of Minimum Number of Notes problem, if the amount to be collected is Rs. 289 then

CV = {100, 100, 50, 20, 10, 5, 2, 2}

(iii) **Set of Considered and Rejected Values:** As the name suggests, this is the set of all those values, which are considered but rejected. Let us call this set as

RV: Set of considered and Rejected Values

A candidate value may belong to both CV and RV. But, once a value is put in RV, then this value can not be put any more in CV. For example, to make an amount of Rs. 289, once we have chosen two notes each of denomination 100, we have

CV = {100, 100}

At this stage, we have collected Rs. 200 out of the required Rs. 289. At this stage RV = {1000, 500}. So, we can chose a note of any denomination except those in RV, i.e., except 1000 and 500. Thus, at this stage, we can chose a note of denomination 100. However, this choice of 100 again will make the total amount collected so far, as Rs. 300, which exceeds Rs. 289. Hence we reject the choice of 100 third time and put 100 in RV, so that now RV = {1000, 500, 100}. From this point onward, we can not chose even denomination 100.

Next, we consider some of the functions, which need to be defined in an algorithm using greedy technique to solve an optimization problem.

(iv) **A function say *SolF*** that checks whether a solution is reached or not. However, the function does not check for the *optimality* of the obtained solution. In the case of Minimum Number of Notes problem, the function SolF finds the sum of all values in the multi-set CV and compares with the desired amount, say Rs. 289. For example, if at one stage CV = {100, 100} then sum of values in CV is 200 which does not equal 289, then the function SolF returns '*Solution not reached*'. However, at a later stage, when CV = {100, 100, 50, 20, 10, 5, 2, 2}, then as the sum of values in CV equals the required amount, hence the function SolF returns the message of the form '*Solution reached*'.

It may be noted that the function only informs about a possible solution. However, solution provided through SolF may not be *optimal*. For instance in the Example 2.2.3, when we reach CV = {60, 10, 10}, then SolF returns '*Solution reached*'. However, as discussed earlier, the solution 80 = 60 + 10 + 10 using three notes is not optimal, because, another solution using only two notes, viz., 80 = 40 + 40, is still cheaper.

(v) **Selection Function** say ***SelF*** finds out the most promising candidate value out of the values not yet rejected, i.e., which are not in RV. In the case of Minimum Number of Notes problem, for collecting Rs. 289, at the stage when RV = {1000, 500} and CV = {100, 100} then first the function SelF attempts the denomination 100. But, through function SolF, when it is found that by

addition of 100 to the values already in CV, the total value becomes 300 which exceeds 289, the value 100 is rejected and put in RV. Next, the function SelF attempts the next lower denomination 50. The value 50 when added to the sum of values in CV gives 250, which is less than 289. Hence, the value 50 is returned by the function SelF.

(vi) **The Feasibility-Test Function,** say **FeaF.** When a new value say v is chosen by the function SelF, then the function FeaF checks whether the new set, obtained by adding v to the set CV of already selected values, is a possible part of the final solution. Thus in the case of Minimum Number of Notes problem, if amount to be collected is Rs. 289 and at some stage, CV = {100, 100}, then the function SelF returns 50. At this stage, the function FeaF takes the control. It adds 50 to the sum of the values in CV, and on finding that the sum 250 is less than the required value 289 informs the main/calling program that {100, 100, 50} can be a part of some final solution, and needs to be explored further.

(vii) **The Objective Function,** say **ObjF,** gives the value of the solution. For example, in the case of the problem of collecting Rs. 289; as CV = {100, 100, 50, 20, 10, 5, 2, 2} is such that sum of values in CV equals the required value 289, the function ObjF **returns the number** of notes in CV, i.e., the number 8.

After having introduced a number of sets and functions that may be required by an algorithm based on greedy technique, we give below the outline of greedy technique, say **Greedy-Structure.** For any *actual* algorithm based on greedy technique, the various *structures* the functions discussed above have to be replaced by *actual* functions.

These functions depend upon the problem under consideration. The Greedy-Structure outlined below takes the set GV of given values as input parameter and returns CV, the set of chosen values. For developing any algorithm based on greedy technique, the following function outline will be used.

2.3.1 Function Greedy-Structure (GV:set): set

```
    CV ← ϕ  {initially, the set of considered values is empty}
While GV ≠ RV and not SolF (CV) do
    begin
        v ← SelF (GV)
        If FeaF (CV ∪ {v})  then
            CV  ←  CV ∪ {v}
        else  RV ← RV ∪ {v}
    end

        // the function Greedy Structure comes out
    // of while-loop when either GV=RV, i.e., all
// given values are rejected or when solution is found

        If  SolF (CV) then returns ObjF (GV)
          else return "No solution is possible"
    end function Greedy–Structure
```

2.4 MINIMUM SPANNING TREE

In this section and some of the next sections, we apply greedy technique to develop algorithms to solve some well-known problems. First of all, we discuss the applications for finding minimum spanning tree for a given (undirected) graph. In order to introduce the subject matter, let us consider some of the relevant definitions.

Definitions

A ***Spanning tree*** of a connected graph, say G = (V, E) with V as set of vertices and E as set of edges, is its **connected** acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.

A ***minimum spanning tree*** of a *weighted* connected graph is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges.

The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

The minimum spanning tree problem has a number of useful applications in the following type of situations:

Suppose, we are given a set of cities alongwith the distances between each pair of cities. In view of the shortage of funds, it is desired that in stead of connecting directly each pair of cities, we provide roads, costing least, but allowing passage between any pair cities along the provided roads. However, the road between some pair of cities may not be direct, but may be passing through a number of other cities.

Next, we illustrate the concept of spanning tree and minimum spanning tree through the following example.

Let us consider the connected weighted graph G given in *Figure 4.1*.

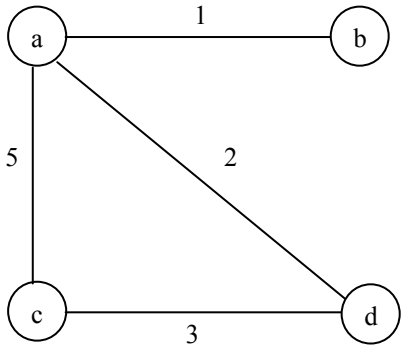


Figure: 2.4.1

For the graph of *Figure 2.4.1*given above, each of *Figure 2.4.2, Figure, 2. 4.3* and *Figure. 2.4.4* shows a spanning tree viz., T_1, T_2 and T_3 respectively.
Out of these, T_1 is a minimal spanning tree of G, of weight 1+2+3 = 6.

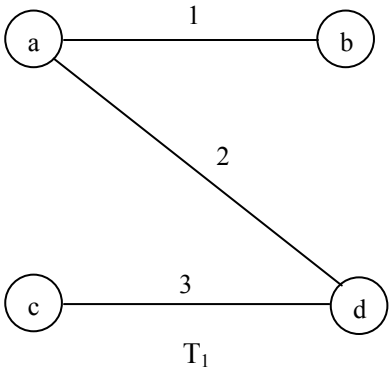
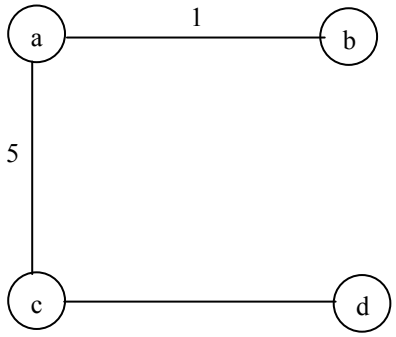
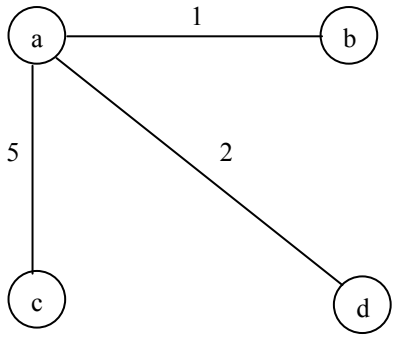


Figure: 2.4.2



T_2

Figure: 2.4.3



T_3

Figure: 2.4.4

Remark 2.4.1:

The weight may denote (i) length of an edge between pair of vertices or (ii) the cost of reaching from one town to the other or (iii) the cost of production incurred in reaching from one stage of production to the immediate next stage of production or (iv) the cost of construction of a part of a road or of laying telephone lines between a pair of towns.

Remark 2.4.2:

The weights on edges are generally positive. However, in some situations the weight of an edge may be zero or even negative. The negative weight *may appear* to be appropriate when the problem under consideration is not about '*minimizing costs*' but about '*maximizing profits'* and we still want to use *minimum spanning tree algorithms*. However, in such cases, it is not appropriate to use negative weights, because, more we traverse the negative-weight edge, lesser the cost. However, with repeated traversals of edges, the cost should increase in stead of decreasing.

Therefore, if we want to apply minimum spanning tree technique to 'maximizing profit problems, then in stead of using negative weights, we replace profits p_i by $M-p_i$ where M is some positive number s.t

$M > \text{Max}\ \{p_{ij} : p_{ij}$ is the profit in traversing from ith node to jth node$\}$

Remark 2.4.3:

From graph theory, we know that a given connected graph with n vertices, must have exactly (n –1) edges.

As mentioned earlier, whenever we want to develop an algorithm based on greedy technique, we use the function Greedy-Structure given under 2.3.1. For this purpose,

we need to find appropriate values of the various sets and functions discussed in Section 3.

In the case of **the problem of finding minimum-spanning tree for a given connected graph,** the appropriate values are as follows:

(i) **GV: The set of candidate or given values** is given by GV = E, the set of edges of the given graph (V, E).

(ii) **CV: The structure of chosen values** is given by those edges from E, which together will form the required minimum-weight spanning tree.

(iii) **RV: set of rejected values** will be given by those edges in E, which at some stage will form a cycle with earlier selected edges.

(iv) **In the case of the problem of minimum spanning tree, the function SolF that checks whether a solution** is reached or not, is the function that checks that

(a) all the edges in CV form a tree,
(b) the set of vertices of the edges in CV equal V, the set of all edges in the graph, and
(c) the sum of the weights of the edges in CV is minimum possible of the edges which satisfy (a) and (b) above.

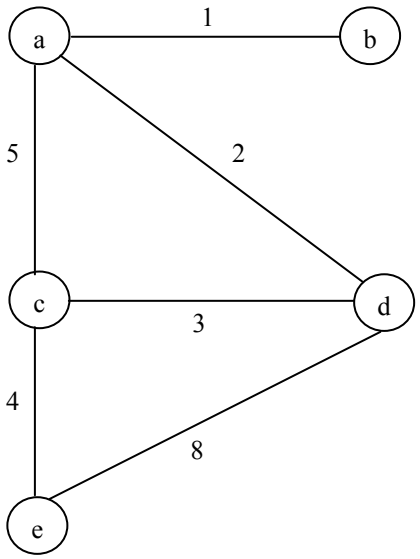
(v) **Selection Function:** depends upon the particular algorithm used for the purpose. There are two well-known algorithms, viz., Prim's algorithm and Kruskal's algorithm for finding the Minimum Spanning Tree. We will discuss these algorithms in detail in subsequent sections.

(vi) **FeaF: Feasibility Test Function:** In this case, when the selection function SelF returns an edge depending on the algorithm, the feasibility test function FeaF will check whether the newly found edge forms a cycle with the earlier slected edges. If the new edge actually forms a cycle then generally the newly found edge is dropped and search for still another edge starts. However, in some of the algorithms, it may happen that some earlier chosen edge is dropped.

(vii) In the case of Minimum Spanning Tree problem, the objective function may return

(a) the set of edges that constitute the required minimum spanning tree and
(b) the weight of the tree selected in (a) above.

Ex. 2) Find a minimal spanning tree for the following graph.



2.5 PRIM'S ALGORITHM

The algorithm due to **Prim** builds up a minimum spanning tree by adding edges to form a sequence of expanding subtrees. The sequence of subtrees is represented by the pair (V_T, E_T), where V_T and E_T respectively represent the set of vertices and the set of edges of a subtree in the sequence. Initially, the subtree, in the sequence, consists of just a single vertex which is selected arbitrarily from the set *V* of vertices of the given graph. The subtree is built-up iteratively by adding an edge that has minimum weight among the remaining edges (i.e., edge selected greedily) and, which at the same time, does not form a cycle with the earlier selected edges.

We illustrate the Prim's algorithm through an example before giving a semi-formal definition of the algorithm.

Example 2.5.1 (of Prim's Algorithm):

Let us explain through the following example how Prim's algorithm finds a minimal spanning tree of a given graph. Let us consider the following graph:

Initially

$V_T = (a)$

$E_T = \phi$

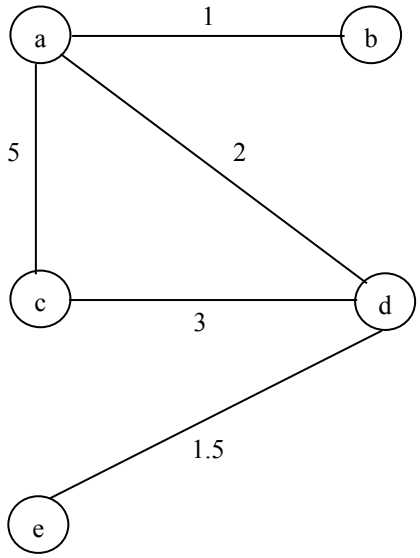


Figure: 2.5.1

In the first iteration, the edge having weight which is the minimum of the weights of the edges having **a** as one of its vertices, is chosen. In this case, the edge ab with weight 1 is chosen out of the edges **ab**, ac and ad of weights respectively 1,5 and 2. Thus, after First iteration, we have the **given graph with chosen edges in bold and V_T and E_T as follows:**

$V_T = (a, b)$

$E_T = (\ (a,b))$

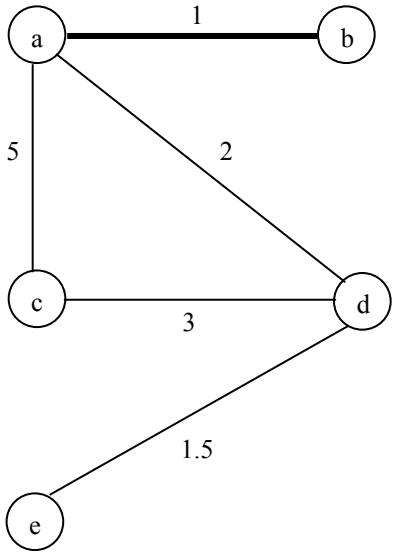


Figure: 2.5.2

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edge and having either a or b as one of its vertices, the edge with minimum weight is chosen. In this case the vertex b does not have any edge originating out of it. In such cases, if required, weight of a non-existent edge may be taken as ∞. Thus choice is restricted to two edges viz., ad and ac respectively of weights 2 and 5. Hence, in the next iteration the edge ad is chosen. **Hence, after second iteration, we have the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d)
E_T = ((a, b), (a, d))

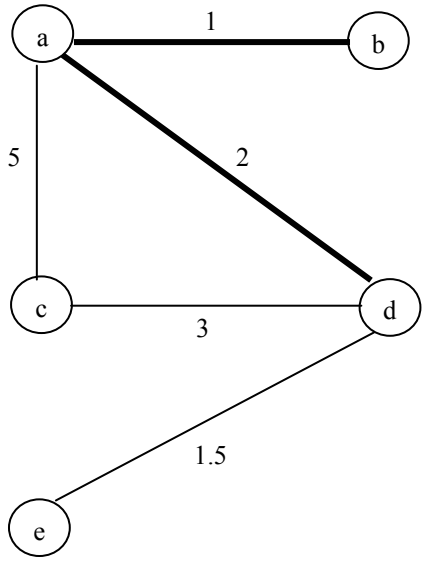


Figure: 2.5.3

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edges and having either a, b or d as one of its vertices, the edge with minimum weight is chosen. Thus choice is restricted to edges ac, dc and de with weights respectively 5, 3, 1.5. The edge de with weight 1.5 is selected. **Hence, after third iteration we have the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d, e)
E_T = ((a, b), (a, d), (d, e))

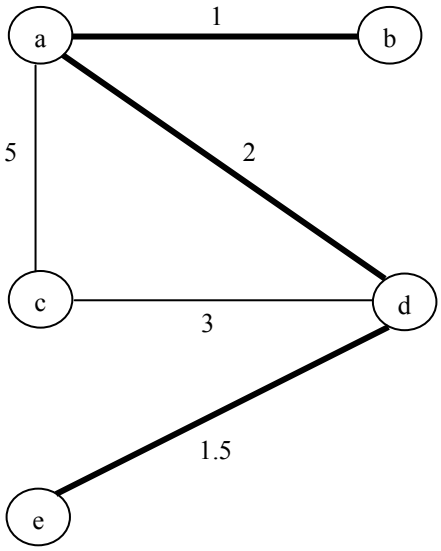


Figure: 2.5.4

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edge and having either a, b, d or e as one of its vertices, the edge with minimum weight is chosen. Thus, choice is restricted to edges dc and ac with weights respectively 3 and 5. Hence the edge dc with weight 3 is chosen. Thus, after fourth iteration, we have **the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d, e, c)
E_T = ((a b), (a d) (d e) (d c))

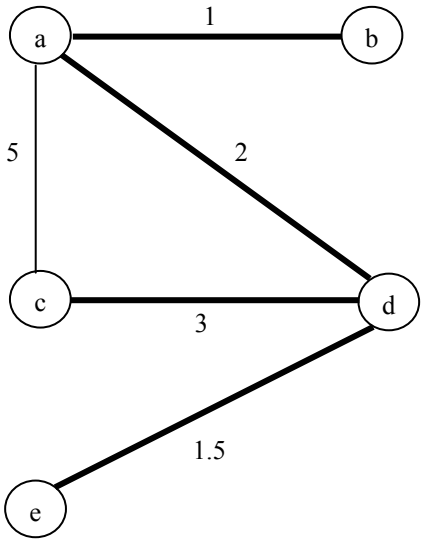


Figure: 2.5.5

At this stage, it can be easily seen that each of the vertices, is on some chosen edge and the chosen edges form a tree.

Given below is the semiformal definition of Prim's Algorithm

Algorithm Spanning-Prim (G)

// the algorithm constructs a minimum spanning tree
// for which the input is a weighted connected graph $G = (V, E)$
// the output is the set of edges, to be denoted by E_T, which together constitute a minimum spanning tree of the given graph G
// for the pair of vertices that are not adjacent in the graph to each other, can be given
// the label ∞ indicating "infinite" distance between the pair of vertices.
// the set of vertices of the required tree is initialized with the vertex v_0
$V_T \leftarrow \{v_O\}$
$E_T \leftarrow \phi$ // initially E_T is empty
// let n = number of vertices in V

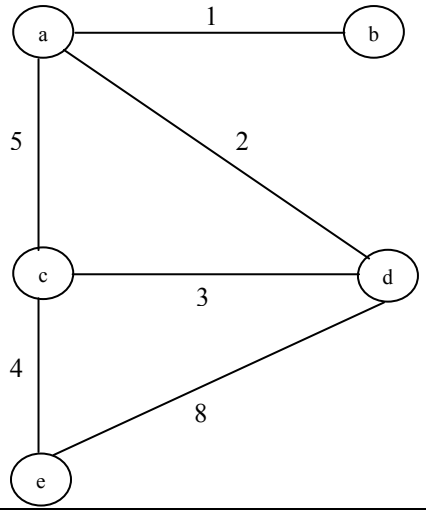
For i = 1 **to** |n| - 1 **do**

find a minimum-weight edge $\underline{e} = (v^1, u^1)$ among all the edges such that v^1 is in V_T and u^1 is in $V - V_T$.

$V_T \leftarrow V_T \cup \{u^1\}$
$E_T = E_T \cup \{\underline{e}\}$

Return E_T

Ex. 3) Using Prim's algorithm, find a minimal spanning tree for the graph given below:



2.6 KRUSKAL'S ALGORITHM

Next, we discuss another method, of finding minimal spanning tree of a given weighted graph, which is suggested by Kruskal. In this method, the emphasis is on the choice of edges of minimum weight from amongst all the available edges, of course, subject to the condition that chosen edges do not form a cycle.

The connectivity of the chosen edges, at any stage, in the form of a subtree, which was emphasized in Prim's algorithm, is **not** essential.

We briefly describe the Kruskal's algorithm to find minimal spanning tree of a given weighted and connected graph, as follows:

(i) First of all, order all the weights of the edges in increasing order. Then repeat the following two steps till a set of edges is selected containing all the vertices of the given graph.

(ii) Choose an edge having the weight which is the minimum of the weights of the edges not selected so far.

(iii) If the new edge forms a cycle with any subset of the earlier selected edges, then drop it, *else,* add the edge to the set of selected edges.

We illustrate the Kruskal's algorithm through the following:

Example 2.6.1:

Let us consider the following graph, for which the minimal spanning tree is required.

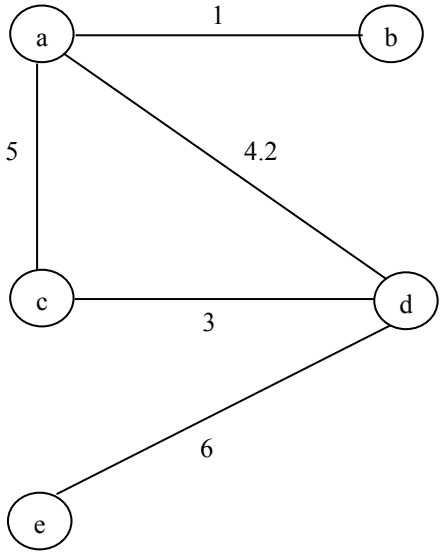


Figure: 2.6.1

Let E_g denote the set of edges of the graph that are chosen upto some stage.

According to the step (i) above, the weights of the edges are arranged in increasing order as the set

{1, 3, 4.2 ,5, 6}

In the first iteration, the edge (a,b) is chosen which is of weight 1, the minimum of all the weights of the edges of the graph.

As single edge do not form a cycle, therefore, the edge (a,b) is selected, so that

E_g = ((a,b))

After first iteration, the graph with selected edges in bold is as shown below:

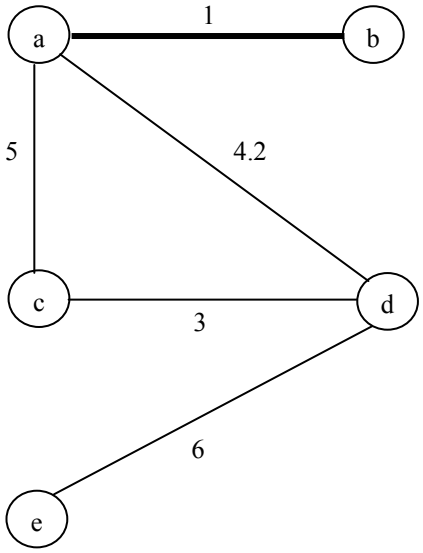


Figure: 2.6.2

Second Iteration

Next the edge (c,d) is of weight 3, minimum for the remaining edges. Also edges (a,b) and (c,d) do not form a cycle, as shown below. Therefore, (c,d) is selected so that,

$E_g = ((a,b), (c,d))$

Thus, after second iteration, the graph with selected edges in bold is as shown below:

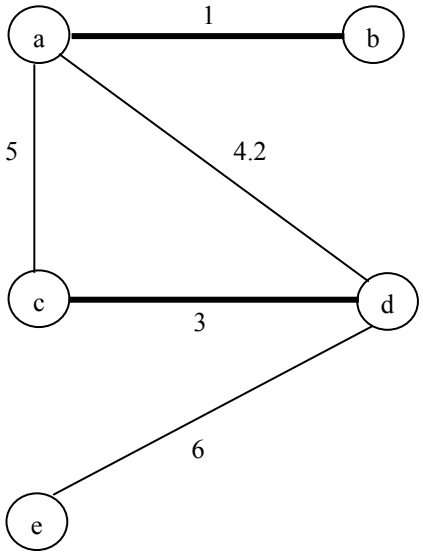


Figure: 2.6.3

It may be observed that the selected edges do not form a connected subgraph or subtree of the given graph.

Third Iteration

Next, the edge (a,d) is of weight 4.2, the minimum for the remaining edges. Also the edges in E_g alongwith the edge (a,d) do not form a cycle. Therefore, (a,d) is selected so that new $E_g = ((a,b), (c,d), (a,d))$. Thus after third iteration, the graph with selected edges in bold is as shown below.

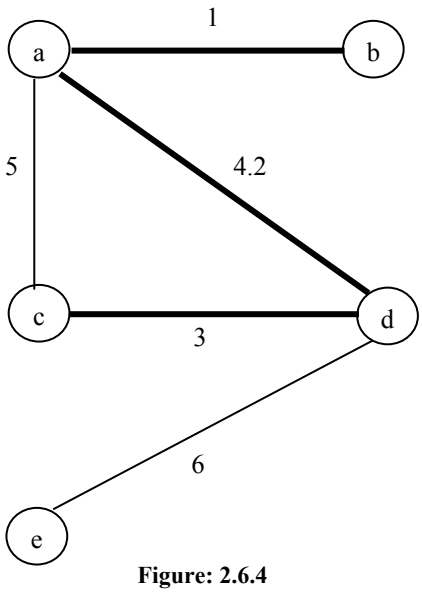


Figure: 2.6.4

Fourth Iteration

Next, the edge (a,c) is of weight 5, the minimum for the remaining edge. However, the edge (a,c) forms a cycles with two edges in E_g, viz., (a,d) and (c,d). Hence (a,c) is not selected and hence not considered as a part of the to-be-found spanning tree.

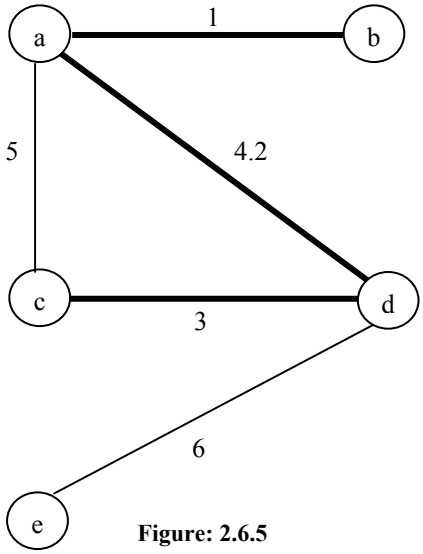


Figure: 2.6.5

At the end of fourth iteration, the graph with selected edges in bold remains the same as at the end of the third iteration, as shown below:

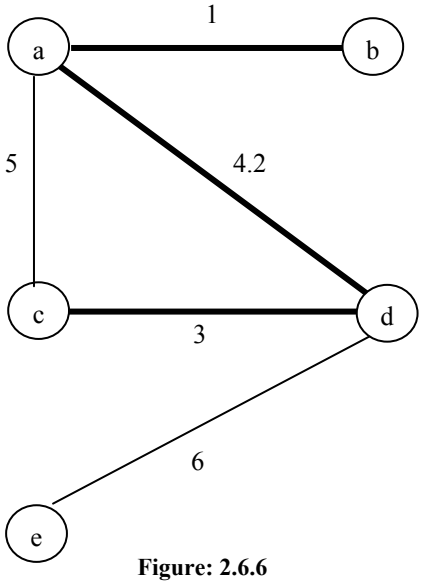


Figure: 2.6.6

Fifth Iteration

Next, the edge (e,d), the only remaining edge that can be considered, is considered. As (e,d) does not form a cycle with any of the edges in E_g. Hence the edge (e,d) is put in E_g. The graph at this stage, with selected edge in bold is as follows.

Error!

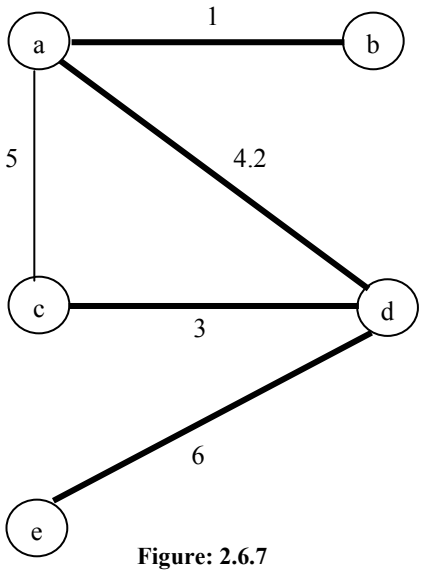


Figure: 2.6.7

At this stage we find each of the vertices of the given graph is a vertex of some edge in E_g. Further we observe that the edges in E_g form a tree, and hence, form the required spanning tree. Also, from the choice of the edges in E_g, it is clear that the spanning tree is of minimum weight. Next, we consider semi-formal definition of Kruskal's algorithm.

ALGORITHM Spanning-Kruskal (*G*)

// The algorithm constructs a minimum spanning tree by choosing successively edges
// of minimum weights out of the remaining edges.
// The input to the algorithm is a connected graph G = (V, E), in which V is the set of
// vertices and E, the set of edges and weight of each edge is also given.
// The output is the set of edges, denoted by E_T, which constitutes a minimum
// spanning tree of *G*
// the variable *edge-counter* is used to count the number of selected edges so far.
// variable t is used to count the number of edges considered so far.

Arrange the edges in *E* in nondecreasing order of the weights *of edges.* After the arrangement, the edges in order are labeled as $e_1, e_2, \ldots e_{|E|}$
$E_T \leftarrow \phi$ // initialize the set of tree edges as empty
edge-counter $\leftarrow 0$ // initialize the ecounter to zero
$t \leftarrow 0$ // initialize the number of processed edges as zero
// let n = number of edges in V

While edge-*counter* $< n-1$
    $t \leftarrow t + 1$ // increment the counter for number of edges considered so far
If *the edges* e_t *does not form a cycle with any subset of edges in* E_T **then**
    begin
    // if, e_t alongwith edges earlier in E_T do not form a cycle
    // then add e_t to E_T and increase edge counter
    $E_T \leftarrow E_T \cup \{ e_t \}$;
    edge-counter $\leftarrow$ edge-counter + 1
end if
return E_T

Summary of Kruskal's Algorithm

- The algorithm is always successful in finding a minimal spanning tree
- (Sub) tree structure may **not** be maintained, however, finally, we get a minimal spanning tree

Computing Complexity of Kruskals's

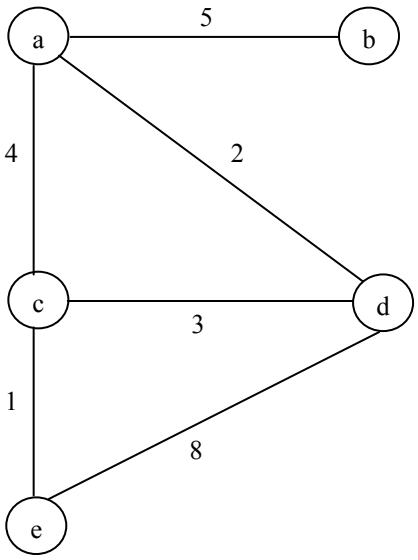
Let **a** be the number of edges and **n** be the number of nodes, initially given.
Then

(i) θ **(a log a)** time is required for sorting the edges in increasing orders of lengths

(ii) An efficient Union-Find operation takes (**2a**) find operations and (**n –1**) merge operations.

Thus Complexity of Kruskal's algorithm is O(**a log a**)

Ex. 4) Using Kruskal's algorithm, find a minimal spanning tree for the following g graph



2.7 DIJKSTRA'S ALGORITHM

Directed Graph:

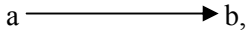
So far we have discussed applications of Greedy technique to solve problems involving undirected graphs in which each edge (a, b) from a to b is also equally an edge from b to a. In other words, the two representations (a,b) and (b,a) are for the same edge. Undirected graphs represent symmetrical relations. For example, the relation of 'brother' between male members of, say a city, is symmetric. However, in the same set, the relation of 'father' is not symmetric. Thus a general relation may be symmetric or asymmetric. A general relation is represented by a **directed** graph, in which the (directed) edge, also called an arc, (a, b) denotes an edge from a to b. However, the directed edge (a,b) is not the same as the directed edge (b,a). In the context of directed graphs, (b,a) denotes the edge from b to a. Next, we formally define a directed graph and then solve some problems, using Greedy technique, involving directed graphs.

Actually, the notation (a,b) in mathematics is used for ordered pair of the two elements viz., a and b in which a comes first and then b follows. And the ordered pair (b,a) denotes a different ordered set in which b comes first and then a follows.

However, we have misused the notation in the sense that we used the notation (a,b) to denote an unordered set of two elements, i.e., a set in which order of occurrence of a and b does not matter. In Mathematics the usual notation for an unordered set is {a,b}. In this section, we use parentheses (i.e., (and)) to denote ordered sets and braces (i.e., {and}) to denote a geneal (i.e., unordered set).

Definition:

A **directed graph or digraph** G = (V(G), E(G)) where V(G) denotes the set of vertices of G and E(G) the set of directed edges, also called arcs, of G. An arc from a to b is denoted as (a, b). Graphically it is denoted as follows:



in which the arrow indicates the direction. In the above case, the vertex a is sometimes called the **tail** and the vertex b is called the head of the arc or directed edge.

Definition:

A **Weighted Directed Graph** is a directed graph in which each arc has an assigned weight. A weighted directed graph may be denoted as G = (V(G), E(G)), where any element of E(G) may be of the form (a,b,w) where w denotes the weight of the arc (a,b). The directed Graph G = ((a, b, c, d, e), ((b, a, 3), (b, d, 2) (a, d,7), (c, b, 4), (c, d, 5), (d, e, 4), (e, c, 6))) is diagrammatically represented as follows:

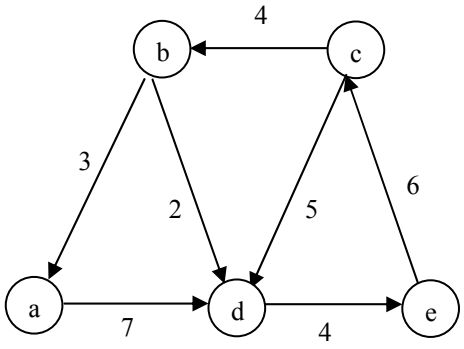


Figure: 2.7.1

Single-Source Shortest Path

Next, we consider the problem of finding the shortest distances of each of the vertices of a given weighted connected graph from some *fixed vertex* of the given graph. All the weights between pairs of vertices are taken as only positive number. The fixed vertex is called the source. The problem is known as **Single-Source Shortest Path Problem (SSSPP)**. One of the well-known algorithms for SSSPP is due to Dijkstra. The algorithm proceeds iteratively, by first consider the vertex nearest to the source. Then the algorithm considers the next nearest vertex to the source and so on. Except for the first vertex and the source, the distances of all vertices are iteratively adjusted, taking into consideration the new minimum distances of the vertices considered earlier. If a vertex is not connected to the source by an edge, then it is considered to have distance ∞ from the source.

Algorithm Single-source-Dijkstra (V,E,s)
// The inputs to the algorithm consist of the set of vertices V, the set of edges E, and s
// the selected vertex, which is to serve as the source. Further, weights w(i,j) between
// every pair of vertices i and j are given. The algorithm finds and returns d_v, the
// minimum distance of each of the vertex v in V from s. An array D of the size of
// number of vertices in the graph is used to store distances of the various vertices
// from the source. Initially Distance of the source from itself is taken as 0

// and Distance D(v) of any other vertex v is taken as ∞.
// Iteratively distances of other vertices are modified taking into consideration the
// minimum distances of the various nodes from the node with most recently modified
// distance

D(s) ← 0
For each vertex v ≠ s do
D (v) ← ∞
// Let Set-Remaining-Nodes be the set of all those nodes for which the final minimum
// distance is yet to be determined. Initially
Set-Remaining-Nodes ← V
while (Set-Remaining-Nodes ≠ ϕ) **do**
begin
choose v ∈ Set-Remaining-Nodes such that D(v) is minimum
Set-Remaining-Nodes ← Set-Remaining-Nodes ~ {v}
For each node x ∈ Set-Remaining-Nodes such that w(v, x) ≠ ∞ do
D(x) ← min {D(x), D(v) + w (v, x)}
end

Next, we consider an example to illustrate the **Dijkstra's Algorithm**

Example 2.7.1

For the purpose, let us take the following graph in which, we take a as the source

Error!

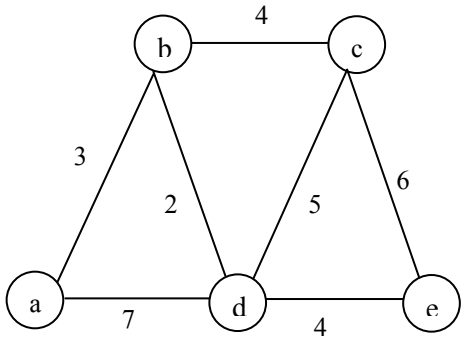
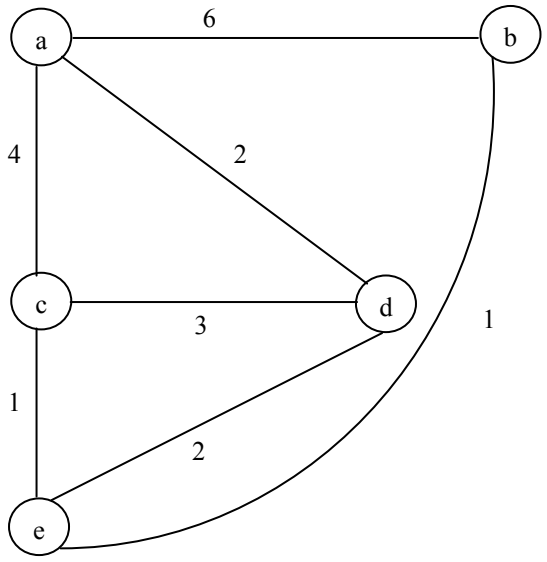


Figure: 2.7.2

Step	***Additional node***	***S = Set-of-Remaining Nodes***	***Distances from source of b, c, d, e***
Initialization	a	(b, c, d, e)	[3, ∞, 7 ∞]
1	b	(c, d, e)	[3, 3 + 4, 3 + 2, ∞]
2	d	(c, e)	[3, 3+4, 3 + 2, 3+2+4]
3	c	(e)	[3, 7, 5, 9]

For minimum distance from a, the node b is directly accessed; the node c is accessed through b; the node d is accessed through b; and the node e is accessed through b and d.

Ex. 5) Using Dijkstra's algorithm, find the minimum distances of all the nodes from node b which is taken as the source node, for the following graph.



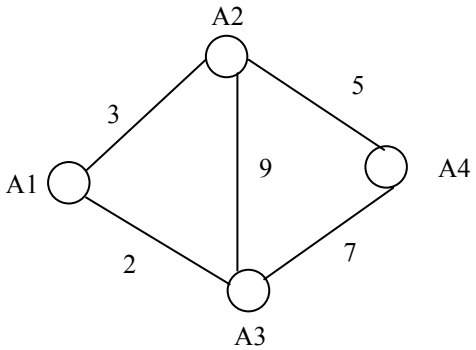
2.8 SUMMARY

In this unit, we have discussed the greedy technique **the essence of which is : In the process of solving an optimization problem, initially and at subsequent stages, evaluate the costs/benefits of the various available alternatives for the next step. Choose the alternative which is *optimal* in the sense that either it is the least costly or it is the maximum profit yielding. In this context, it may be noted that the *overall solution*, yielded by choosing *locally optimal steps*, may *not* be optimal.** Next, well-known algorithms viz., Prim's and Kruskal's that use greedy technique, to find spanning trees for connected graphs are discussed. Also Dijkstra's algorithm for solving Single-Source-Shortest path problem, again using greedy algorithm, is discussed.

2.9 SOLUTIONS/ANSWERS

Ex.1)

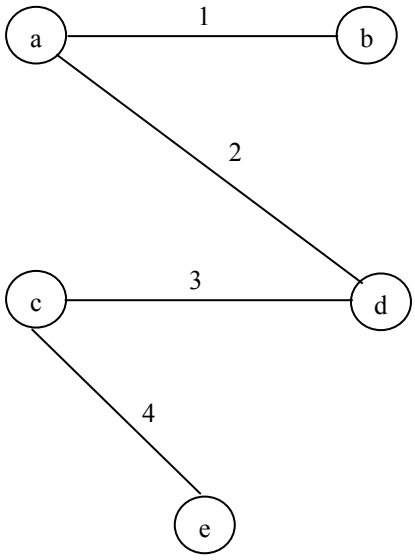
Consider the following graph, in which vertices/nodes represent cities of a country and each edge denotes a road between the cities denoted by the vertices of the edge. The label on each edge denotes the distance in 1000 kilometers between the relevant cities. The problem is to find an optimal path from A1 to A4.



Then greedy techniques suggests the route A1, A3, A4 of length 9000 kilometers, whereas the optimal path A1, A2, A4 is of length 8000 kilometers only.

Ex.2)

We will learn some systematic methods of finding a minimal spanning tree of a graph in later sections. However, by hit and trial, we get the following minimal spanning tree of the given graph.

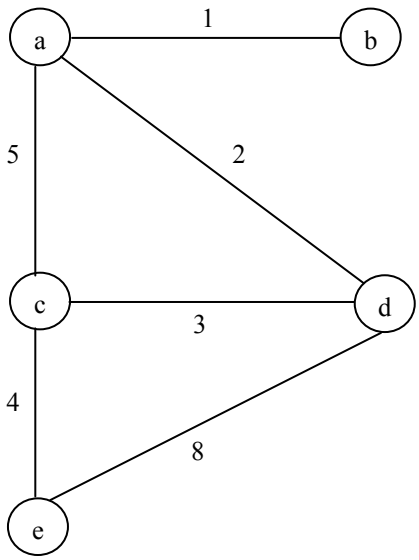


Ex.3)

The student should include the explanation on the lines of Example 2.5.1. However, the steps and stages in the process of solving the problem are as follows.

Initially

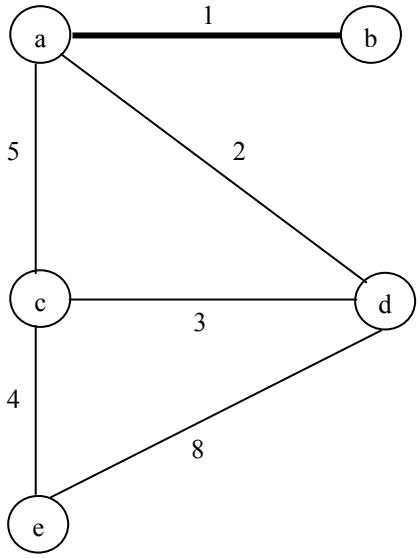
$V_T = (a)$
$E_T = \phi$



In the following figures, the edges in bold denote the chosen edges.

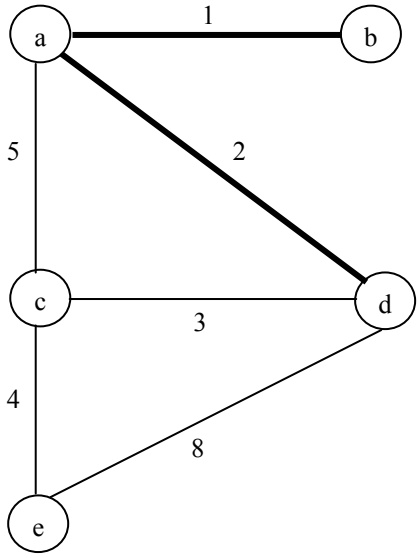
After First Iteration

V_T = (a, b)
E_T = ((a, b))



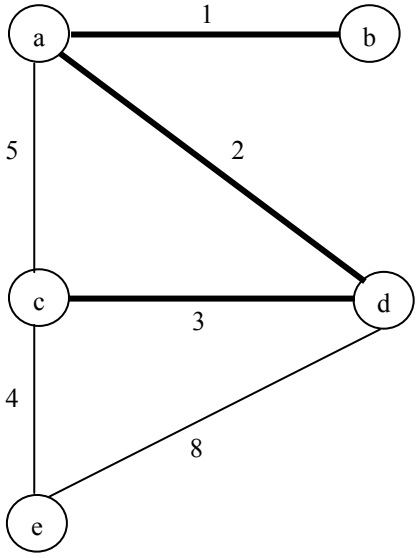
After Second Iteration

V_T = (a, b, d)
E_T = ((a, b), (a, d))



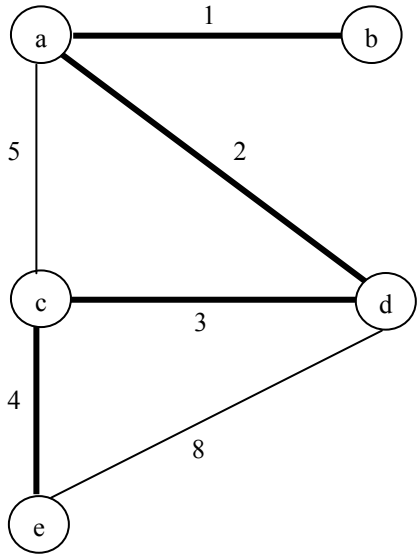
After Third Iteration

V_T = (a, b, c, d)
E_T = ((a, b), (a, d), (c, d))



After Fourth Iteration

V_T = (a, b, c, d, e)
E_T = ((a, b), (a, d), (c, d), (c, e))



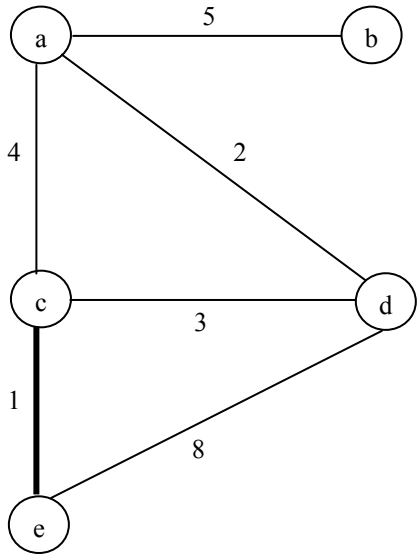
Ex. 4)

The student should include the explanation on the lines of Example 2.6.1. However, the steps and stages in the process of solving the problem are as follows:

The edges in bold denote the selected edges.

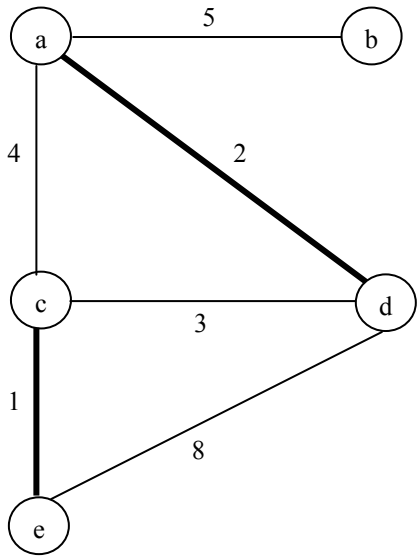
After First Iteration

E_g = ((c, e))



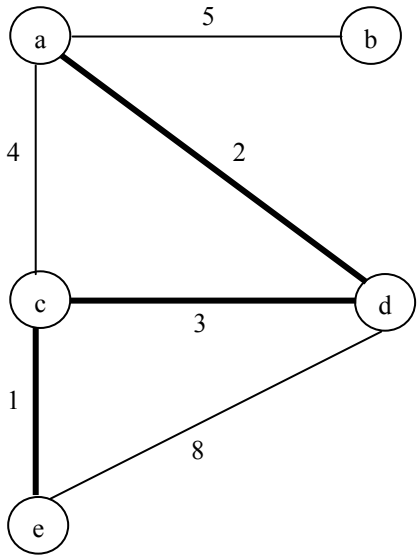
After Second Iteration

E_g = ((c, e), (a, d))



After Third Iteration

E_g = ((c, e), (a, d), (c, d))

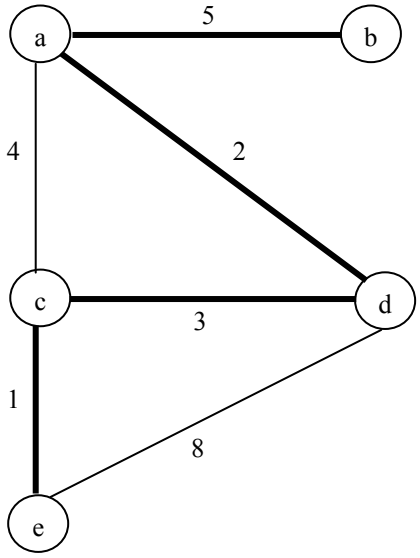


After Fourth Iteration

We can not take edge ac, because it forms a cycle with (a, d) and (c, d)

After Fifth Iteration

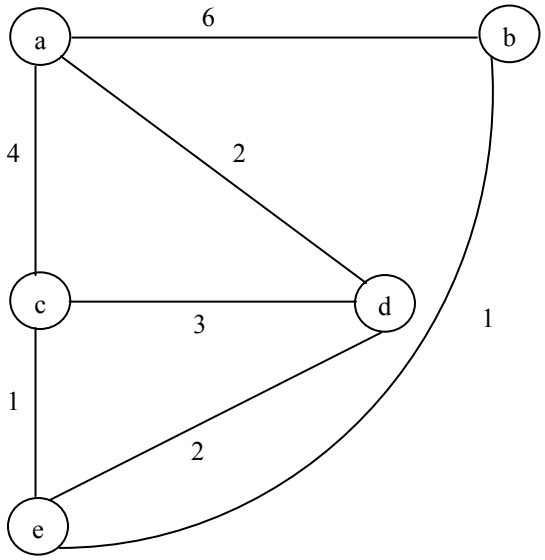
Eg = ((c, e), (a, d), (c, d), (a, b))



Now, on the above four edges all the vertices of the graph lie and these edges form a tree which is the required minimal spanning tree.

Ex. 5)

A copy of the graph is given below



Step	*Additional node*	*S = Set-of-Remaining Nodes*	*Distances from source of a, c, d, e*
Initialization	B	(a, c, d, e)	[6, ∞, ∞, 1]
1	E	(a, c, d)	[6, 2, 3 1,]
2	c	(a, d)	[6, 2, 3 1,]
3	d	(a)	[5, 2, 3, 1]

For minimum distance from b, node a is accessed through d and e; node c is accessed through e; node d is accessed through e and node e is accessed directly.

2.10 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour, (D.C. Health & Company, 1996).

2. *Algoritmics: The Spirit of Computing,* D. Harel, (Addison-Wesley Publishing Company, 1987).

3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth, (Narosa Publishing House).

4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately (Prentice-Hall International, 1996).

5. *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni, (Galgotia Publications).

6. *The Design and Analysis of Algorithms,* Anany Levitin, (Pearson Education, 2003).

7. *Programming Languages* (*Second Edition*) – *Concepts and Constructs,* Ravi Sethi, (Pearson Education, Asia, 1996).

UNIT 3 MODELS FOR EXECUTING ALGORITHMS-I: FA

Structure **Page Nos.**

3.0 INTRODUCTION

In the earlier two blocks and unit 1 and unit 2 of this block, we discussed a number of issues and techniques about designing algorithms. However, there are a number of problems for each of which, no algorithmic solution exists. Examples of such problems will be provided in unit 2 of block 4. However, many of these examples are found from the discipline of the well-known models of computation viz., finite automata, push-down automata and Tuning machines. In this unit, we discuss the topic of Finite Automata.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- define a finite automata for computation of a language;
- obtain a finite automata for a known language;
- create a grammar from language and vice versa;
- explain and create context free grammar and language;
- define the pushdown automata;
- apply the pumping lemma for non-context free languages; and
- find the equivalence of context free grammar and Pushdown Automata.

3.2 REGULAR EXPRESSIONS

In this unit, first we shall discuss the definitions of alphabet, string, and language with some important properties.

3.2.1 Introduction to Defining of Languages

For a language, defining rules can be of two types. The rules can either tell us how to test a string of alphabet letters that we might be presented with, to see if it is a valid word, i.e., a word in the language or the rules can tell us how to construct all the words in the language by some clear procedures.

Alphabet: A finite set of symbols/characters. We generally denote an alphabet by Σ. If we start an alphabet having only one letter, say, the letter z, then Σ = {z}.

Letter: Each symbol of an alphabet may also be called a letter of the alphabet or simply a letter.

Language over an alphabet: A set of words over an alphabet. Languages are denoted by letter L with or without a subscript.

String/word over an alphabet: Every member of any language is said to be a string or a world.

Example 1: Let L_1 be the language of all possible strings obtained by
L_1 = {z, zz, zzz, zzzz . . . . . .}

This can also be written as
$L_1 = \{z^n\}$ for n = 1, 2, 3, …..

A string of length zero is said to be **null string** and is represented by ∧.
Above given language L_1 does not include the null string. We could have defined it so as to include ∧. Thus, $L = \{z^n \mid n=0, 1, 2, 3\ldots\}$ contains the null string.

In this language, as in any other, we can define the operation of concatenation, in which two strings are written down side by side to form a new longer string. Suppose u = ab and v = baa, then uv is called concatenation of two strings u and v and is uv = abbaa and vu = baaab. The words in this language clearly analogous to the positive integers, and the operation of concatenation are analogous to addition:

z^n concatenated with z^m is the word z^{n+m}.

Example 2: If the word zzz is called c and the word zz is called d, then the word formed by concatenating c and d is

cd = zzzzz

When two words in our language L_1 are concatenated they produce another word in the language L_1. However, this may not be true in all languages.

Example 3: If the language is L_2 = {z, zzz, zzzzz, zzzzzzz…..}
$= \{z^{odd}\}$
$= \{z^{2n+1}$ for n = 0, 1, 2, 3….}
then c = zzz and d = zzzzz are both words in L_2, but their concatenation cd = zzzzzzzz is not a word in L_2. The reason is simple that member of L_2 are of odd length while after concatenation it is of even length.

Note: The alphabet for L_2 is the same as the alphabet for L_1.

Example 4: A Language L_3 may denote the language having strings of even lengths include of length 0. In other words, L_3 = {∧, zz, zzzz, …..}

Another interesting language over the alphabet Σ = {z} may be

Example 5: $L_4 = \{z^p : p \text{ is a prime natural number}\}$.
There are infinitely many possible languages even for a single letter alphabet Σ = {z}.

In the above description of concatenation we find very commonly, that for a single letter alphabet when we concatenate c with d, we get the same word as when we concatenate d with c, that is cd = dc **But this relationship does not hold for all**

languages. For example, in the English language when we concatenate "Ram" and "goes" we get "Ram goes". This is, indeed, a word but distinct from "goes Ram".

Now, let us define the reverse of a language L. If c is a word in L, then reverse (c) is the same string of letters spelled backward.
The reverse (L) = {reverse (w), w∈L}

Example 6: Reverse (zzz) = zzz
Reverse (173) = 371

Let us define a new language called PALINDROME over the alphabet Σ = {a,b}.

PALINDROME = {∧, and all strings w such that reverse (w) = w}

= {∧, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, …}

Concatenating two words in PALINDROME may or may not give a word in palindrome, e.g., if u = abba and v = abbcba, then uv = abbaabbcbba which is not palindrome.

3.2.2 Kleene Closure Definition

Suppose an alphabet Σ, and define a language in which any string of letters from Σ is a word, even the null string. We shall call this language the closure of the alphabet. We denote it by writing * after the name of the alphabet as a superscript, which is written as Σ^*. This notation is sometimes also known as **Kleene Star**.

For a given alphabet Σ, the language L consists of all possible strings, including the null string.

For example, If Σ = {z}, then, $\Sigma^* = L_1$ = {∧, z, zz, zzz …..}

Example 7: If Σ = {0, 1}, then, Σ* = {∧, 0, 1, 00, 01, 10, 11, 000, 001 …..}
So, we can say that Kleene Star is an operation that makes an infinite language of strings of letters out of an alphabet, if the alphabet, $\Sigma \neq \phi$. However, by the definition alphabet Σ may also be ϕ. In that case, Σ^* is finite. By "infinite language, we mean a language with infinitely many words.

Now, we can generalise the use of the star operator to languages, i.e., to a set of words, not just sets of alphabet letters.

Definition: If s is a set of words, then by s^* we mean the set of all finite strings formed by concatenating words from s, where any word may be used as often.

Example 8: If s = {cc, d}, then
s^* = {∧ or any word composed of factors of cc and d}
= {∧ or all strings of c's and d's in which c's occur in even clumps}.
The string ccdcccd is not in s^* since it has a clump of c's of length 3.
$\{x : x = \wedge \text{ or } x = (cc)^{i_1} d^{j_1} (cc)^{i_2} d^{j_2} .....(cc)^{i_m} (d)^{j_m}\}$ where $i_1, j_1, ... i_m, j_m \geq 0$.

Positive Closure: If we want to modify the concept of closure to refer to only the concatenation leading to non-null strings from a set s, we use the notation + instead of *. This plus operation is called positive closure.

Theorem 1: For any set s of strings prove that $s^* = (s^*)^* = s^{**}$

Proof: We know that every word in s^{**} is made up of factors from s^*.

Also, every factor from s^* is made up of factors from s.

Therefore, we can say that every word in s^{**} is made up of factors from s.

First, we show $s^{**} \subset s^*$. (i)
Let $x \in s^{**}$…. Then $x = x_1 \ldots x_n$ for some $x_1 \in s^*$ which implies $s^{**} \subset s^*$

Next, we show $s^* \subset s^{**}$.
$s^* \subset s^{**}$ (ii)

By above inclusions (i) and (ii), we prove that
$s^* = s^{**}$

Now, try some exercises.

Ex.1) If u = ababb and v = baa then find
(i) uv (ii) vu (iii) uv (iv) vu (v) uuv.

Ex.2) Write the Kleene closure of the following:
(i) {aa, b}
(ii) {a, ba}

3.2.3 Formal Definition of Regular Expressions

Certain sets of strings or languages can be represented in algebraic fashion, then these algebraic expressions of languages are called **regular expressions**. Regular expressions are in **Bold** face. The symbols that appear in regular use of the letters of the alphabet Σ are the symbol for the null string ∧, parenthesis, the star operator, and the plus sign.

The set of regular expressions is defined by the following rules:

1. Every letter of Σ can be made into a regular expression ∧ itself is a regular expression.
2. If **l** and **m** are regular expressions, then so are
   (i) **(l)**
   (ii) **lm**
   (iii) **l+m**
   (iv) $\mathbf{l^*}$
   (v) $\mathbf{l^+ = ll^*}$
3. Nothing else is regular expression.

For example, now we would build expression from the symbols 0,1 using the operations of union, concatenation, and Kleene closure.

(i) **01** means a zero followed by a one (concatenation)
(ii) **0+1** means either a zero or a one (union)
(iii) $\mathbf{0^*}$ means $\wedge$**+0+00+000+**….. (Kleen closure).

With parentheses, we can build larger expressions. And, we can associate meanings with our expressions. Here's how

Expression	**Set represented**
$\mathbf{(0+1)^*}$	all strings over {0,1}
$\mathbf{0^*10^*10^*}$	strings containing exactly two ones
$\mathbf{(0+1)^*11}$	strings which end with two ones.

The language denoted/represented by the regular expression R is L(R).

Example 9: The language L defined by the regular expression **ab*a** is the set of all strings of a's and b's that begin and end with a's, and that have nothing but b's inside.

L = {aa, aba, abba, abbba, abbbba, }

Example 10: The language associated with the regular expression $\mathbf{a^*b^*}$ contains all the strings of a's and b's in which all the a's (if any) come before all the b's (if any).

L = {∧, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaa,…)

Note that ba and aba are not in this language. Notice also that there need not be the same number of a's and b's.

Example 11: Let us consider the language L defined by the regular expression $\mathbf{(a+b)^* a(a+b)^*}$**.** The strings of the language L are obtained by concatenating a string from the language corresponding to $\mathbf{(a+b)^*}$ followed by a string from the language associated with $\mathbf{(a+b)^*}$. We can also say that the language is a set of all words over the alphabet Σ = {a,b} that have an a in them somewhere.

To make the association/correspondence/relation between the regular expressions and their associated languages more explicit, we need to define the operation of multiplication of set of words.

Definition: If S and T are sets of strings of letters (they may be finite or infinite sets), we define the product set of strings of letters to be. ST = {all combinations of a string from S concatenated with a string from T in that order}.

Example 12: If S = {a, aa, aaa}, T = {bb, bbb}
Then, ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}.

Example 13: If S = {a bb bab}, T = {∧ bbbb}
Then, ST = {a bb bab abbbb bbbbbb babbbbb}

Example 14: If L is any language, Then, L∧ = ∧L = L.

Ex.3) Find a regular expression to describe each of the following languages:

(a) {a,b,c}
(b) {a,b,ab,ba,abb,baa,….}
(c) {∧,a,abb,abbbb,….}

Ex.4) Find a regular expression over the alphabet {0,1,} to describe the set of all binary numerals without leading zeroes (except 0 itself). So the language is the set

{0,1,10,11,100,101,110,111,…}.

3.2.4 Algebra of Regular Expressions

There are many general equalities for regular expressions. We will list a few simple equalities together with some that are not so simple. All the properties can be verified by using properties of languages and sets. We will assure that R,S and T denote the arbitrary regular expressions.

Properties of Regular Expressions

1. (R+S)+T = R+(S+T)

2. $R+R = R$
3. $R+\phi = \phi+R = R$.
4. $R+S = S+R$
5. $R\phi = \phi R = \phi$
6. $R\wedge = \wedge R = R$
7. $(RS)T = R(ST)$
8. $R(S+T) = RS+RT$
9. $(S+T)R = SR+TR$
10. $\phi^* = \wedge^* = \wedge$
11. $R^*R^* = R^* = (R^*)^*$
12. $RR^* = R^*R = R^* = \wedge+RR^*$
13. $(R+S)^* = (R^*S^*)^* = (R^*+S^*)^* = R^*S^* = (R^*S)^*R^* = R^*(SR^*)^*$
14. $(RS)^* = (R^*S^*)^* = (R^*+S^*)^*$

Theorem 2: Prove that R+R = R

Proof : We know the following equalities:

$L(\mathbf{R+R}) = L(\mathbf{R})UL(\mathbf{R}) = L(\mathbf{R})$

So $\mathbf{R}+\mathbf{R} = \mathbf{R}$

Theorem 3: Prove the distributive property

$R(S+T) = RS+RT$

Proof: The following set of equalities will prove the property:

$$\begin{aligned} L(R\,(S+T)) &= L(R)L(S+T) \\ &= L(R)(L(S)UL(T)) \\ &= (L(R)L(S))U(L(R)L(T)) \\ &= L(RS+RT) \end{aligned}$$

Similarly, by using the equalities we can prove the rest. The proofs of the rest of the equalities are left as exercises.

Example 15: Show that $R+RS^*S = a^*bS^*$, where $R = b+aa^*b$ and S is any regular expression.

$R+RS^*S = R\wedge+RS^*S$ (property 6)

$= R(\wedge+S^*S)$ (property 8)

$= R(\wedge+SS^*)$ (property 12)

$= RS^*$ (property 12)

$= (b+aa^*b)S^*$ (definition of R)
$= (\wedge+aa^*)\,bS^*$ (properties 6 and 8)

$= a^*bS^*$. (Property 12)

Try an exercise now.

Ex.5) Establish the following equality of regular expressions:
$b^*(abb^*+aabb^*+aaabb^*)^* = (b+ab+aab+aaab)^*$

As we already know the concept of language and regular expressions, we have an important type of language derived from the regular expression, called **regular language.**

3.3 REGULAR LANGUAGES

Language represented by a regular expression is called a regular language. In other words, we can say that a regular language is a language that can be represented by a regular expression.

Definition: For a given alphabet Σ, the following rules define the regular language associated with a regular expression.

Rule 1: $\phi, \{\wedge\}$ and {a} are regular languages denoted respectively by regular expressions $\boldsymbol{\phi}$ and $\wedge$.

Rule 2: For each a in Σ, the set {a} is a regular language denoted by the regular expression **a**.

Rule 3: If **l** is a regular expression associated with the language L and **m** is a regular expression associated with the language M, then:

(i) The language = {xy : x∈L and y∈M} is a regular expression associated with the regular expression **lm.**

(ii) The regular expression **l+m** is associated with the language formed by the union of the sets L and M.

language (**l+m**) = L∪M

(iii) The language associated with the regular expression $(\mathbf{l})^*$ is L^*, the Kleen Closure of the set L as a set of words:

language $(\mathbf{l}^*) = L^*$.

Now, we shall derive an important relation that, all finite languages are regular.

Theorem 4: If L is a finite language, then L can be defined by a regular expression. In other words, all finite languages are regular.

Proof: A language is finite if it contains only finitely many words.

To make one regular expression that defines the language L, turn all the words in L into bold face type and insert plus signs between them. For example, the regular expression that defines the language L = {baa, abbba, bababa} is **baa + abbba + bababa.**

Example 16: If L = {aa, ab, ba, bb}, then the corresponding regular expression is **aa + ab +ba + bb.**

Another regular expression that defines this language is **(a+b) (a+b).**

So, a particular regular language can be represented by more than one regular expressions. Also, by definition, each regular language must have at least one regular expression corresponding to it.

Try some exercises.

Ex.6) Find a language to describe each of the following regular expressions:
(a) **a+b** (b) $\mathbf{a+b^*}$ (c) $\mathbf{a^*bc^*+ac}$

Ex.7) Find a regular expression for each of the following languages over the alphabet {a,b}.

(a) strings with even length.
(b) strings containing the sub string aba.

In our day to day life we oftenly use the word Automatic. Automation is the process where the output is produced directly from the input without direct involvement of mankind. The input passes from various states in process for the processing of a language we use very important finite state machine called finite automata.

3.4 FINITE AUTOMATA

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (and, since the advent of VLSI systems sometimes finite automata represent circuits.) computer scientists adore them because they adapt very likely to algorithm design. For example, the lexical analysis portion of compiling and translation. Mathematicians are introduced by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept.

Can a machine recognise a language? The answer is yes for some machine and some elementary class of machines called finite automata. Regular languages can be represented by certain kinds of algebraic expressions by Finite automaton and by certain grammars. For example, suppose we need to compute with numbers that are represented in scientific notation. Can we write an algorithm to recognise strings of symbols represented in this way? To do this, we need to discuss some basic computing machines called finite automaton.

An automata will be a finite automata if it accepts all the words of any regular language where language means a set of strings. In other words, The class of regular language is exactly the same as the class of languages accepted by FA's, a deterministic finite automata.

3.4.1 Definition

A system where energy and information are transformed and used for performing some functions without direct involvement of man is called automaton. Examples are automatic machine tools, automatic photo printing tools, etc.

A finite automata is similar to a finite state machine. A finite automata consists of five parts:

(1) a finite set of states;
(2) a finite set of alphabets;
(3) an initial state;
(4) a subset of set of states (whose elements are called "yes" state or; accepting state;) and
(5) a next-state function or a transition state function.

A finite automata over a finite alphabet A can be thought of as a finite directed graph with the property that each node omits one labelled edge for each distinct element of A. The nodes are called states. There is one special state called **the start** (or **initial**) state, and there is a possible empty set of states called **final states**.

For example, the labelled graph in *Figure1* given below represents a DFA over the alphabet A = {a,b} with start state 1 and final state 4.

Figure 1: Finite automata

We always indicate the start state by writing the word start with an arrow painting to it. Final states are indicated by double circle.

The single arrow out of state 4 labelled with a,b is short hand for two arrows from state 4, going to the same place, one labelled a and one labelled b. It is easy to check that this digraph represents a DFA over {a,b} because there is a start state, and each state emits exactly two arrows, one labelled with a and one labelled with b.

So, we can say that a finite automaton is a collection of three tuples:

1. A finite set of states, one of which is designated as the initial state, called the start state, and some (may be none) of which we designated as final states.

2. An alphabet Σ of possible input letters from which are formed strings that are to be read one letter at a time.

3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.

For example, the input alphabet has only two letters a and b. Let us also assume that there are only three states, x, y and z. Let the following be the rules of transition:

1. from state x and input a go to state y;

2. from state x and input b go to state z;

3. from state y and input b go to state x;

4. from state y and input b go to state z; and

5. from state z and any input stay at state z.

Let us also designate state x as the starting state and state z as the only final state. Let us examine what happens to various input strings when presented to this FA. Let us start with the string aaa. We begin, as always, in state x. The first letter of the string is an a, and it tells us to go state y (by rule 1). The next input (instruction) is also an a, and this tells us (by rule 3) to go back to state x. The third input is another a, and (by Rule 1) again we go to the state y. There are no more input letters in the

input string, so our trip has ended. We did not finish in the final state (state z), so we have an unsuccessful termination of our run.

The string aaa is not in the language of all strings that leave this FA in state z. The set of all strings that do leave as in a final state is called the language defined by the finite automaton. The input string aaa is not in the language defined by this FA. We may say that the string aaa is not accepted by this FA because it does not lead to a final state. We may also say "aaa is rejected by this FA." The set of all strings accepted is the language associated with the FA. So, we say that L is the language accepted by this FA. FA is also called a language recogniser.

Let us examine a different input string for this same FA. Let the input be abba. As always, we start in state x. Rule 1 tells us that the first input letter, a, takes us to state y. Once we are in state y we read the second input letter, which is ab. Rules 4 now tells us to move to state z. The third input letter is a b, and since we are in state z, Rule 5 tells us to stay there. The fourth input letter is an a, and again Rule 5 says state z. Therefore, after we have followed the instruction of each input letter we end up in state z. State z is designated as a final state. So, the input string abba has taken us successfully to the final state. The string abba is therefore a word in the language associated with this FA. The word abba is accepted by this FA.

It is not difficult for us to predict which strings will be accepted by this FA. If an input string is made up of only the letter a repeated some number of times, then the action of the FA will be jump back and forth between state x and state y. No such word can ever be accepted.

To get into state z, it is necessary for the string to have the letter b in it as soon as a b is encountered in the input string, the FA jumps immediately to state z no matter what state it was before. Once in state z, it is impossible to leave. When the input strings run out, the FA will still be in state z, leading to acceptance of the string.

So, the FA above will accept all the strings that have the letter b in them and no other strings. Therefore, the language associated with this FA is the one defined by the regular expression $\mathbf{(a+b)^* b(a+b)^*}$.

The list of transition rules can grow very long. It is much simpler to summarise them in a table format. Each row of the table is the name of one of the states in FA, and each column of this table is a letter of the input alphabet. The entries inside the table are the new states that the FA moves into the transition states. The transition table for the FA we have described is:

Table 1

State	Input	
	a	b
Start x	y	z
y	x	z
Final z	z	z

The machine we have already defined by the transition list and the transition table can be depicted by the state graph in *Figure 2*.

Figure 2: State transition graph

Note: A single state can be start as well as final state both. There will be only one start state and none or more than one final states in Finite Automaton.

3.4.2 Another Method to Describe FA

There is a traditional method to describe finite automata which is extremely intuitive. It is a picture called a graph. The states of the finite automaton appear as vertices of the graph while the transitions from state to state under inputs are the graph edges. The state graph for the same machine also appears in *Figure 3* given below.

Figure 3: Finite automata

The finite automata shown in *Figure 3* can also be represented in Tabular form as below:

Table 2

	State	Input		Accept?
		0	1	
Start	1	1	2	No
Final	2	2	3	Yes
	3	3	3	No

Before continuing, let's examine the computation of a finite automaton. Our first example begins in state one and reads the input symbols in turn changing states as necessary. Thus, a computation can be characterized by a sequence of states. (Recall that Turing machine configurations needed the state plus the tape content. Since a finite automata on never writes, we always know what is on the tape and need only look at a state as a configuration). Here is the sequence for the input 0001001.

Input Read : 0 0 0 1 0 0 1

States : $1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 3$

Example 17 (An elevator controller): Let's imagine an elevator that serves two floors. Inputs are calls to a floor either from inside the elevator or from the floor itself. This makes three distinct inputs possible, namely:

0 - no calls
1 - call to floor one
2 - call to floor two

The elevator itself can be going up, going down, or halted at a floor. If it is on a floor, it could be waiting for a call or about to go to the other floor. This provides us with the six states shown in *Figure 4* along with the state graph for the elevator controller.

W1 Waiting on first floor
U1 About to go up
UP Going up
DN Going down
W2 Waiting-second floor
D2 About to go down.

Figure 4: Elevator control

A transition state table for the elevator is given in *Table 3*:

Table 3: Elevator Control

State	Input		
	None	**call to 1**	**call to 2**
W1 (wait on 1)	W1	W1	UP
U1 (start up)	UP	U1	UP
UP	W2	D2	W2
DN	W1	W1	U1
W2 (wait on 2)	W2	DN	W2
D2 (start down)	DN	DN	D2

Accepting and rejecting states are not included in the elevator design because acceptance is not an issue. If we were to design a more sophisticated elevator, it might have states that indicated:

Finite automata

a) power faukyrem
b) overloading, or
c) breakdown

In this case, acceptance and rejection might make sense.

Let us make a few small notes about the design. If the elevator is about to move (i.e., in state U1 or D2) and it is called to the floor it is presently on it will stay. (This may be good Try it next time you are in an elevator.) And, if it is moving (up or down) and gets called back the other way, it remembers the call by going to the U1 or D2 state upon arrival on the next floor. Of course, the elevator does not do things like open and close doors (these could be states too) since that would have added complexity to the design. Speaking of complexity, imagine having 100 floors. That is our levity for this section. Now that we know what a finite automaton is, we must (as usual) define it precisely.

Definition : ***A finite automation M is a quintuple M = (Q,Σ,δ,q_0,F) where :***

Q is a finite set (of states)
Σ is a finite alphabet (of input symbols)
$\delta: Q \times \Sigma \rightarrow Q$ (next state function)
$q_0 \in Q$ (the starting state)
$F \subseteq Q$ (the accepting states)

We also need some additional notation. The next state function is called the transition function and the accepting states are often called final states. The entire machine is usually defined by presenting a transition state table or a transition diagram. In this way, the states, alphabet, transition function, and final states are constructively defined. The starting state is usually the lowest numbered state. Our first example of a finite automaton is:

$$M = (\{q_1, q_2, q_3\}, \{0,1\}, \delta, q_1, \{q_2\}$$

Where the transition function δ, is defined explicitly by either a state table or a state graph.

3.5 SUMMARY

In this unit we introduced several formulations for regular languages, regular expressions are algebraic representations of regular languages. Finite Automata are machines that recognise regular languages. From regular expressions, we can derive regular languages. We also made some other observations. Finite automata can be used as output devices - Mealy and Moore machines.

3.6 SOLUTIONS/ANSWERS

Ex.1)

(i) ababbbaa
(ii) baaababb
(iii) ab abb ab abb
(iv) baa baa
(v) ababbababb baa

Ex.2)

(i) Suppose aa = x
Then { x, b}* = {$\wedge$, x, b, xx, bb, xb, bx, xxx, bxx, xbx, xxb, bbx, bxb, xbb, bbb} substituting x = aa
{aa,b}* = { $\wedge$, aa, b, aaaa, bb, aab, baa, aaaaaa, baaaa, aabaa,

(ii) {a,ba}*= { $\wedge$, a, ba, aa, baba, aba, baa, … }

Ex.3)

(a) **a+b+c**
(b) **ab^*+ba^***
(c) **$\wedge+a(bb)^*$**

Ex.4)

$0+1(0+1)^*$

Ex.5)

Starting with the left side and using properties of regular expressions, we get

$b^*(abb^* + aabb^*+aaabb^*)^*$
= **$b^*((ab+aab+aaab)b^*)^*$** (property 9)
= **$(b + ab + aab + aaab)^*$** (property 7).

Ex.6)

(a) {a,b}
(b) {a,$\wedge$,b,bb,….b^n,….}
(c) {a,b,ab,bc,abb,bcc,…ab^n,bc^n,….}

Ex.7)

(a) **$(aa+ab+ba+bb)^*$**
(b) **$(a+b)^*aba(a+b)^*$**

3.7 FURTHER READINGS

1. *Elements of the Theory of Computation*, H.R. Lewis & C.H.Papadimitriou, PHI, (1981).

2. *Introduction to Automata Theory, Languages, and Computation* (II Ed.), J.E. Hopcroft, R.Motwani & J.D.Ullman: Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language, and Computation*, J.E. Hopcroft and J.D. Ullman: Narosa Publishing House (1987).

4. *Introduction to Languages and Theory of Computation*, J.C. Martin, Tata-Mc Graw-Hill (1997).

5. *Computers and Intractability – A Guide to the theory of NP-Completeness*, M.R. Garey & D.S. Johnson: W.H. Freeman and Company (1979).

UNIT 4 MODELS FOR EXECUTING ALGORITHMS-II: PDFA & CFG

Structure **Page Nos.**

4.0 INTRODUCTION

We have mentioned earlier that not every problem can be solved algorithmically and that good sources of examples of such problems are provided by formal models of computation viz., FA, PDFA and TA. In the previous unit, we discussed FA. In this unit, we discuss PDFA, CFG and related topics.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- explain and create context free grammar and language;
- define the pushdown automata;
- find the equivalence of context free grammar and Pushdown Automata, and
- create a grammar from language and vice versa.

4.2 FORMAL LANGUAGE & GRAMMAR

In our day-to-day life, we often use the common words such as grammar and language. Let us discuss it through one example.

Example 1: If we talk about a sentence in English language, “Ram reads”, this sentence is made up of Ram and reads. Ram and reads are replaced for <noun> and <verb>. We can say simply that a sentence is changed by noun and verb and is written as

<sentence> $\rightarrow$ <noun> <verb>

where noun can be replaced with many such values as Ram, Sam, Gita…. and also <verb> can be replaced with many other values such as read, write, go …. As noun and verb are replaced, we easily write

<noun> $\rightarrow$ I
<noun> $\rightarrow$ Ram
<noun> $\rightarrow$ Sam
<verb> $\rightarrow$ reads
<verb> $\rightarrow$ writes

From the above, we can collect all the values in two categories. One is with the parameter changing its values further, and another is with termination. These

collections are called variables and terminals, respectively. In the above discussion variables are, <sentence>, <noun> and <verb>, and terminals are I, Ram, Sam, read, write. As the sentence formation is started with <sentence>, this symbol is special symbol and is called **start symbol**.

Now **formally**, a Grammar G = (V, Σ, P, S) where,

- V is called the set of variables. e.g., {S, A, B, C}
- Σ is the set of terminals, e.g., {a, b}
- P is a set of production rules
  (- Rules of the form $A \rightarrow \alpha$ where $A \in (VU\Sigma)^+$ and $\alpha \in (VU\Sigma)^+$ e.g., $S \rightarrow aA$).
- S is a special variable called the start symbol $S \in V$.

Structure of grammar: If L is a language over an alphabet A, then a grammar for L consists of a set of grammar rules of the form

$$x \rightarrow y$$

where x and y denote strings of symbols taken from A and from a set of grammar symbols disjoint from A. The grammar rule $x \rightarrow y$ is called a production rule, and application of production rule (x is replaced by y), is called derivation.

Every grammar has a special grammar symbol called the start symbol and there must be at least one production with the left side consisting of only the start symbol. For example, if S is the start symbol for a grammar, then there must be at least one production of the form $S \rightarrow y$.

Example 2: Suppose A = {a, b, c} then a grammar for the language A^* can be described by the following four productions:

$S \rightarrow \wedge$ (i)
$S \rightarrow aS$ (ii)
$S \rightarrow bS$ (iii)
$S \rightarrow cS$ (iv)

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb = aacb$
using prod.(u) using prod.(ii) using prod.(iv) using prod.(iii) using prod.(i)

The desired derivation of the string is aacb. Each step in a derivation corresponds to a branch of a tree and this true is called **parse tree**, whose root is the start symbol. The completed derivation and parse tree are shown in the *Figure 1,2,3*:

Figure 1: S ⇒ aS **Figure 2: S ⇒ aS ⇒ aaS**

Figure 3: S ⇒ aS ⇒ aaS ⇒ aacS

Let us derive the string aacb, its parse tree is shown in *Figure 4*.

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\wedge = aacb$

Figure 4: Parse tree deriving aacb

Sentential form: A string made up of terminals and/or non-terminals is called a sentential form.

In example 1, formally grammar is rewritten as

In G = (V, Σ, P, S) where
V = {<sentence>, <noun>, <verb>}
Σ = {Ram, reads,…}
P = <sentence> → <noun> <verb>
<noun> → Ram
<verb> → reads, and
S = <sentence>

If x and y are sentential forms and $\alpha \rightarrow \beta$ is a production, then the replacement of α by β in $x\alpha y$ is called a derivation, and we denote it by writing

$$x\alpha y \Rightarrow x\beta y$$

To the left hand side of the above production rule x is left context and y is right context. If the derivation is applied to left most variable of the right hand side of any production rule, then it is called leftmost derivation. And if applied to rightmost then is called rightmost derivation.

The language of a Grammar:

A language is generated from a grammar. If G is a grammar with start symbol S and set of terminals $\sum$, then the language of G is the set

$$L(G) = \{W \mid W \in \Sigma^* \text{ and } S \underset{G}{\overset{*}{\Rightarrow}} W\}.$$

Any derivation involves the application production Rules. If the production rule is applied once, then we write $\alpha \underset{G}{\Rightarrow} B$. When it is more than one, it is written as $\alpha \underset{a}{\overset{*}{\Rightarrow}} \beta$.

Recursive productions: A production is called recursive if its left side occurs on its right side. For example, the production $S \rightarrow aS$ is recursive. A production $A \rightarrow \alpha$ is indirectly recursive. If A derives a sentential form that contains A, Then, suppose we have the following grammar:

$$S \rightarrow b/aA$$
$$A \rightarrow c/bS$$

the productions $S \rightarrow aA$ and $A \rightarrow bs$ are both indirectly recursive because of the following derivations:

$$S \Rightarrow aA \Rightarrow abS,$$
$$A \Rightarrow bS \Rightarrow baA$$

A grammar is recursive if it contains either a recursive production or an indirectly recursive production.

A grammar for an infinite language must be recursive.

Example 3: Consider $\{\wedge, a, aa, \ldots, a^n, \ldots\} = \{a^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form ax for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \wedge/aS.$$

Now, we shall derive the string aaa:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa.$$

Example 4: Consider $\{\wedge, ab, aabb, \ldots, a^n b^n, \ldots\} = \{a^n b^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form axb for some string x in the language. The following grammar will derive any of the strings:

$$S \rightarrow \wedge/aSb.$$

For example, we will derive the string aaabbb;

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

Example 5: Consider a language $\{\wedge, ab, abab, \ldots, (ab)^n, \ldots\} = \{(ab)^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form abx for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \wedge/abS.$$

For example, we shall derive the string ababab:

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow ababab.$$

Sometimes, a language can be written in terms of simpler languages, and a grammar can be constructed for the language in terms of the grammars for the simpler languages. We will now concentrate on operations of union, product and closure.

Suppose M and N are languages whose grammars have disjoint sets of non-terminals. Suppose also that the start symbols for the grammars of M and N are A and B, respectively. Then, we use the following rules to find the new grammars generated from M and N:

Union Rule: The language $M \cup N$ starts with the two productions

$$S \rightarrow A/B.$$

Product Rule: The language MN starts with the production.

$$S \rightarrow AB$$

Closure Rule: The language M^* starts with the production

$$S \rightarrow AS/\wedge.$$

Example 6: Using the Union Rule:

Let's write a grammar for the following language:

$$L = \{\wedge, a, b, aa, bb, \ldots, a^n, b^n, \ldots\}.$$

L can be written as union.

$$L = M \cup N,$$

Where $M = \{a^n \mid n \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$.

Thus, we can write the following grammar for L:

$S \rightarrow A \mid B$ union rule,
$A \rightarrow \wedge/aA$ grammar for M,
$B \rightarrow \wedge/bB$ grammar for N.

Example 7: Using the Product Rule:

We shall write a grammar for the following language:

$$L = \{a^m b^n \mid m,n \geq 0\}.$$

L can be written as a product L = MN, where $M = \{a^m \mid m \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$. Thus we can write the following grammar for L:

$S \rightarrow AB$ product rule
$A \rightarrow \wedge/aA$ grammar for M,
$B \rightarrow \wedge/bB$ grammar for N,

Example 8: Using the Closure Rule: For the language L of all strings with zero or more occurrence of aa or bb. $L = \{aa, bb\}^*$. If we let M = {aa, bb}, then $L = M^*$. Thus, we can write the following grammar for L:

$S \rightarrow AS/\wedge$ closure rule,
$A \rightarrow aa/bb$ grammar for M.

We can simplify the grammar by substituting for A to obtain the following grammar:

$$S \rightarrow aaS/bbS/\wedge$$

Example 9: Let $\Sigma = \{a, b, c\}$. Let S be the start symbol. Then, the language of palindromes over the alphabet Σ has the grammar.

$$S \rightarrow aSa/bSb/cSc/a/b/c/\wedge.$$

For example, the palindrome abcba can be derived as follows:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abcba$$

Ambiguity: A grammar is said to be ambiguous if its language contains some string that has two different parse tree. This is equivalent to saying that some string has two distinct leftmost derivations or that some string has two distinct rightmost derivations.

Example 10: Suppose we define a set of arithmetic expressions by the grammar:

$$E \rightarrow a/b/E–E$$

Figure 5: Parse tree **Figure 6: Parse tree showing ambiguity**

This is the parse tree for an ambiguous string.

The language of the grammar $E \rightarrow a/b/E\text{-}E$ contains strings like a, b, b–a, a–b–a, and b–b–a–b. This grammar is ambiguous because it has a string, namely, a–b–a, that has two distinct parse trees.

Since having two distinct parse trees mean the same as having two distinct left most derivations.

$$E \Rightarrow E–E \Rightarrow a–E \Rightarrow a–E–E \Rightarrow a – b – E \Rightarrow a – b – a.$$
$$E \Rightarrow E–E \Rightarrow E–E–E \Rightarrow a–E–E \Rightarrow a – b – E \Rightarrow a – b – a.$$

The same is the case with rightmost derivation.

- A derivation is called a leftmost derivation if at each step the leftmost non-terminal of the sentential form is reduced by some production.
- A derivation is called a rightmost derivation if at each step the rightmost non-terminal of the sentential form is reduced by some production.

Let us try some exercises.

Ex.8) Given the following grammar

$S \rightarrow S[S]/\wedge$

For each of the following strings, construct a leftmost derivation, a rightmost derivation and a parse tree.

(a) [ ] (b) [[ ]] (c) [ ] [ ] (d) [[] [[]]]

Ex.9) Find a grammar for each language

(a) $\{a^m b^n \mid m,n \in N, n>m\}$.
(b) $\{a^m bc^n \mid n \in N\}$.

Ex.10) Find a grammar for each language:

(a) The even palindromes over {a, b}.
(b) The odd palindromes over {a, b}.

Chomsky Classification for Grammar:

As you have seen earlier, there may be many kinds of production rules. So, on the basis of production rules we can classify a grammar. According to Chomsky classification, grammar is classified into the following types:

Type 0: This grammar is also called **unrestricted grammar**. As its name suggests, it is the grammar whose production rules are unrestricted.

All grammars are of type 0.

Type 1: This grammar is also called **context sensitive grammar**. A production of the form $xAy \rightarrow x\alpha y$ is called a type 1 production if $\alpha \neq \wedge$, which means length of the working string does not decrease.

In other words, $|xAy| \leq |x\alpha y|$ as $\alpha \neq \wedge$. Here, x is left context and y is right context.

A grammar is called type 1 grammar, if all of its productions are of type 1. For this, grammar $S \rightarrow \wedge$ is also allowed.

The language generated by a type 1 grammar is called a type 1 or **context sensitive language**.

Type 2: The grammar is also known as **context free grammar**. A grammar is called type 2 grammar if all the production rules are of type 2. A production is said to be of type 2 if it is of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. In other words, the left hand side of production rule has no left and right context. The language generated by a type 2 grammar is called **context free language**.

Type 3: A grammar is called type 3 grammar if all of its production rules are of type 3. (A production rule is of type 3 if it is of form $A \rightarrow \wedge$, $A \rightarrow a$ or $A \rightarrow aB$ where $a \in \Sigma$ and $A,B \in V$), i.e., if a variable derives a terminal or a terminal with one variable. This type 3 grammar is also called **regular grammar**. The language generated by this grammar is called **regular language**.

Ex.11) Find the highest type number that can be applied to the following grammar:

(a) $S \rightarrow ASB/b$, $A \rightarrow aA$
(b) $S \rightarrow aSa/bSb/a/b/\wedge$
(c) $S \rightarrow Aa$, $A \rightarrow S/Ba$, $B \rightarrow abc$.

4.3 CONTEXT FREE GRAMMAR (CFG)

We know that there are non-regular languages. For example, $\{a^n b^n \mid n \geq 0\}$ is non-regular language. Therefore, we can't describe the language by any of the four representations of regular languages, regular expressions, DFAs, NFAs, and regular grammars.

Language $\{a^n b^n \mid n \geq 0\}$ can be easily described by the non-regular grammar:

$$S \rightarrow \wedge / aSb.$$

So, a context-free grammar is a grammar whose productions are of the form :

$$S \rightarrow x$$

Where S is a non-terminal and x is any string over the alphabet of terminals and non-terminals. Any regular grammar is context-free. A language is context-free language if it is generated by a context-free grammar.

A grammar that is not context-free must contain a production whose left side is a string of two or more symbols. For example, the production $Sc \rightarrow x$ is not part of any context-free grammar.

Most programming languages are context-free. For example, a grammar for some typical statements in an imperative language might look like the following, where the words in bold face are considered to be the single terminals:

$$S \rightarrow \text{while E do S/ if E then S else S/\{SL\}/I: = E}$$

$$L \rightarrow SL/\wedge$$

$$E \rightarrow \ldots.\text{(description of an expression)}$$
$$I \rightarrow \ldots.\text{(description of an identifier).}$$

We can combine context-free languages by union, language product, and closure to form new context-free languages.

Definition: A context-free grammar, called a CFG, consists of three components:

1. An alphabet Σ of letters called terminals from which we are going to make strings that will be the words of a language.
2. A set of symbols called non-terminals, one of which is the symbols, start symbol.
3. A finite set of productions of the form

   One non-terminal $\rightarrow$ finite string of terminals and/or non-terminals.

Where the strings of terminals and non-terminals can consist of only terminals or of only non-terminals, or any combination of terminals and non-terminals or even the empty string.

The language generated by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions. A language generated by a CFG is called a context-free language.

Example 11: Find a grammar for the language of decimal numerals by observing that a decimal numeral is either a digit or a digit followed by a decimal numeral.

$$S \rightarrow D/DS$$
$$D \rightarrow 0/1/2/3/4/5/6/7/8/9$$

$S \Rightarrow DS \Rightarrow 7S \Rightarrow 7DS \Rightarrow 7DDS \Rightarrow 78DS \Rightarrow 780S \Rightarrow 780D \Rightarrow 780.$

Example 12: Let the set of alphabet A = {a, b, c}

Then, the language of palindromes over the alphabet A has the grammar:

$S \rightarrow aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \wedge$

For example, the palindrome abcba can be derived as follows:

$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abcba$

Example 13: Let the CFG is $S \rightarrow L \mid LA$

$A \rightarrow LA \mid DA \mid \wedge$
$L \rightarrow a \mid b \mid \ldots \mid Z$
$D \rightarrow 0 \mid 1 \mid \ldots \mid 9$

The language generated by the grammar has all the strings formed by a, b,c ….z, 0, 1,…..9.

We shall give a derivation of string a2b to show that it is an identifier.

$S \Rightarrow LA \Rightarrow aA \Rightarrow aDA \Rightarrow a2A \Rightarrow a2LA \Rightarrow a2bA \Rightarrow a2b$

Context-Free Language: Since the set of regular language is closed under all the operations of union, concatenation, Kleen star, intersection and complement. The set of context free languages is closed under union, concatenation, Kleen star only.

Union

Theorem 1: if L_1 and L_2 are context-free languages, then $L_1 U L_2$ is a context-free language.

Proof: If L_1 and L_2 are context-free languages, then each of them has a context-free grammar; call the grammars G_1 and G_2. Our proof requires that the grammars have no non-terminals in common. So we shall subscript all of G_1's non-terminals with a 1 and subscript all of G_2's non-terminals with a 2. Now, we combine the two grammars into one grammar that will generate the union of the two languages. To do this, we add one new non-terminal, S, and two new productions.

$S \rightarrow S_1$
$\mid S_2$

S is the starting non-terminal for the new union grammar and can be replaced either by the starting non-terminal for G_1 or for G_2, thereby generating either a string from L_1 or from L_2. Since the non-terminals of the two original languages are completely different, and once we begin using one of the original grammars, we must complete the derivation using only the rules from that original grammar. Note that there is no need for the alphabets of the two languages to be the same.

Concatenation

Theorem 2: If L_1 and L_2 are context-free languages, then L_1L_2 is a context-free language.

Proof : This proof is similar to the last one. We first subscript all of the non-terminals of G_1 with a 1 and all the non-terminals of G_2 with a 2. Then, we add a new nonterminal, S, and one new rule to the combined grammar:

$S \rightarrow S_1S_2$

S is the starting non-terminal for the concatenation grammar and is replaced by the concatenation of the two original starting non-terminals.

Kleene Star

Theorem 3: If L is a context-free language, then L^* is a context-free language.

Proof : Subscript the non-terminals of the grammar for L with a 1. Then add a new starting nonterminal, S, and the rules

$$S \rightarrow S_1S \mid \Lambda$$

The rule $S \rightarrow S_1S$ is used once for each string of L that we want in the string of L^*, then the rule $S \rightarrow \Lambda$ is used to kill off the S.

Intersection

Now, we will show that the set of context-free languages is not closed under intersection. Think about the two languages $L_1 = \{a^nb^nc^m \mid n,m \geq 0\}$ and $L_2 = \{a^mb^nc^n \mid n,m \geq 0\}$. These are both context-free languages and we can give a grammar for each one:

G_1:

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid \Lambda$$
$$B \rightarrow cB \mid \Lambda$$

G_2:

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \Lambda$$
$$B \rightarrow bBc \mid \Lambda$$

The strings in L_1 contain the same number of a's as b's, while the strings in L_2 contain the same number of b's as c's. Strings that have to be both in L_1 and in L_2, i.e., strings in the intersection, must have the same numbers of a's as b's and the same number of b's as c's.

Thus, $L_1 \cap L_2 = \{a^nb^nc^n \mid n \geq 0\}$. Using Pumping lemma for context-free languages it can be proved easily that $\{a^n b^n c^n \mid n \geq 0\}$ is not context-free language. So, the class of context-free languages is **not closed** under intersection.

Although the set is not closed under intersection, there are cases in which the intersection of two context-free languages is context-free. Think about regular languages, for instance. All regular languages are context-free, and the intersection of two regular languages is regular. We have some other special cases in which an intersection of two context-free languages is context, free.

Suppose that L_1 and L_2 are context-free languages and that $L_1 \subseteq L_2$. Then $L_2 \cap L_1 = L_1$ which is a context-free language. An example is EQUAL $\cap \{a^nb^n\}$. Since strings in

$\{a^nb^n\}$ always have the same number of a's as b's, the intersection of these two languages is the set $\{a^nb^n\}$, which is context-free.

Another special case is the intersection of a regular language with a non-regular context-free language. In this case, the intersection will always be context-free. An example is the intersection of $L_1 = a^+b^+a^+$, which is regular, with L_2= PALINDROME. $L_1 \cap L_2 = \{a^nb^ma^n \mid m,n \geq 0\}$. This language is context-free.

Complement

The set of context-free languages is not closed under complement, although there are again cases in which the complement of a context-free language is context-free.

Theorem 4: The set of context-free languages is not closed under complement.
Proof: Suppose the set is closed under complement. Then, if L_1 and L_2 are context-free, so are L_1' and L_2'. Since the set is closed under union, $L_1' \cup L_2'$ is also context-free, as is $(L_1' \cup L_2')'$. But, this last expression is equivalent to $L_1 \cap L_2$ which is not guaranteed to be context-free. So, our assumption must be incorrect and the set is not closed under complement.

Here is an example of a context-free language whose complement is not context-free. The language $\{a^nb^nc^n \mid n \geq 1\}$ is not context-free, but the author proves that the complement of this language is the union of seven different context-free languages and is thus context-free. Strings that are not in $\{a^nb^nc^n \mid n \geq 1\}$ must be in one of the following languages:

1. $M_{pq} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } p>q\}$ (more a's than b's)
2. $M_{qp} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } q>p\}$ (more b's than a's)
3. $M_{pr} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } s>r\}$ (more a's than c's)
4. $M_{rp} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } r>p\}$ (more c's than a's)
5. M = the complement of $a^+b^+c^+$ (letters out of order)

Using Closure Properties

Sometimes, we can use closure properties to prove that a language is not context-free. Consider the language our author calls DOUBLEWORD = $\{ww \mid w \in (a+b)^*\}$. Is this language context-free? Assume that it is. Form the intersection of DOUBLEWORD with the regular language $a^+ b^+ a^+ b^+$, we know that the intersection of a context-free language and a regular language is always context-free. The intersection of DOUBLEWORD and is $a^nb^ma^nb^m \mid n,m \geq 1\}$. But, this language is not context-free, so DOUBLEWORD cannot be context-free.

Think carefully when doing unions and intersections of languages if one is a superset of the other. The union of PALINDROME and $(a+b)^*$ is $(a+b)^*$, which is regular. So, sometimes the union of a context-free language and a regular language is regular. The union of PALINDROME and a^* is PALINDROME, which is context-free but not regular.

Now try some exercises:

Ex.12) Find CFG for the language over $\Sigma = \{a,b\}$.

(a) All words of the form

$a^x b^y a^z$, where x, y, z = 1,2,3… and y = 5x+7z

(b) For any two positive integers p and q, the language of all words of the form $a^x b^y a^z$, where x, y, z = 1, 2, 3… and y = px + qz.

4.4 PUSHDOWN AUTOMATA (PDA)

Informally, a pushdown automata is a finite automata with stack. The corresponding acceptor of context-free grammar is pushdown automata. There is one start state and there is a possibly empty-set of final states. We can imagine a pushdown automata as a machine with the ability to read the letters of an input string, perform stack operations, and make state changes.

The execution of a PDA always begins with one symbol on the stack. We should always specify the initial symbol on the stack. We assume that a PDA always begins execution with a particular symbol on the stack. A PDA will use three stack operations as follows:

(i) The **pop** operation reads the top symbol and removes it from the stack.

(ii) The **push** operation writes a designated symbol onto the top of the stack. For example, push (x) means put x on top of the stack.

(iii) The **nop** does nothing to the stack.

We can represent a pushdown automata as a finite directed graph in which each state (i.e., node) emits zero or more labelled edges. Each edge from state i to state j labelled with three items as shown in the *Figure 7*, where L is either a letter of an alphabet or $\wedge$, S is a stack symbol, and 0 is the stack operation to be performed.

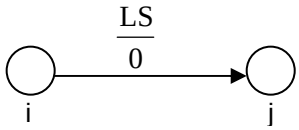


Figure 7: Directed graph

It takes fine pieces of information to describe a labelled edge. We can also represent it by the following 5-tuple, which is called a PDA instruction.

(i, L, S, 0, j)

An instruction of this form is executed as follows, where w is an input string whose letters are scanned from left to right.

If the PDA is in state i, and either L is the current letter of w being scanned or L = $\wedge$, and the symbol on top of the stack is S, then perform the following actions:

(1) execute the stack operation 0;
(2) move to the state j; and
(3) if L $\neq \wedge$, then scan right to the next letter of w.

A string is accepted by a PDA if there is some path (i.e., sequence of instructions) from the start state to the final state that consumes all letters of the string. Otherwise, the string is rejected by the PDA. The language of a PDA is the set of strings that it accepts.

Nondeterminism: A PDA is deterministic if there is at most one move possible from each state. Otherwise, the PDA is non-deterministic. There are two types of non-determinism that may occur. One kind of non-determinism occurs exactly when a state emits two or more edges labelled with the same input symbol and the same stack symbol. In other words, there are two 5-tuples with the same first three components. For example, the following two 5-tuples represent nondeterminism:

(i, b, c, pop, j)
(i, b, c, push(D), k).

The second kind of nondeterminism occurs when a state emits two edges labelled with the same stack symbol, where one input symbol is $\wedge$ and the other input symbol is not. For example, the following two 5-tuples represent non-determinism because the machine has the option of consuming the input letter b or cleaning it alone.

(i, $\wedge$, c, pop, j)
(i, b, c, push(D), k).

Example 14: The language $\{a^n b^n \mid n \geq 0\}$ can be accepted by a PDA. We will keep track of the number of a's in an input string by pushing the symbol Y onto the stack for each a. A second state will be used to pop the stack for each b encountered. The following PDA will do the job, where x is the initial symbol on the stack:

Figure 8: Pushdown automata

The PDA can be represented by the following six instructions:

(0, $\wedge$, X, nop, 2)
(0, a, X, push(Y), 0),
(0, a, Y, push(Y), 0),
(0, b, Y, pop,1),
(1, b, Y, pop,1),
(1, $\wedge$, X, nop,2).

This PDA is non-deterministic because either of the first two instructions in the list can be executed if the first input letter is a and X is on the top of the stack. A computation sequence for the input string aabb can be written as follows:

(0, aabb, X) start in state 0 with X on the stack,
(0, abb, YX) consume a and push Y,
(0, bb, YYX) consume a and push Y,
(1, b, YX) consume b and pop.
(0, $\wedge$, X) consume b and pop .
(2, $\wedge$, X) move to the final state.

Equivalent Forms of Acceptance:

Above, we defined acceptance of a string by a PDA in terms of final state acceptance. That is a string is accepted if it has been consumed and the PDA is in a final state. But, there is an alternative definition of acceptance called empty stack acceptance, which requires the input string to be consumed and the stock to be empty, with no requirement that the machine be in any particular state. The class of languages accepted by PDAs that use empty stack acceptance is the same class of languages accepted by PDAs that use final state acceptance.

Example 15: (An empty stack PDA): Let's consider the language $\{a^n b^n \mid n \geq 0\}$, the PDA that follows will accept this language by empty stack, where X is the initial symbol on the stack.

Figure 9: Pushdown automata

PDA shown in *Figure 9* can also be represented by the following three instructions:

$(0, a, X, \text{push } (X), 0)$,
$(0, \wedge, X, \text{pop}, 1)$,
$(1, b, X, \text{pop}, 1)$.

This PDA is non-determinstic. Let's see how a computation proceeds. For example, a computation sequence for the input string aabb can be as follows:

(0, aabb, X) start in state 0 with X on the stack
(0, abb, XX) consume a and push X
(0, bb, XXX) consume a and push X
(1, bb, XX) pop.
(1, b, X) consume b and pop
(1, $\wedge$, $\wedge$) consume b and pop (stack is empty)

Now, try some exercises.

Ex.13) Build a PDA that accepts the language odd palindrome.

Ex.14) Build a PDA that accepts the language even palindrome.

4.5 SUMMARY

In this unit we have considered the recognition problem and found out whether we can solve it for a larger class of languages. The corresponding accepter for the context-free languages are PDA's. There are some languages which are not context free. We can prove the non-context free languages by using the pumping lemma. Also in this unit we discussed about the equivalence two approaches, of getting a context free language. One approach is using context free grammar and other is Pushdown Automata.

4.6 SOLUTIONS/ANSWERS

Ex.1)

(a) $S \rightarrow S[S] \rightarrow [S] \rightarrow [\ ]$

(b) $S \rightarrow S[S] \rightarrow [S] \rightarrow [S[S\ ]] \rightarrow [[S] \rightarrow [[]]$.

Similarly rest part can be done.

Ex.2)

(a) $S \rightarrow aSb/aAb$
$A \rightarrow bA/b$

Ex.3)

(a) $S \rightarrow aSa/bSb/\wedge$

(b) $S \rightarrow aSa/bSb/a/b.$

Ex.4)

(a) $S \rightarrow ASB$ (type 2 production)
$S \rightarrow b$ (type 3 production)
$A \rightarrow aA$ (type 3 production)

So the grammar is of type 2.

(b) $S \rightarrow aSa$ (type 2 production)
$S \rightarrow bSb$ (type 2 production)
$S \rightarrow a$ (type 3 production)
$S \rightarrow b$ (type 3 production)
$S \rightarrow \wedge$ (type 3 production)

So the grammar is of type 2.

(c) Type 2.

Ex.5)

(a) $S \rightarrow AB$
$S \rightarrow aAb^5/\wedge$
$B \rightarrow b^7Ba/\wedge$

(b) $S \rightarrow AB$
$A \rightarrow aAb^p/\wedge$
$B \rightarrow b^qBa/\wedge$

Ex.6)

Suppose language is $\{wcw^T : w \in \{a,b\}^*\}$ then pda is

(0, a, x, push (a), 0), (0, b, x, push (b), 0),
(0, a, a, push (a), 0), (0, b, a, push (b), 0),
(0, a, b, push (a), 0), (0, b, b, push (b), 0),
(0, c, a, nop, 1), (0, c, b, nop, 1),
(0, c, x, nop, 1), (1, a, a, pop, 1),
(1, b, b, pop, 1), (1, $\wedge$, x, nop, 2),

Ex.7)

Language is $\{ww^T : w \in \{a,b\}^*\}$. Similarly as Ex 6.

4.7 FURTHER READINGS

1. *Elements of the Theory of Computation*, H.R. Lewis & C.H.Papadimitriou, PHI, (1981).

2. *Introduction to Automata Theory, Languages, and Computation* (*II Ed.)*, J.E. Hopcroft, R.Motwani & J.D.Ullman, Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language, and Computation*, J.E. Hopcroft and J.D. Ullman, Narosa Publishing House (1987).

UNIT 1 MODELS FOR EXECUTING ALGORITHMS – III : TM

Structure **Page Nos.**

1.0 INTRODUCTION

In unit 3 and unit 4 of block 4, we discussed two of the major *approaches* to modeling of computation viz. the automata/machine approach and linguistic/grammatical approach. Under grammatical approach, we discussed two models viz., Regular Languages and Context-free Languages.

Under automata approach, we discussed two *models* viz., Finite Automata and Pushdown Automata. Next, we discuss still more powerful automata for computation.

Turing machine (*TM*) is the next more powerful model of *automata approach* which recognizes more languages than Pushdown automata models do. Also **Phrase-structure model** is the corresponding grammatical model that matches Turing machines in computational power.

Notations: Turing Machine (**TM**), Deterministic Turing Machine, Non-Deterministic Turing Machine, Turing Thesis, Computation, Computational Equivalence, Configuration of TM, Turing-Acceptable Language, Turing Decidable Language, Recursively Enumerable Language, Turing Computable Function.

TM	:	Turing Machine
Γ	:	Set of tape symbols, includes #, the blank symbol
Σ	:	Set of input/machine symbols, does not include #
Q	:	the finite set of states of TM
F	:	Set of final states
a,b,c…	:	Members of Σ
σ	:	Variable for members of Σ
$\neg$x or $\bar{x}$	:	Any symbol of Σ *other than* x
#	:	The blank symbol
α, β, γ	:	Variables for String over Σ
L	:	Move the Head to the Left
R	:	Move the Head to the Right
q	:	A state of TM, i.e., $q \in Q$
s or q_0	:	The start/initial state

Halt or h:	The halt state. The same symbol h is used for the purpose of denoting halt state for all halt state versions of TM. And then h is not used for other purposes.
e or ∈ :	The empty string
$C_1 \vdash_M C_2$:	Configuration C_2 is obtained from configuration C_1 in *one* move of the machine M
$C_1 \vdash^* C_2$:	Configuration C_2 is obtained from configuration C_1 in *finite* number of moves.
$w_1 \underline{a} w_2$:	The symbol a is the symbol currently being scanned by the Head

Or

w_1 a w_2:
  ↑ The symbol a is the symbol currently being scanned by the Head.

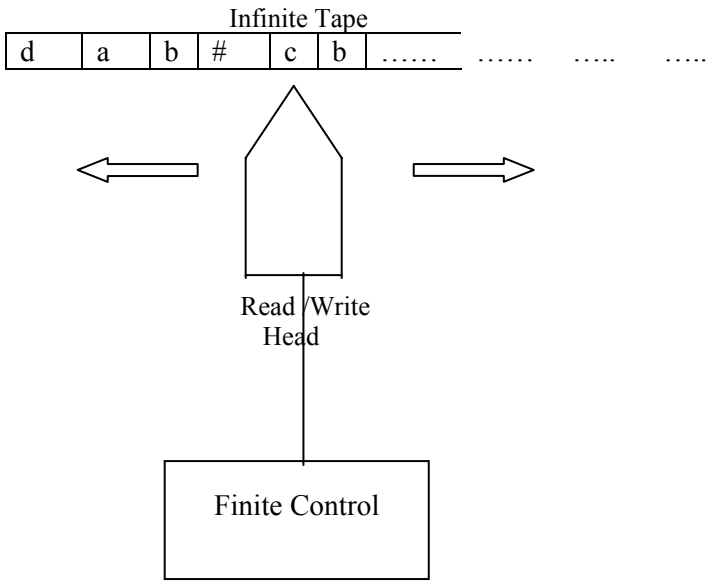
1.1 OBJECTIVES

After going through this unit, you should be able to:

- define and explain various terms mentioned under the title *key words* in the previous section;
- construct TMs for simple computational tasks;
- realize some simple mathematical functions as TMs; and
- apply modular techniques for the construction of TMs for more complex functions and computational tasks from TMs already constructed for simple functions and tasks.

1.2 PRELUDE TO FORMAL DEFINITION

In the next section, we will notice through a formal *definition of TM* that a TM is an *abstract* entity constituted of *mathematical objects* like sets and a (partial) function. However, in order to help our understanding of the subject-matter of TMs, we can *visualize* a TM as a physical computing device that can be represented as a *figure as shown in 1.2.1 below:*



TURING MACHINE
Figure: 1.2.1

Such a view, in addition to being more comprehensible to human beings, can be a quite *useful aid* in the design of TMs accomplishing some computable tasks, by allowing *informal* explanation of the various steps involved in arriving at a particular design. Without physical view and informal explanations, whole design process would be just a sequence of derivations of new formal symbolic expressions from earlier known or derived symbolic expressions — not natural for human understanding.

According to this view of TM, it consists of

(i) ***a tape***, with an end on the left but infinite on the right side. The tape is divided into squares or *cells*, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only *finitely* many cells of the tape that can contain non-blank symbols. The set of **tape symbols** is denoted by Γ.

As the very first step in the sequence of operations of a TM, **the input, as a finite sequence of the input symbols is placed in the left-most cells of the tape**. The set of **input symbols** denoted by Σ, does not contain the blank symbol #. However, during operations of a TM, a cell may contain a ***tape symbol*** which is not necessarily an input symbol.

There are versions of TM, to be discussed later, in which the tape may be infinite in both left and right sides — having neither left end nor right end.

(ii) ***a finite control***, which can be in any one of the finite number of states. *The states in TM can be divided in three categories viz.*,

(a) the ***Initial state,*** *the state of the control just at the time when TM starts its operations. The initial state of a TM is generally denoted by q_0 or s.*

(b) the ***Halt state, which*** *is the state in which TM stops all further operations.*

The halt state is generally denoted by h. The halt state is distinct from the initial state. Thus, a TM HAS AT LEAST TWO STATES.

(c) ***Other states***

(iii) ***a tape head*** *(or simply Head)*, is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The Head can ***read or scan*** the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including–

- the change of the symbol in the cell being scanned and/or
- change of its state and/or
- moving the head to the Left or to the Right. The control may decide not to move the head.

The course of action is called **a *move* of the Turing Machine. In other words, the move is a function of current state of the control and the tape symbol being scanned.**

In case the control decides for change of the symbol in the cell being *scanned*, then the *change is carried out by the head*. This change of symbol in the cell being scanned is called ***writing of the cell by the head***.

Initially, the head scans the left-most cell of the tape.

Now, we are ready to consider a **formal definition** of a Turing Machine in the next section.

1.3 TURING MACHINE: FORMAL DEFINITION AND EXAMPLES

There are a number of versions of a TM. We consider below *Halt State version* of formal definition a TM.

Definition: Turing Machine (*Halt State Version*)
A Turing Machine is a sextuple of the form (Q, Σ, ⌈, δ, q_o, h), where

(i) Q is the finite set of states,

(ii) Σ is the finite set of non-blank information symbols,

(iii) ⌈ is the set of tape symbols, including the blank symbol #

(iv) δ is the **next-move** *partial* function *from* $Q \times \lceil$ *to* $Q \times \lceil \times \{L, R, N\}$, where '***L***' denotes *the tape Head moves to the left adjacent cell*, '***R***' denotes *tape Head moves to the Right adjacent cell* and '***N*** ' denotes *Head does not move*, i.e., continues scanning the same cell.

In other words, for $q_i \in Q$ and $a_k \in \lceil$, there exists (*not necessarily always, because δ is a partial function*) some $q_j \in Q$ and some $a_l \in \lceil$ such that $\delta(q_i a_k) = (q_j, a_l, x)$, where x may assume any one of the values 'L', 'R' and 'N'.

The **meaning** of $\delta(q_i, a_k) = (q_j, a_l, x)$ is that if q_i is the current state of the TM, and a_k is cell currently under the Head, then TM writes a_l in the cell currently under the Head, enters the state q_j and the *Head moves to the right adjacent cell, if the value of x is R, Head moves to the left adjacent cell, if the value of x is L* and continues scanning the same cell, if the value of x is N.

(v) $q_0 \in Q$, is the initial/start state.

(vi) $h \in Q$ is the '*Halt State*', in which the machine stops any further activity.

Remark 1.3.1

Again, there are a number of variations in literature of even the above version of TM. For example, some authors allow at one time only one of the two actions viz., (i) writing of the current cell and (ii) movement of the Head to the left or to the right. However, this restricted version of TM can easily be seen to be computationally equivalent to the definition of TM given above, because one move of the TM given by the definition can be replaced by at most two moves of the TM introduced in the Remark.

In the next unit, we will discuss different versions of TM and issues relating to equivalences of these versions.

In order to illustrate the ideas involved, let us consider the following simple examples.

Example 1.3.2

Consider the Turing Machine $(Q, \Sigma, \lceil, \delta, q_o, h)$ defined below that erases all the non-blank symbols on the tape, where the sequence of non-blank symbols does not contain any blank symbol # in-between**:**

Q= $\{q_o, h\}$ $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \#\}$
and the next-move function δ is defined by the following table:

q: State	**σ : Input Symbol**	**δ (q, σ)**
q_0	a	$\{q_0, \#, R\}$
q_0	b	$\{q_0, \#, R\}$
q_0	#	{h, #, N}
h	#	ACCEPT

Next, we consider how to design a Turing Machine to accomplish some computational task through the following example. For this purpose, we need the definition.

A string Accepted by a TM

A string α over Σ is said to be accepted by a TM M = ($Q, \Sigma, \Gamma, \delta, q_0, h$) **if** when the string α is placed in the left-most cells on the tape of M and TM is started in the initial state q_0 **then** after a finite number of moves of he TM as determined by δ, Turing Machine is in state h (and hence stops further operations). The concepts will be treated in more details later on. Further, a *string is said to be rejected* if under the conditions mentioned above, the TM enters a state $q \neq h$ and scans some symbol x, then δ (q, x) is not defined.

Example 1.3.3

Design a TM which accepts all strings of the form $b^n d^n$ for $n \geq 1$ and rejects all other strings.
Let the TM M to be designed is given by M = ($Q, \Sigma, \Gamma, \delta, q_0, h$) with Σ = { b, d}. **The values of Q, Γ, δ, shall be determined by the design process explained below. However to begin with we take Γ = {b, d, #}.**

We *illustrate the design process* by considering various types of strings which are to be accepted or rejected by the TM.

As input, we consider only those strings which are over {b, d}. Also, it is assumed that, when moving from left, occurrence of first # indicates termination of strings over Γ.

Case I: When the given string is of the form $b^n d^m (b \mid d)^*$ for **$n \geq 1$**, **$m \geq 1$** as shown below for n = 2 m = 1

We are considering this particular type of strings, because, by taking simpler cases of the type, we can determine some initial moves of the required TM both for strings to be accepted and strings to be rejected.

B	b	d	-	-	-	-

Where '- ' denotes one of b, d or #

Initially, TM should *mark* left-most b. *The term **mark** is used here in this sense that the symbol is scanned matching with corresponding b or d as the case may be*. To begin with, the TM should attempt to *match, from the left, the first b to the d which is the first d after all b's have exhausted*. For this purpose, TM **should move right skipping over all b's. And after scanning the corresponding d, it should move left**, until we reach the b, which is the last b that was marked.

Next, TM should mark the b, if it exists, which is immediately on the right of the previously marked b. i.e., should mark the b which is the left-most b which is yet to be marked.

But, in order to recognize the yet-to-be-marked left-most b, we must change each of the b's, immediately on marking, to some other symbol say B. Also, for each b, we attempt to find the left-most yet-to-be-marked d. In order to identify the left-most yet-to-be-marked d, we should change each of the d's immediately on marking it, by some other symbol say D.

Thus we require two additional Tape symbols B and D, i.e., ⌈ = {b, d, B, D #}.

After one iteration of replacing one b by B and one d by D the tape would be of the form

B	b	D	-	-	-	-

and the tape Head would be scanning left-most b.

In respect of the states of the machine, we observe that in the beginning, in the initial state q_0, the cell under the Head is a b, and then this b is replaced by a B; and at this stage, **if we do not change the state then TM would attempt to change next b also to B** without matching the previous b to the corresponding d. But in order to recognize the form $b^n d^n$ of the string we do not want, in this round, other b's to be changed to B's before we have marked the corresponding d. Therefore,

$$\delta (q_0, b) = (q_1, B, R).$$

Therefore, the state must be changed to some new state **say q_1**. Also in order to locate corresponding d, the movement of the tape Head must be to the right. Also, in state q_1, the TM Head should skip over all b's to move to the right to find out the first d from left. Therefore, even on encountering b, we may still continue in state q_1. Therefore, we should have

$$\delta (q_1, b) = (q_1, b, R).$$

However, on encountering a d, the behaviour of the machine would be different, i.e., now TM would change the first d from left to D and *start leftward journey. Therefore, after a d is changed to D,* ***the state should be changed to say q_2.*** *In state q_2 we start leftward journey jumping over D's and b's*. Therefore

$$\delta (q_1, d) = (q_2, D, L) \text{ and}$$
$$\delta (q_2, D) = (q_2, D, L) \text{ and}$$
$$\delta (q_2, b) = (q_2, b, L).$$

In q_2, when we meet the first B, we know that none of the cells to the left of the current cell contains b and, if there is some b still left on the tape, then it is in the cell just to the right of the current cell. Therefore, we should move to the right and then if it is a b, **it is the left-most b on the tape and therefore the whole process should be repeated, starting in state q_0 again**.

Therefore, before entering b from the left side, TM should enter the initial state q_0. Therefore,

$$\delta (q_2, B) = (q_0, B, R).$$

For to-be-accepted type string, when all the b's are converted to B's and when the last d is converted to D in q_2, we move towards left to first B and then move to right in q_0 then we get the following transition:

from configuration

B	B	D	D	#	#

↑
q_2

to configuration

B	B	D	D	#	#

↑
q_0

Now we consider a special subcase of $b^n d^m (b \mid d)^*$, in which initially we have the following input

b	D	b	……..

Which after some moves changes to

B	D	b	

↑
q_0

The above string is to be rejected. But if we take $\delta (q_0, D)$ as q_0 then whole process of matching b's and d's will be again repeated and then even the (initial) input of the form

b	d	b	#	#

will be incorrectly accepted. In general, in state q_0, we encounter D, if all b's have already been converted to B's and corresponding d's to D's. Therefore, the next state of $\delta (q_0, D)$ cannot be q_0.

Let

$$\delta (q_0, D) = (q_3, D, R).$$

As explained just above, for a string of the to-be-accepted type, i.e., of the form $b^n d^n$, in q_3 we do not expect symbols b, B or even another d because then there will be more d's than b's in the string, which should be rejected.

In all these cases, strings are to be rejected. One of the ways of rejecting a string say s by a TM is first giving the string as (initial) input to the TM and then by not providing a value of δ in some state $q \neq h$, while making some move of the TM.

Thus the TM, not finding next move, stops in a state $q \neq h$. Therefore, the string is rejected by the TM.

Thus, each of $\delta (q_3, b)$, (q_3, B) and (q_3, D) is undefined

Further, in q_3, we skip over D's, therefore

$$\delta (q_3, D) = (q_3, D, R)$$

Finally when in q_3, if we meet #, this should lead to accepting of the string of the form $b^n d^n$, i.e., we should enter the state h. Thus,

$$\delta (q_3, \#) = (h, \#, N)$$

Next, we consider the cases *not* covered by $b^n d^m (b \mid d)^*$ with $n \geq 1, m \geq 1$ are. Such

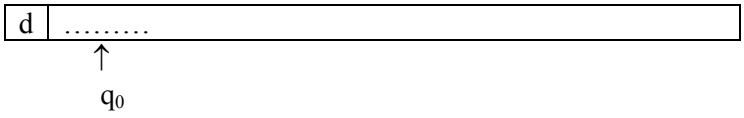
Case II when $n = 0$ but $m \neq 0$, i.e., when input string is of the form $d^m (b \mid d)^*$ for $m \neq 0$.

Case III when the input is of the form b^n #, $n \neq 0$

Case IV when the input is of the form # ……..

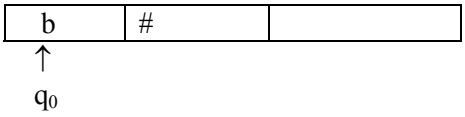
Now we consider the cases in detail.

Case II:

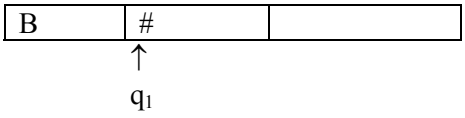


The above string is to be rejected, therefore, **we take $\delta(q_0, d)$ as undefined**

Case III: When the input is of the form b^n # #…#…, say



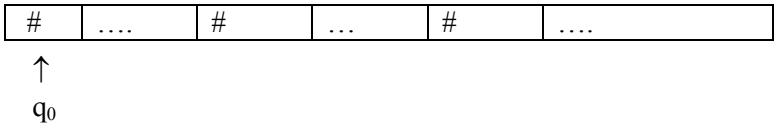
After one round we have



As the string is to be rejected, therefore,

$\delta(q_1, \#)$ is undefined

Case IV: When # is the left-most symbol in the input



As the string is to be rejected, therefore, we take **$\delta(q_0, \#)$ as undefined**

We have considered all possible cases of input strings over Σ = {b,d} and in which, while scanning from left, occurrence of the first # indicates termination of strings over Γ.

After the above discussion, the design of the TM that accepts strings of the form $b^n d^n$ and rejects all other strings over {b, d}***, may be summarized as follows:***

The TM is given by $(Q, \Sigma, \Gamma, \delta, q_0, h)$ where
$Q = \{q_0, q_1, q_2, q_3, h\}$
$\Sigma = \{ b, d\}$
$\Gamma = \{b, d, B, D, \#\}$

The next-move partial function δ is given by

	b	d	B	D	#
q_0	$\{q_1, B, R)$	*	*	(q_3, D, R)	*
q_1	$\{q_1, b, R)$	$\{q_2, D, L)$	*	$\{q_1, D, R)$	*
q_2	$\{q_2, b, L)$	*	(q_0, B, R)	$\{q_2, D, L)$	*
q_3	*	*	*	(q_3, D, R)	(h, #, N)
h	*	*	*	*	Accept

'*' Indicates the move is not defined.

Remark 1.3.4

In general, such lengthy textual *explanation* as provided in the above case of design of a TM, *is not given*. We have included such lengthy explanation, as the *purpose is to explain the very process of design.* In general, table of the type given above along with some supporting textual statements are sufficient as solutions to such problems. In stead of tables, we may give Transition Diagrams (*to be defined*).

Ex. 1) Design a TM that recognizes the language of all strings of even lengths over the alphabet {a, b}.

Ex. 2) Design a TM that accepts the language of all strings which contain aba as a sub-string.

1.4 INSTANTANEOUS DESCRIPTION AND TRANSITION DIAGRAMS

The following **differences** in the roles of tape and tape Head of **Finite** Automaton **(FA) and pushdown** Automaton **(PDA)** on one hand and in the roles of tape and tape head of **Tuning Machine** on other hand need to be noticed:

(i) The cells of the tape of an FA or a PDA are only read/scanned but are **never** changed/**written** into, whereas the cells of the tape of a TM **may be written** also.

(ii) The tape head of an FA or a PDA **always** moves from left to right. However, the tape head of a TM **can move in both** directions.

As a consequence of facts mentioned in (i) and (ii) above, we conclude that in the case of FA and PDA **the information in the tape cells already scanned do not play any role** in deciding future moves of the automaton, but in the case of a TM, the information contents of all the cells**, including the ones earlier scanned also play** a role in deciding future moves. This leads to the **slightly different definitions of configuration or Instantaneous Description (ID)** in the case **of a TM**.

1.4.1 Instantaneous Description

The **total configuration** or, for short just, **configuration** of a Turing Machine is the information in respect of:

(i) Contents of all the cells of the tape, starting from the left−most cell up to atleast the last cell containing a non-blank symbol and containing all cells upto the cell being scanned.

(ii) The cell currently being scanned by the machine and

(iii) The state of the machine.

Some authors use the term *Instantaneous Description* instead of *Total Configuration*.

Initial Configuration: The total configuration at the start of the (Turing) Machine is called the initial configuration.

Halted Configuration: is a configuration whose state component is the Halt state.

There are various *notations* used for denoting the total configuration of a Turing Machine.

Notation 1: *We use the notations, illustrated below through an example*:

Let the TM be in state q_3 scanning the symbol g with the symbols on the tape as follows:

#	#	b	d	a	f	#	g	h	k	#	#	#	#

Then one of the notations is

#	#	b	d	a	f	#	g	h	k	#	#	#	#
							↑						
							q_3						

Notation 2: However, the above being a two-dimensional notation, is sometimes inconvenient. Therefore the following linear notations are frequently used:

(q_3,##bdaf#,g,hk), in which third component of the above 4-component vector, contains the symbol being scanned by the tape head.

Alternatively, the configuration is also denoted by (q_3,## bdaf# $\underline{g}$ hk), where the symbol under the tape head is underscored but two last commas are dropped.

It may be noted that the sequence of blanks after the last non-blank symbol, is not shown in the configuration. The notation may be alternatively written (q_3, w, g, u) where w is the string *to the left* and u the string *to the right* respectively of the symbol that is currently being scanned.

In case g is the left-most symbol then we *use the empty string* e instead of w. Similarly, if g is being currently scanned and there is no non-blank character to the right of g then we use e, the empty string instead of u.

Notation 3: The next notation neither uses parentheses nor commas. Here the state is written just to the left of the symbol currently being scanned by the tape Head. Thus the configuration (q_3, ##bdaf#, g, h, k) is denoted as # # bdaf#**q_3**ghk
Thus if the tape is like

g	w	#	……
↑			
q_5			

then we may denote the corresponding configuration as (q_5, e, g, u). And, if the tape is like

a	b	c	g	#	#	…
			↑			
			q_6			

Then the configuration is (q_6, abc, g, e) or (q_6, abc $\underline{g}$) or alternatively as abcq_6g by the following notation.

1.4.2 Transition Diagrams

In some situations, **graphical** representation of the next-move (partial) function δ of a Turing Machine may give better idea of the behaviour of a TM in comparison to the **tabular** representation of δ.

A **Transition Diagram** of the next-move functions δ of a TM is a graphical representation consisting of a finite number of nodes and (directed) labelled arcs between the nodes. Each node represents a state of the TM and a label on an arc from one state (say p) to a state (say q) represents the information about the required **input symbol say x** for the transition from p to q to take place **and the action** on the part of the control of the TM. The action part consists of (i) the symbol say y to be written in the current cell and (ii) the movement of the tape Head.

Then the label of an arc is generally written as x/(y, M) where M is L, R or N.

Example 1.4.2.1

Let $M = \{Q, \Sigma, \Gamma, \delta, q_0, h\}$
Where $Q = \{ q_0, q_1, q_2, h\}$
$\Sigma = \{ 0, 1\}$
$\Gamma = \{0, 1, \#\}$
and δ be given by the following table.

	0	1	#
q_0	-	-	$(q_2, \#, R)$
q_1	$(q_2, 0, R)$	$(q_1, \#, R)$	$(h, \#, N)$
q_2	$(q_2, 0, L)$	$(q_1, 1, R)$	$(h, \#, N)$
h	-	-	-

Then, the above Turing Machine may be denoted by the Transition Diagram shown below, where we assume that q_0 is the initial state and h is a final state.

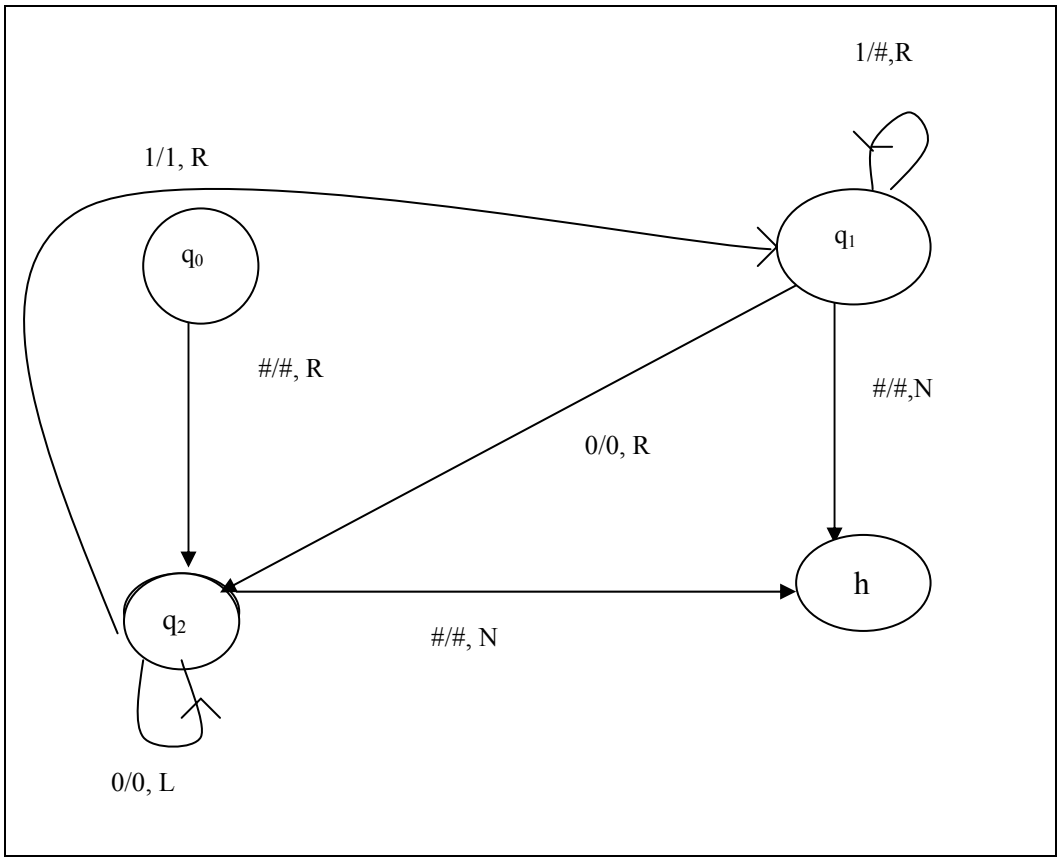


Figure: 1.4.2.1

Ex. 3) Design a TM M that recognizes the language L of all strings over {a, b, c} with
(i) number of a's = Number of b's = Number of c's and

(ii) if (i) is satisfied, the final contents of the tape are the same as the input, i.e., the initial contents of the tape are also the final contents of the tape, else rejects the string.

Ex. 4) Draw the Transition Diagram of the TM that recognizes strings of the form $b^n d^n$, $n \geq 1$ and was designed in the previous section.

Ex. 5) Design a TM that accepts all the language of all palindromes over the alphabet {a, b}. A palindrome is a string which equals the string obtained by reversing the order of occurrence of letters in it. Further, find computations for each of the strings (i) babb (ii) bb (iii) bab.

Ex. 6) Construct a TM that copies a given string over {a, b}. Further find a computation of the TM for the string aab.

1.5 SOME FORMAL DEFINITIONS

In the previous sections of the unit, we have used, without formally defining some of the concepts like *move*, *acceptance* and *rejection* of strings by a TM. In this section, we define these concepts formally

In the rest of the section we assume the TM under consideration is

$$\mathbf{M= (Q, \Sigma, \Gamma, \delta, q_0\ h)}$$

Definition: Move of a Turing Machine. *We give formal definition of the concept by considering three possible different types of moves,* viz.,

- '*move to the left',*
- '*move to the right*', and
- '*Do not Move'*.

For the definition and notation for Move, *assume the TM is in the configuration* $\mathbf{(q, a_1\ a_2 \ldots\ a_{i-1}, a_i, a_{i+1} \ldots a_n)}$

Case (i) $\delta(a_i, q) = \delta(b, p, L)$, for motion to left
Consider the following three subcases:

Case i(a) *if $i > 1$*, then the move is the activity of TM of going from the configuration

$(q, a_1\ a_2 \ldots\ a_{i-1}, \mathbf{a_i}, a_{i+1} \ldots a_n)$ to the configuration
$(p, a_1 \ldots a_{i-2}, \mathbf{a_{i-1}}, a_i\ a_{i+1} \ldots a_n)$ and is denoted as
$q, a_1\ a_2 \ldots\ a_{i-1}, \mathbf{a_i}, a_{i+1} \ldots a_n) \vdash_m (p, a_1 \ldots, a_{i-2}, \mathbf{a_{i-1}}, b, a_{i+1} \ldots a_n)$.

The suffix M, denoting the TM under consideration, may be dropped, if the machine under consideration is known from the context.

Case i(b) *if $i = 1$*, the move leads to hanging configuration, as TM is already scanning left-most symbol and attempts to move to the left, which is not possible. *Hence move is not defined*.

Case i(c) *when $i = n$ and b is the blank symbol #*, then the move is denoted as
$(q, a_1\ a_2 \ldots\ a_{n-1}, \mathbf{a_n}, e) \vdash (q, a_1\ a_2 \ldots\ a_{n-2}, a_{n-1}, \in, e)$.

Case (ii) $\delta(a_i, q) = \delta(b, p, R)$, for motion to the right

Consider the following two subcases:

Case ii(a) if i < n then the move is denoted as
$(q, a_1 \ldots a_{i-1}, \mathbf{a_i}, a_{i+1} \ldots a_n) \vdash (p, a_1, \ldots a_{i-1}\ b\ \mathbf{a_{i+1}}, a_{i+2} \ldots a_n)$

Case ii(b) *if i = n* the move is denoted as
$(q, a_1 \ldots a_{n-1}, \mathbf{a_n}, e) \vdash (p, a_1 \ldots, \#, e)$

Case (iii) $\delta(a_i, q) =$ **(b, p, 'No Move') when Head does not move.**
then the move is denoted as
$(q, a_1 \ldots a_{i-1}, \mathbf{a_i}, a_{i+1} \ldots a_n) \vdash (p, a_1 \ldots a_{i-1}, \mathbf{b}, a_{i+1} \ldots a_n)$

Definition: A configuration results (or is derived) from another configuration.

We illustrate the concept through an example based **on say Case (iii)** above of the *definition of 'move'*. In this case, we say the configuration $(p, a_1 \ldots a_{i-1}, \mathbf{b}, a_{i+1} \ldots a_n)$ **results in a single move or is derived in a single move** from the configuration $(q, a_1 \ldots a_{i-1}, \mathbf{a_i}, a_{i+1} \ldots a_n)$. **Also**, we may say that the ***move yields*** the configuration $(p, a_1 \ldots a_{i-1}, b, a_{i+1} \ldots a_n)$ **or *the configuration*** $(q, a_1 \ldots a_{i-1}, a_i, a_{i+1} \ldots a_n)$ yields the configuration $(p, a_1 \ldots a_{i-1}, b, a_{i+1} \ldots a_n)$ **in a single move**.

Definition: Configuration results in n Moves or finite number of moves.

If, for some positive integer n, the configurations $c_1, c_2 \ldots c_n$ are such that c_i results from c_{i-1} **in a single move, i.e.,**

$c_{i-1} \vdash c_i$ for i = 2, … n

then, we may say that *c_n results from c_1 **in n moves or a finite number of moves***. The fact is generally denoted as

$\mathbf{c_1} \vdash^{\mathbf{n}} \mathbf{c_n}$ **or** $\mathbf{c_1} \vdash^{*} \mathbf{c_n}$

The latter notation is the preferred one, because generally n does not play significant role in most of the relevant discussions.

The notation $c_1 \vdash^{*} c_n$ also **equivalently stands** for the statement that **c_1 yields c_n in finite number of steps.**

Definition: Computation

If c_0 is an *initial* configuration and for some n, the configurations $c_1, c_2, \ldots, c_n$ are such that $c_0, \vdash c_1 \vdash \ldots \vdash c_n$, then, the sequence of configurations $c_0, c_1 \ldots c_n$ constitutes a *computation.*

Definition: A string $\omega \in \Sigma^*$ acceptable by a TM

ω is said to be acceptable by TM M if $(q_0, \omega) \vdash^{*} (h, r)$ for $r \in \Gamma^*$
Informally, ω is acceptable by M, **if** when *the machine M is started in the initial state q_0 after writing ω on the leftmost part of the tape, **then**, if after finite number of moves, the machine M halts (i.e., reaches state h and of course, does not hang and does not continue moving for ever) with some string γ of tape symbols, **the original string ω is said to be accepted by the machine M.***

Definition: Length of Computation

If C_0 is initial configuration of a TM M and $C_0, C_1 \ldots, C_n$ is a computation, then n is called the length of the computation $C_0, C_1, \ldots C_n$.

Definition: Input to a Computation

In the initial configuration, the string, which is on that portion of the tape beginning with the first non-blank square and ending with the last non-blank square, is called input to the computation.

Definition: Language accepted by a TM

$M = (\theta, \Sigma, \Gamma, \delta, q_0, h)$, ***denoted by L(M),*** *and is defined as*

$L(M) = \{\omega \mid \omega \in \Sigma^*$ and if $\omega = a_1 \ldots a_n$ then

$(q_0, e, a_1, a, \ldots a_n) \vdash^*$

$(h, b_1 \ldots b_{j-1}, b_j, \ldots b_{j,+1} \ldots bn)$

for some $b_1 b_2 \ldots b_n \in \Gamma^*$

L(M), the language accepted by the TM M is the set of all finite strings ω over Σ which are accepted by M.

Definition: Turing Acceptable Language

A language L over some alphabet is said to be *Turing Acceptable Language*, if there exists a Turing Machine M such that L = L (M).

Definition: Turing Decidable Language

There are at least two alternate, but of course, equivalent ways of defining a Turing Decidable Language as given below:

Definition: A language L over Σ, i.e., $L \subseteq \Sigma^*$ is said to be Turing Decidable, if both the languages L and its complement $\Sigma^* \sim L$ are Turing acceptable.

Definition: A language L over Σ, i.e., $L \subseteq \Sigma^*$ is said to be Turing Decidable, if there is a function

$f_L: \Sigma^* \rightarrow \{\underline{Y}, \underline{N}\}$
such that for each $\omega \in \Sigma^*$

$$f_L(\omega) = \begin{cases} \underline{Y} & \text{if } \omega \in L \\ \underline{N} & \text{if } \omega \notin L \end{cases}$$

Remark 1.5.1

A very important fact in respect of Turing acceptability of a string (or a language) needs our attention. The fact has been discussed in details in a later unit about undecidability. However, we briefly mention it below.

For a TM M and an input string $\omega \in \Sigma^*$, even after a large number of moves we may not reach the halt state. However, from this we can neither conclude that 'Halt state will be reached in a finite number of moves' nor can we conclude that Halt state will not be reached in a finite number moves.

This raises the question of how to decide that an input string w is *not* accepted by a TM M.

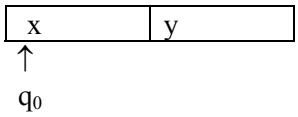
An input string w is said to be '*not accepted*' by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ if any of the following three cases arise:

(i) There is a configuration of M for which there is no next move i.e., there may be a state and a symbol under the tape head, for which δ does not have a value.

(ii) The tape Head is scanning the left-most cell containing the symbol x and the state of M is say q and δ (x, q) suggests a move to the 'left' of the current cell. However, there is no cell to the left of the left-most cell.

Therefore, move is not possible. The potentially resulting situation (can't say exactly configuration) is called **Hanging configuration**.

(iii) The TM on the given input w enters an infinite loop. For example, if configuration is as



and we are given

$\delta (q_0, x) = (q_1, x, R)$

and $\delta (q_1, y) = (q_0, y, L)$

Then we are in an **infinite loop.**

1.6 OBSERVATIONS

The concept of TM is one of the most important concepts in the theory of Computation. In view of its significance, we discuss a number of issues in respect of TMs through the following remarks.

Remark 1.6.1

Turing Machine is not just another computational model, which may be further extended by another still more powerful computational model. It is *not only the most powerful computational model* known so far *but also is conjectured* to be *the ultimate computational model*.

Turing Thesis: ***The power of any computational process is captured within the class of Turing Machines.***

It may be noted that Turing thesis is just a ***conjecture and not a theorem, hence, Turing Thesis*** can not be logically deduced from more elementary facts. *However*, the conjecture can be shown to be *false, if* a more powerful computational model is proposed that can recognize all the languages which are recognized by the TM model *and also* recognizes at least one more language that is not recognized by any TM.

In view of the unsuccessful efforts made in this direction since 1936, when Turing suggested his model, at least at present, it seems to be unlikely to have a more powerful computational model than TM Model.

Remark 1.6.2

The *Finite Automata* and *Push-Down Automata models* were used only as **accepting devices** for languages in the sense that the automata, when given an input string from a language, tells whether the string is *acceptable* or not. ***The Turing Machines are designed to play at least the following three different roles:***

(i) **As accepting devices for languages**, similar to the role played by FAs and PDAs.

(ii) **As a computer of functions.** In this role, a TM represents a particular function (*say the SQUARE function which gives as output the square of the integer given as input*). *Initial input* is treated as representing an **argument** of the function. And the (*final) string* on the tape when the TM enters the *Halt State* is treated as representative of the **value** obtained by an application of the function to the argument represented by the initial string.

(iii) **As an enumerator of strings of a language** that outputs the strings of a language, one at a time, in *some systematic order*, i.e., as a list.

Remark 1.6.3

Halt State of TM vs. set of Final States of FA/PDA

We have already briefly discussed the differences in the behaviour of TM on entering the Halt State and the behaviour of Finite Automata or Push Down Automata on entering a Final State.

A TM on entering the Halt State stops making moves and whatever string is there on the tape, is taken as output irrespective of whether the position of Head is at the end or in the middle of the string on the tape. However, an FA/PDA, while scanning a symbol of the input tape, if enters a final state, can still go ahead (*as it can do on entering a non-final state*) with the *repeated activities of moving to the right, of scanning the symbol under the head and of entering a new state etc. In the case of FA / PDA, the portion of string from left to the symbol under tape Head is accepted if the state is a final state and is not accepted if the state is not a final state of the machine.*

To be more clear we repeat: the only *difference* in the two situations when an FA/PDA enters a final state *and* when it enters a non-final state is that in the case of the *first situation*, the part of the input scanned so far is said to be **accepted/recognized**, *whereas* in the *second situation* the input scanned so far is said to be **unaccepted**.

Of course, in the Final State version of TM (discussed below), the Head is allowed movements even after entering a Final State. Some definite statement like 'Accepted/Recognized' can be made if, in this version, the TM is in Final State.

Remark 1.6.4

Final State Version of Turing Machine

Instead of the version discussed above, in which a particular state is designated as *Halt* State, some authors define TM in which a subset of the set of states Q is designated as *Set of Final State*s, which may be denoted by F. *This version is extension of Finite automata with the following changes, which are minimum required changes to get a Turing Machine from an FA.*

(i) The Head can move in both Left and Right directions whereas in PDA/FA the head moves only to the Right.

(ii) The TM, while scanning a cell, can both read the cell and also, if required, change the value of the cell, i.e., can *write in the cell*. In Finite Automata, the Head *only* can read the cell. It can be shown that the *Halt State* version of TM is *equivalent* to the *Final State* version of Turing Machine.

(iii) In this version, the TM machine halts only if in a given state and a given symbol under the head, no next move is possible. Then the (initial) input on the tape of TM, is unacceptable.

Definition: Acceptability of $\omega \in \Sigma^*$ in Final State Version

Let $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, F)$
be a TM in final state version. Then w is said to be acceptable if C_0 is the initial configuration with w as input string to M_1 and

$$C_0 \vdash^* C_n$$

is such that

$C_n = (p, \alpha, a, \beta)$

with p in F, set of final states, and a $\in \Gamma$, the set of tape symbols, and $\alpha, \beta \in \Gamma^*$

Equivalence of the Two Versions

We discuss the equivalence only informally. If in the Halt state version of a TM in stead of the halt state h, we take F= {h} then it is the Final state version of the TM. *Conversely*, if F= { $f_1, f_2, \ldots\ldots f_r$} is the set of final states then we should note the fact that in the case of acceptance of a string, a TM in final state version enters a final state *only once* and then halts with acceptance. Therefore, if we rename each of the final state as h, it will not make any difference to the computation of an acceptable or unacceptable string over Σ. Thus F may be treated as {h}, which further may be treated as just h.

1.7 TURING MACHINE AS A COMPUTER OF FUNCTIONS

In the previous section of this unit, we mentioned that a Turing Machine may be used as–

(i) A language Recognizer/acceptor

(ii) A computer of Functions

(iii) An Enumerator of Strings of a language.

We have already discussed the Turing Machine in the role of language accepting device. *Next, we discuss how a TM can be used as a computer of functions.*

Remark 1.7.1

For the purpose of discussing TMs as computers of functions, we make the following assumptions:

- A string ω over some alphabet say Σ will be written on the tape as #ω#, where # is the blank symbol.
- Also initially, the TM will be scanning the *right-most* # of the string #ω#.

Thus, the initial configuration, $(q_0, \#\omega\underline{\#})$ represents the starting point for the computation of the function with ω as input.

The assumption facilitates computation of composition of functions.

Though, most of the time, we require functions of one or more arguments having only integer values with values of arguments under the functions again as integers, yet, we consider functions with domain and codomain over *arbitrary* alphabet sets say Σ_0 and Σ_1 respectively, neither of which contains the blank symbol #.

Next we define what is meant by computation, using Turing Machine, of a function

$$f: \Sigma_0^* \rightarrow \Sigma_1^*$$

Definition: A function f: $f: \Sigma_0^* \rightarrow \Sigma_1^*$ is said to be ***Turing-Computable, or simply computable***, **if** there is a Turing Machine M = (Q, $\Sigma, \Gamma, \delta, q_0$, h), where Σ contains the following holds:

$(q_0, \#\omega\underline{\#},)$ $\vdash^*_m$ (h, # μ $\underline{\#}$,)

whenever $\omega \in \Sigma_0^*$ and $\mu \in \Sigma_1^*$ satisfying $f(\omega) = \mu$.

Remark 1.7.2

It may be noted that, **if** the string ω contains some symbols from the set $\Sigma - \Sigma_0$, i.e., symbols not belonging to the domain of f, **then** the TM may hang or may not halt at all.

Remark 1.7.3

Next, we discuss the case of **functions which require *k arguments***, where k may be any finite integer, greater than or equal to zero. For example,
the operation PLUS takes **two arguments** m and n and returns m + n.

The function f with the rule
f (x, y, z) = (2x + y) * z
takes **three arguments**.

The function C with rule
C ( ) = 17
takes **zero** number of arguments

Let us now discuss how to represent k distinct arguments of a function f on the tape. Suppose k = 3 and $x_1\ x_2$, $y_1\ y_2\ y_3$ and $z_1\ z_2$ are the three strings as three arguments of function f. **If** these three arguments are written on the tape as

#	x_1	x_2	y_1	Y_2	y_3	z_1	z_2	#

then the above tape contents may even be *interpreted as a single argument* viz., $x_1\ x_2, y_1\ y_2\ y_3\ z_1\ z_2$. Therefore, in order, **to avoid such an incorrect interpretation**, the arguments are separated by #. Thus, the above *three arguments* will be written on the tape as

#	x_1	x_2	#	y_1	Y_2	y_3	#	z_1	z_2	#

In general, if a function *f* takes k ≥ 1 arguments say $\omega_1, \omega_2, \ldots, \omega_k$ where each of these arguments is a string over Σ_0 (i.e., each ω_i belongs to Σ_0^*) and if $f(\omega_1, \omega_2, \ldots, \omega_k) = \mu$ for some $\mu \in \Sigma_1^*$; then **we say *f* is Turing Computable** if there is a Turing Machine M such that

$$(q_0, e, \#\omega_1\#\omega_2 \ldots \#\omega_k\#, e) \quad \vdash^*_M (h, e, \#\mu\#, e)$$

Also, **when f takes zero number of arguments and f( ) = μ then, we say f is computable**, if there is a Turing Machine M such that

$$(q_0, e, \#\,\underline{\#}, e) \quad \vdash^*_M (h, e, \#\mu\underline{\#}, e)$$

Remark 1.7.4

In stead of functions with countable, but otherwise arbitrary sets as domains and ranges, we consider only those functions, for each of which the domain and range is the *set of natural numbers*. This is not a serious restriction in the sense that any countable set can, through proper encoding, be considered as a set of natural numbers.

For natural numbers, there are various representations; some of the well-known representations are *Roman Numerals* (e.g., VI for six), *Decimal Numerals* (6 for six), *Binary Numerals* (110 for six). Decimal number system uses 10 symbols viz., 0, 1, 2, 3,4, 5, 6, 7, 8 and 9. Binary number system uses two symbols denoted by 0 and 1.
In the discussion of Turing Computable Functions, the *unary* representation described below is found useful. *The unary number system uses one symbol only:*

Let the symbol be denoted by I then the number with *name six is represented as* I I I I I I. In this notation, *zero is represented by empty/null string*. Any other number say twenty is represented in unary systems by writing the symbol I, twenty times. In order to facilitate the discussion, **the number n, in unary notation will be denoted by I^n in stead of writing the symbol I, n times**.

The *advantage of the unary representation* is that, in view of the fact that most of the symbols on the tape are input symbols and if the input symbol is just one, then the *next state* will generally be determined by **only the current state**, because the other determinant of the next state viz., tape symbol is most of the time the unary symbol.

We recall that for the set X, the notation X^* represents the set of all finite strings of symbols from the set X. Thus, any function *f* from the set of natural number to the set of natural numbers, *in the unary notation*, is a function of the form $f: \{I\}^* \rightarrow \{I\}^*$

Definition: The function f: N $\rightarrow$ N with f(n) = m for each n $\in$ N and considered as f: $\{I\}^* \rightarrow \{I\}^*$, with {I} a unary number system, **will be called Turing Computable function**, if a TM M can be designed such that M **starting** in *initial tape configuration*

I I ……. I

with *n consective I's* between the two #'s of the above string, **halts** in the following configuration

I I …… I

containing f(n) = m I's between the two #'s

The above idea may be further generalized to the functions of more than one integer arguments. For example, SUM of two natural numbers n and m takes two integer arguments and returns the integer (n + m). The initial configuration with the tape containing the representation of the two arguments say n and m respectively, is of the form

I I … I # I I ……I

where the string contains respectively n and m I's between respective pairs of #'s and Head scans the last #. **The function SUM will be Turing computable if** we can design a TM which when started with the initial tape configuration as given above, *halts* in the Tape configuration as given below:

I I … I I ….. I

where the above string contains n + m consecutive I's between pair of #'s.

Example 1.7.5

Show that the SUM function is Turing Computable.

The problem under the above-mentioned example may also be stated as: ***Construct a TM that finds the sum of two natural numbers.***

The following design of the required TM, *is not efficient* yet explains a number of issues about which a student should be aware while designing a TM for computing a function.

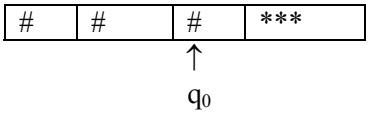
***Legal and Illegal Configurations for SUM function*:**

In order to understand the design process of any TM for a (computable) function in general and that of SUM in particular, let us consider the possible *legal as well as illegal* initial configuration types as follows:

*Note: in the following, the sequence '…' denotes any sequence of I's possibly empty and the sequences ' ***' denotes any sequence of Tape symbols possibly empty and possibly including #. Underscore denotes the cell being scanned.*

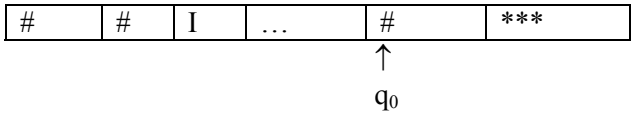
Legal initial configuration types:

Configuration (i)



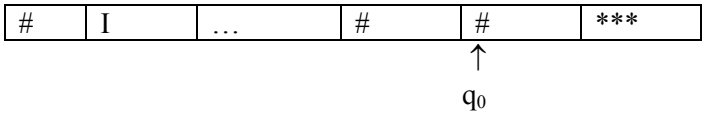
representing n = 0, m =0

Configuration (ii)



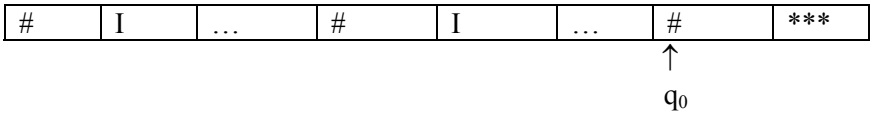
$n = 0, m \neq 0$

Configuration (iii)



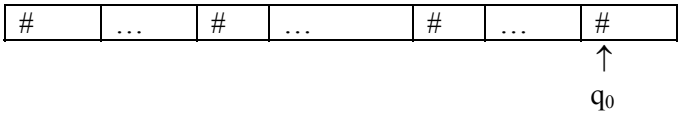
$n \neq 0, m = 0$

Configuration (iv)



$n \neq 0, m \neq 0$

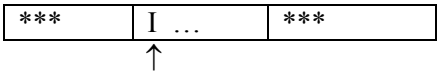
We treat the following configuration



containing two or more than two #'s to the left of # being scanned in initial configuration, as ***valid***, where '…' denotes sequence of I's only.

Some illegal initial configurations:

Configuration (v)

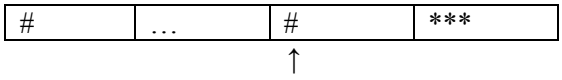


Where at least one of *** does not contain # and initially the Head is scanning an I or any symbol other than # . The configuration is invalid as it does not contain required number of #'s.

Configuration (vi), though is a special case of the above-mentioned configuration, yet it needs to be mentioned separately.

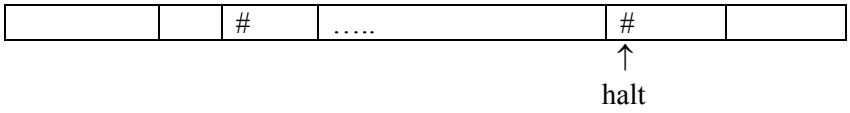
Left most symbol is I or any other non-# symbol
Where *** does not contain any #,

Configuration (vii)



Where *** does not contain # then the configuration represents only one of the natural numbers.

Also, in case of legal initial configurations, the final configuration that represents the result m + n should be of the firm.



with '…' representing exactly m + n I's.

Also in case of illegal initial configurations, the TM to be designed, should be in one of the following three situations indicating non-computability of the function with an illegal initial input, as explained at the end of Section 1.5:

(i) the TM has an infinite loop of moves;

(ii) the TM Head attempts to fall off the left edge (i.e., the TM has Hanging configuration); or

(iii) the TM does not have a move in a non-Halt state.

We use the above-mentioned description of initial configurations and the corresponding final configurations, in helping us to decide about the various components of the TM to be designed:

At this stage, we plan how to reach from an initial configuration to a final configuration. In the case of this problem of designing TM for SUM function, it is easily seen that for a legal initial configuration, we need to remove the middle # to get a final configuration.

(a) **Summing up** initially the machine is supposed to be in the initial state (say) q_0

(b) In this case of legal moves for TM for SUM function, first move of the Head should be to the *Left* only

(c) In this case, initially there are at least two more #'s on the left of the # being scanned. Therefore, to keep count of the #'s, *we must change state after scanning each #* . Let q_1, q_2 and q_3 be the states in which the required TM enters **after** scanning the three #'s

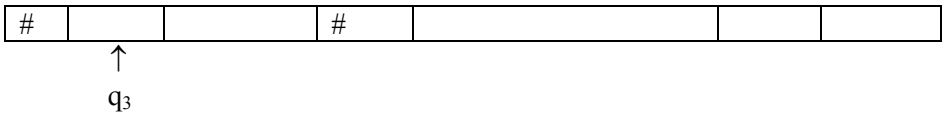
(d) In this case the movement of the Head, after scanning the initial # and also after scanning one more # on the left, should continue to move to the Left only, so as to be able to ensure the presence of third # also. Also, in states q_1 and q_2, the TM need not change state on scanning I.

Thus we have,

$\delta (q_0, \#) = (q_1, \#, L)$,

$\delta(q_1, \#) = (q_2, \#, L)$
and
$\delta(q_1, I) = (q_1, I, L)$, $\delta(q_2, I) = (q_2, I, L)$.

However, from this point onward, the Head should *start moving to the Right.*
$\therefore \delta(q_2, \#) = (q_3, \#, R)$.
Thus, at this stage we are in a configuration of the form.



For further guidance in the matter of the design of the required TM, we again look back on the legal configurations.

(e) In the configuration just shown above in q_3, if the symbol being scanned is # (as in case of *configuration (i) and configuration (ii))*, then the only action required is to skip over I's, if any, and halt at the next # on the right.

However, if the symbol being scanned in q_3 of the above configuration, happens to be an I (*as in case of configuration (iii) and configuration (iv))* then the actions to be taken, that are to be discussed after a while, have to be different.

But in both cases, movement of the Head has to be to the Right. Therefore, we need two new states say q_4 and q_5 such that

$\delta(q_3, \#) = (q_4, \#, R)$
(***the processing /scanning argument on the left, is completed***).

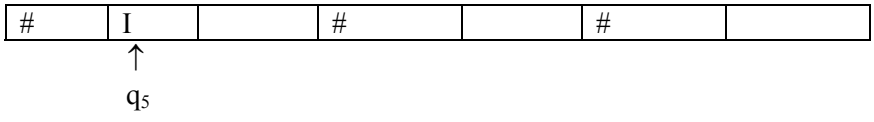
$\delta(q_3, I) = (q_5, I, R)$
(*the scanning of the argument on the left, is initiated*).

Taking into consideration the cases of the initial configuration (i) and configuration

(ii) we can further say that
$\delta(q_4, I) = (q_4, I, R)$
$\delta(q_4, \#) = (halt, \#, N)$

Next, taking into consideration the cases of initial configuration (iii) and configuration (iv) cases, we decide about next moves including the states etc., in the current state q_5.

We are in the following general configuration
(that subsumes the initial configuration (iii) and configuration (iv) cases)



Where the blank spaces between #'s may be empty or non-empty sequence of I's. Next landmark symbol is the next # on the right. Therefore, we may skip over the I's without changing the state i.e.,

$\delta(q_5, I) = (q_5, I, R)$

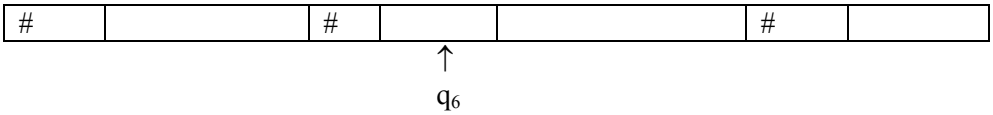
But we must change the state when # is encountered in q_5, **otherwise, the** next sequence of I's will again be skipped over and we will not be able to distinguish between configuration (iii) and configuration (iv) for further necessary action.

Therefore,

$\delta(q_5, \#) = (q_6, \#, R)$

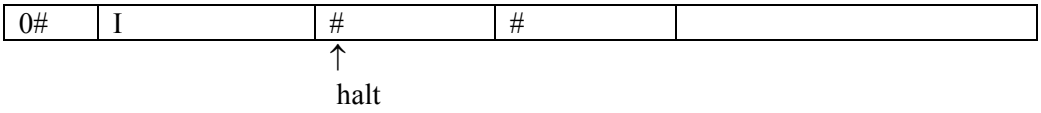
(*notice that, though at this stage, scanning of the argument on the left is completed, yet we can not enter in state q_4, as was done earlier, because in this case, the sequence of subsequent actions have to be different. In this case, the# in the middle has to be deleted, which is not done in state q_4*).

Thus, at this stage we have the general configuration as

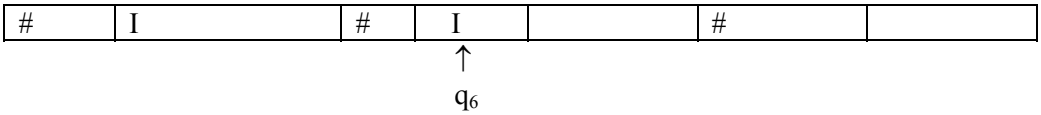


Next, in q_6, if the current symbol is a #, as is the case in configuration (iii), then we must halt after moving to the left i.e.,

$\delta(q_6, \#) = (\text{halt}, \#, L)$
we reach the final configuration



However, if we are in the configuration (iv) then we have



Then the following sequence of actions is required for deleting the middle #:

Action (i): To remove the # in the middle so that we get a continuous sequence of I's to represent the final result. For this purposes, we move to the left and replace the # by I. But then it will give one I more than number of I's required in the final result.

Therefore,

Action (ii): We must find out the rightmost I and replace the rightmost I by # and stop, i.e., enter halt state. In order to accomplish Action (ii) we reach the next # on the right, skipping over all I's and then on reaching the desired #, and then move left to an I over there. Next, we replace that I by # and halt.

Translating the above actions in terms of formal moves, we get

For Action (i)

$\delta(q_6, I) = (q_7, I, L)$
$\delta(q_7, \#) = (q_8, I, R)$

(*at this stage we have replaced the # in the middle of two sequences of I's by an I*)

For Action (ii)

$\delta(q_8, I) = (q_8, I, R)$
$\delta(q_8, \#) = (q_9, \#, L)$
$\delta(q_9, I) = (\text{halt}, \#, N)$

It can be verified that through above-mentioned moves, the designed TM does not have a next-move at some stage in the case of each of the illegal configurations.

Formally, the SUM TM can be defined as:

SUM = $(Q, \Sigma, \Gamma, \delta, q_0, h)$
where $Q = \{ q_0, q_1, \ldots q_{10}, \text{halt}\}$
$\Sigma = \{ I \}$
$\Gamma = \{ I, \# \}$
and

the next-move (partial) function δ is given by the Table

	I	#
q_0	-	$(q_1, \#, L)$
q_1	(q_1, I, L)	$(q_2, \#, L)$
q_2	(q_2, I, L)	$(q_3, \#, R)$
q_3	(q_5, I, R)	$(q_4, \#, R)$
q_4	(q_4, I, R)	(halt, #, N)
q_5	(q_5, I, R)	$(q_6, \#, R)$
q_6	(q_7, I, L)	(halt, #, L)
q_7	-	(q_8, I, R)
q_8	(q_8, I, R)	$(q_9, \#, L)$
q_9	(halt, #, N)	
halt	-	-

'–' indicates that *δ is not defined.*

Remark 1.7.6

As mentioned earlier also in the case of design of TM for recognizing the language of strings of the form $b^n d^n$, the design given above **contains too detailed explanation** of the various steps. The purpose is to explain the involved design **process** in fine details for better understanding of the students. However, **the students need not supply such details** while solving a problem of designing TM for computing a function. While giving the values of Q, Σ, Γ explicitly and representing δ either by a table or a transition diagram, we need to give **only** some supporting statements to help understanding of the ideas involved in the definitions of Q, Σ, Γ and δ.

Example 1.7.7

Construct a TM that multiplies two integers, each integer greater than or equal to zero (*Problem may also be posed as: show that multiplication of two natural numbers is Turing Computable*).

Informal Description of the solution:

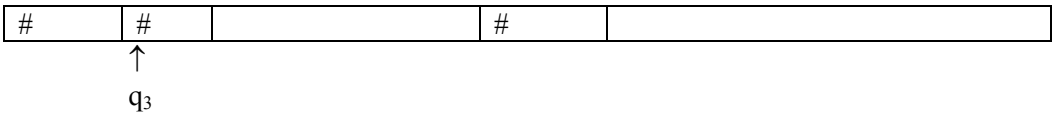
The legal and illegal configurations for this problem are the same as those of the problem of designing TM for SUM function. Also, the moves required to check the validity of input given for SUM function are the same and are repeated below:

$\delta(q_0, \#) = (q_1, \#, L)$
$\delta(q_1, \#) = (q_2, \#, L)$
$\delta(q_1, I) = (q_1, I, L)$
$\delta(q_2, \#) = (q_3, \#, R)$
$\delta(q_2, I) = (q_2, I, L)$

Next, we determine the rest of the behaviour of the proposed TM.

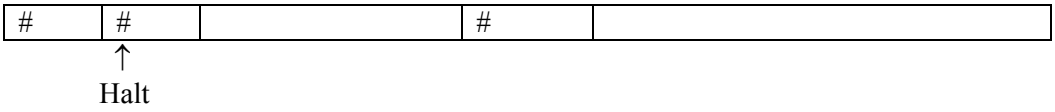
Case I

When n = 0 covering configuration (i) and configuration (ii) The general configuration is of the form



To get representation of zero, as, one of the multiplier and multiplic and is zero, the result must be zero. We should enter state say q_4 which skips all I's and meets the next # on the right.

Once the Head meets the required #, Head should move to the left replacing all I's by #'s and halt on the # it encounters so that we have the configuration

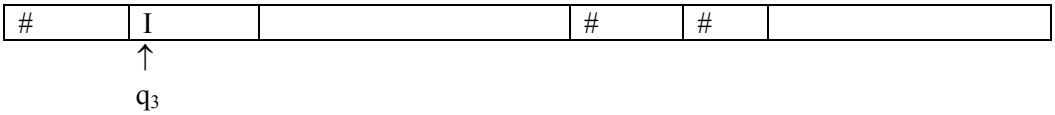


The moves suggested by the above explanation covering configuration (i) and configuration (ii) are:

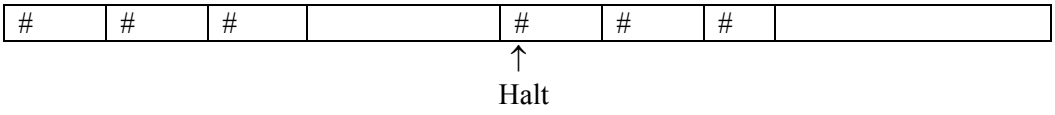
$\delta(q_3, \#) = (q_4, \#, R)$
$\delta(q_4, I) = (q_4, I, R)$
$\delta(q_4, \#) = (q_5, \#, L)$
$\delta(q_5, I) = (q_5, \#, L)$
$\delta(q_5, \#) = (\text{Halt}, \#, R)$

Case II

Covering configuration (iii), we have at one stage

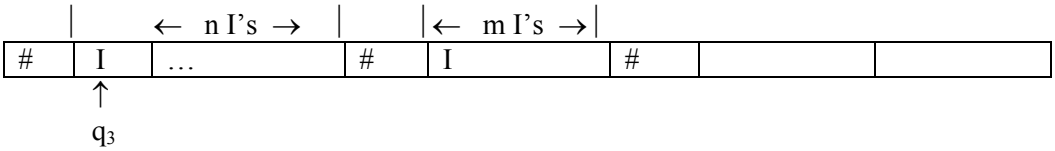


If we take $\delta(q_3, I) = (q_4, \#, R)$, then we get the following desired configuration in finite number of moves:

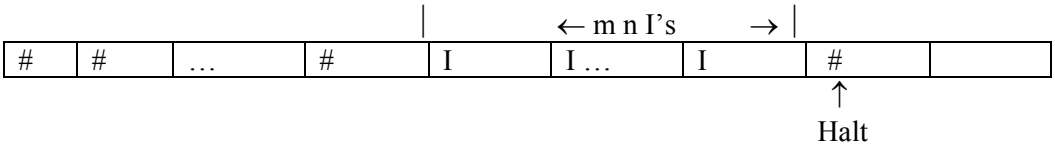


Case III

While covering the configuration (iv), At one stage, we are in the configuration



In this case, the final configuration is of the form



The strategy to get the representation for n m I's consists of the following steps:

(i) Replace the left-most I in the representation of n by # and then copy the m I's in the cells which are on the right of the # which was being scanned in the initial configuration. In the subsequent moves, copying of I's is initiated in the cells which are in the left-most cells on the right hand of last I's on the tape, containing continuous infinite sequence of #'s.

Repeat the process till all I's of the initial representation of n, are replaced by #. At this stage, as shown in the following figure, the tape contains m I's of the initial representation of the integer m and additionally n.m I's. Thus the tape contains m extra #'s than are required in the representation of final result. Hence, we replace all I's of m by #'s and finally skipping over all I's of the representation of (n . m) we reach the # which is on the right of all the (n . m) I's on the tape as required.

Alternatively: In stead of copying n times of the m I's, we copy only (n–1) times to get the configuration

#		#	I	#	I ….. I	#	I	……	I	#	

$|\leftarrow$ m I's $\rightarrow|$ $|\leftarrow$ ((n–1).m) I's $\rightarrow|\uparrow$

Then we replace the # between two sequences of I's by I and replace the right-most I by # and halt.

The case of illegal initial configurations may be handled on similar lines as were handed for SUM Turing machine

Remark 1.7.8

The informal details given above for the design of TM for multiplication function are acceptable as complete answer/solution for any problem about design of a Turing Machine. However, if more detailed formal design is required, the examiner should explicitly mention about the required details.

Details of case (iii) are not being provided for the following reasons:

(i) Details are left as an exercise for the students

(ii) After some time we will learn how to construct more complex machines out of already constructed machines, starting with the construction of very simple machines. One of the simple machines discussed later is a ***copying machine*** which copies symbols on a part of the tape, in other locations on the tape.

Ex. 7) Design a TM to compute the binary function MONUS (or also called PROPER SUBTRACTION) defined as follows:

Monus : $N \times N \rightarrow N$

(*Note 0 also belongs to N*)

such that

$$\text{monus (m, n)} = \begin{cases} m-n & \text{if } m \geq n \\ 0 & \text{else} \end{cases}$$

Ex.8) To compute the function n (mod 2)

Let if f denotes the function, then

f: $N \to \{0, 1\}$
is such that

$$f(n) = \begin{cases} 0 & \text{if } n \text{ is even,} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

1.8 SUMMARY

In this unit, after giving informal idea of what a Turing machine is, the concept is formally defined and illustrated through a number of examples. Further, it is explained how TM can be used to compute mathematical functions. Finally, a technique is explained for designing more and more complex TMs out of already designed TMs, starting with some very simple TMs.

1.9 SOLUTIONS/ANSWERS

Ex. 1)

The transition diagram of the required TM is as shown below:

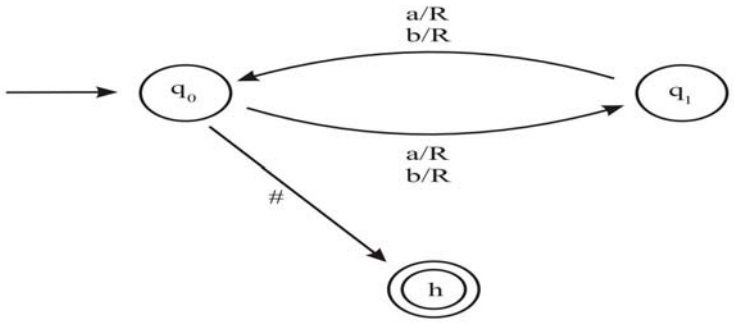


Figure: 1.9.1

The required TM M = (Q, Σ, Γ, δ, q_0, h) with

Q = {q_0, q_1, h}
Σ = {a, b} and Γ = {a, b, #}.

The next move function δ is given by the transition diagram above. If the input string is of even length the TM reaches the halt state h. However, if the input string is of odd length, then TM does not find any next move in state q_1 indicating rejection of the string.

Ex. 2)

The transition diagram of the required TM is as shown below:

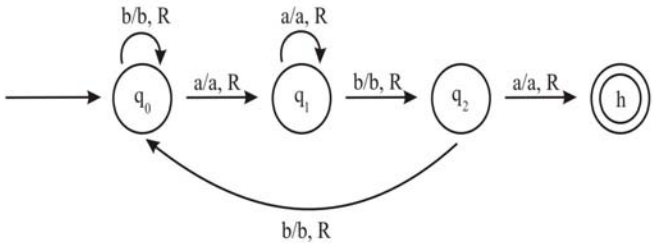


Figure: 1.9.2

The required TM M = (Q, Σ, Γ, δ, q_0, h) with

$Q = \{q_0, q_1, q_2, h\}$
$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function is given by the transition diagram above.
The transition diagram almost explains the complete functioning of the required TM. However, it may be pointed out that, if a string is not of the required type, then the blank symbol # is encountered either in state q_0 or in state q_1 or in state q_2. As there is no next move for $(q_0, \#)$, $(q_1, \#)$ or $Q(q_2, \#)$, therefore, the string is rejected.

Ex. 3)

The transition diagram of the required TM is as shown below:

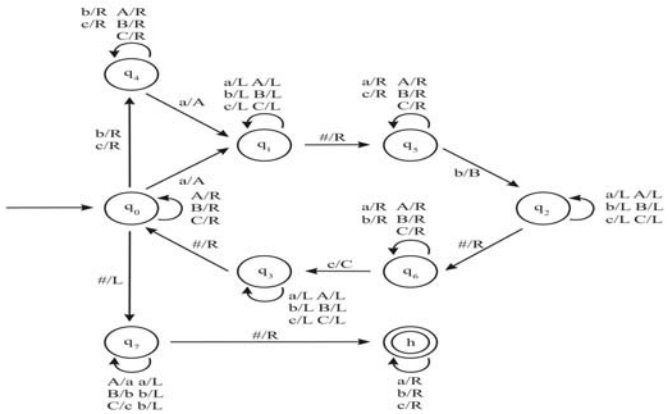


Figure: 1.9.3

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, h\}$
$\Sigma = \{a, b, c\}$ and $\Gamma = \{a, b, c, A, B, C, \#\}$ δ is shown by the diagram.

The design strategy is as follows:

Step I: While moving from left to right, we find the first occurrence of **a** if it exists. If such an **a** exists, then we replace it by A and enter state q_1 either directly or after skipping b's and c's through state q_4.

In state q_1, we move towards left skipping over all symbols to reach the leftmost symbol of the tape and enter state q_5.

In q_5, we start searching for b by moving to the right skipping over all non-blank symbols except b and if such b exists, reach state q_2.
In state q_2, we move towards left skipping over all symbols to reach the leftmost symbol of the tape and enter q_6.

In q_6, we start searching for c by moving to the right skipping over all non-blank symbols except c and if such c exists, reach state q_3.

In state q_2, we move towards left skipping all symbols to reach the leftmost symbol of the tape and enter state q_0.

If in any one of the states q_4, q_5 or q_6 no next move is possible, then reject the string.
Else repeat the above process till all a's are converted to A's, all b's to B's and all c's to C's.

Step II: is concerned with the restoring of a's from A's, b's from B's and c's from C's, while moving from right to left in state q_7 and then after successfully completing the work move to halt state h.

Ex. 4)

The Transition Diagram of the TM that recognizes strings of the form $b^n d^n$, $n \geq 1$ and designed in the previous section is given by the following *Figure*.

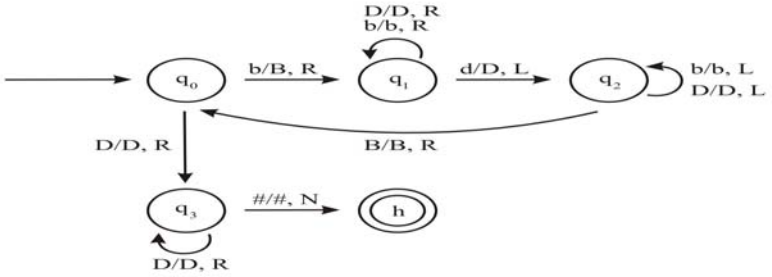


Figure: 1.9.4

Ex. 5)

The transition *Figure* of the required TM is as shown below.

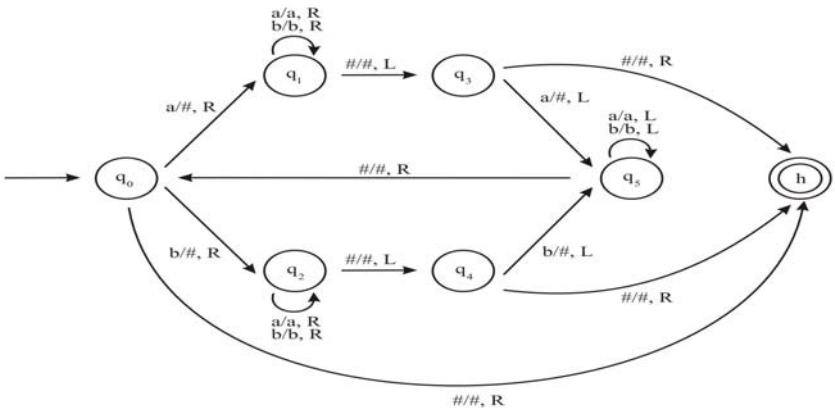


Figure: 1.9.5

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, h\}$
$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

The next move function is given by the transition diagram above.

The proposed TM functions as follows:

(i) In state q_0, at any stage if TM finds the blank symbol then TM **has found a palindrome of even length**. Otherwise, it notes the symbol being read and attempts to match it with last non-blank symbol on the tape. **If the symbol is a, the TM replaces it by # goes to state q_1,** in which it skips all a's and b's and on #, the TM from q_1 will go to q_3 to find a matching a in last non-blank symbol position. **If a is found**, TM goes to q_5 replace a by #. However, if b is found then TM has no more indicating the string is not a palindrome. However, if in state q_2 only #'s are found, then it indicates that the previous 'a' was the middle most symbol of the **given string indicating palindrome of odd length.**

Similar is the case when b is found in state q_0, except that the next state is q_2 in this case and roles of a's and b's are interchanged in the above argument.

(ii) The fact of a string not being a palindrome is indicated by the TM when in state q_3 the symbol b is found or in state q_4 the symbol a is found. **The initial configuration is $q_0$babb.**

The required computations are:

(i) $q_0babb \# q_2babb \vdash \#aq_2bb \vdash \#abbq_2\# \vdash \#abq_4b \vdash \#aq_5b\# \vdash \#q_5ab \vdash$

$q_5\#ab \vdash \#q_0ab \vdash \#\#q_1b \vdash \#bq_1\# \vdash \#\#q_3b,$

As there is no move in state q_3 on b, therefore, string is not accepted.

(ii) The initial configuration is $q_0$bb. Consider the computation:

$q_0bb \vdash \#q_2b \vdash \#bq_2 \vdash \#q_4b\# \vdash q_5\#\#\# \vdash q_0\# \vdash h\#$

(*We may drop #'s in the rightmost positions*).

(iii) The initial configuration is $q_0$bab. Consider the computation:

$q_0bab \vdash \#q_2ab \vdash^* \#aq_4b \vdash \#q_5a\# \vdash q_5\#\# \vdash \#q_0\# \vdash h\#$

(Note $\vdash^*$ denotes sequence of any finite number of $\vdash$).

Ex.6)

The transition diagram of the required TM is as shown below.

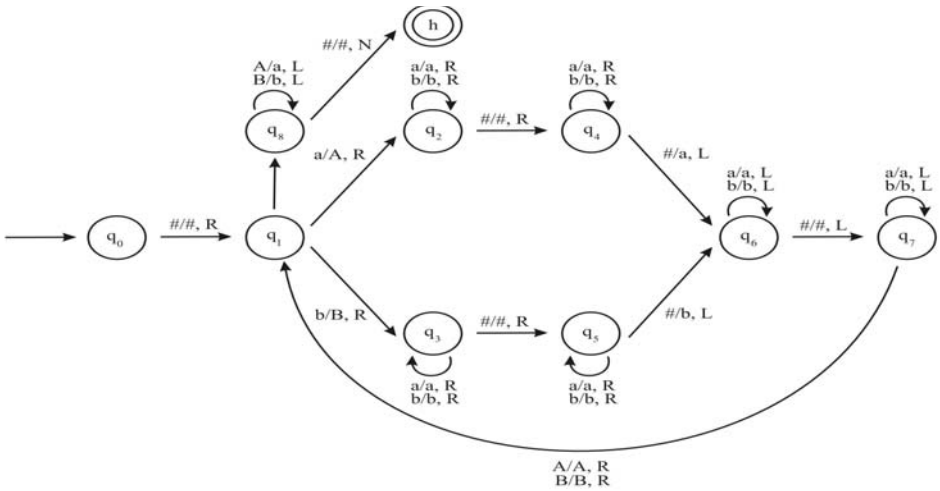


Figure: 1.9.6

The required TM M = (Q, Σ, Γ, δ, q_0, h) with
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, h}
Σ = {a, b} and Γ = {a, b, #}.

The next move function is given by the transition diagram above.

In the solution of the problem, we can deviate slightly from our convention of placing the input string on the left-most part of the tape. **In this case, we place # in the leftmost cell of the tape followed by the** input string.

Therefore, in the beginning in the initial state q_0, the TM is scanning # in stead of the first symbol of the input.
Before we outline the functioning of the proposed TM let us know that for the input string aab is placed on the tape as

#	a	A	b	#	#	***

and for the input, output on the tape is of the form

#	a	a	b	#	A	a	b	#	#	***

Outline of the functioning of the proposed TM

The TM in state q_1 notes the leftmost a or b, replaces it by A or B respectively and copies it in the next available # (the first # on the right is left as marker and is not taken as available). If the symbol in the state q_1 is a, then TM while skipping symbols passes through state q_2 and reaches q_4. However, if the symbol in state q_1 is b, then TM while skipping symbols passes through state q_3 and reaches state q_5. Then TM copies the symbol and reaches the state q_6. Next, TM starts its leftward journey skipping over a's, b's, A's, B's and # and meets A or B in q_7. At this stage, TM goes to state q_1. Then repeats the whole process until the whole string is copied in the second part of the tape.

But, in this process original string of a's and b's is converted to a string of A's and B's. At this stage TM goes from q_1 to state q_8 to replace each A by a and each B by b. This completes the task.

The Computation of the TM on input aab

The initial configuration is q_0#abb. Therefore, the computation is
q_0#abb $\vdash$ #$q_1$abb $\vdash$ #A$q_2$ab

$\vdash$ #Aaq_2b $\vdash$ #Aabq_2#

$\vdash$ #Aab#q_4 $\vdash$ #Aabq_5#a

$\vdash$ #Aabq_6b#a

$\vdash$ #A$q_6$ab#a $\vdash$ #q_6Aab#a

$\vdash$ #Aqab#a
(*At this point whole process is repeat and, therefore, we use* $\vdash^*$*, representing a finite number of* $\vdash$*)*

$\vdash^*$ #AAq_0b#aa

$\vdash^*$ #AABq_0b#aab
At this stage TM enters state q_7.

$\vdash$ #AAq_7B#aab

$\vdash$ #Aq_7Ab#aab

$\vdash$ #q_7Aab#aab

$\vdash$ q_7#Aab#aab

$\vdash$ h#aab#aab

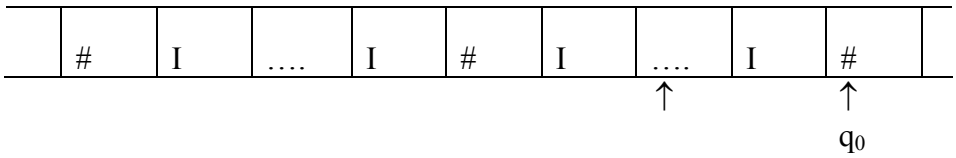
Ex.7)

In respect of the design of the TM $(Q, \Sigma, \Gamma, \delta, q_0, h)$, where
$\Sigma = \{ I \}$ $\Gamma = \{ I, \# \}$ where we made the following observations:

Observation 1: General form of the tape is

There are three significant positions of #, which need to be distinguished viz., right-most # on left of I's, middle #, middle # and left-most # on the right of I's. Therefore, there should be change of state on visiting each of these positions of #.

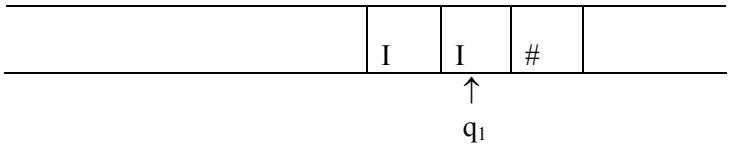
Observation 2: Initial configration is



and as observed above

$\delta (q_0, \#) = (q_1, \#, L)$

The following forms of the tape



and

guide us to moves

$\delta (q_1, I) = (q_2, \#, L)$

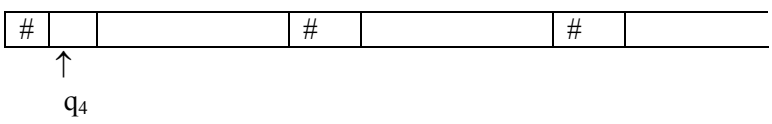
change of state is essential else other I's will also be converted to #'s,

$\delta (q_1, \#) = (\text{halt}, \#, N)$

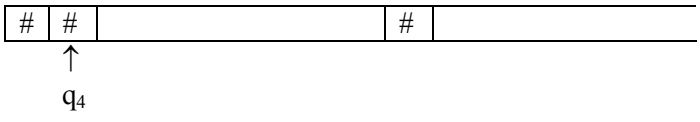
Observation 3: The moves are guided by principle that convert the left-most I to # on the right side the corresponding right-most I to # on the left-side

$\delta (q_2, I) = (q_2, I, L)$
$\delta (q_2, \#) = (q_3, \#, L)$
$\delta (q_3, I) = (q_3, I, L)$
$\delta (q_3, \#) = (q_4, \#, R)$

(We have reached the right-most # on the left of all I's as shown below)



If we have configration of the form



then it must have resulted from initial configuration in which m < n represented by say

#	I	I	#	I	I	I	#

↑

q_4

Therefore, we must now enter a state say q_7 which skips all I's on the right and then halts

Therefore

$\delta(q_4, \#) = (q_7, \#, R)$
$\delta(q_7, I) = (q_7, I, R)$
$\delta(q_7, \#) = (\text{halt}, \#, N)$

Next, we consider $\delta(q_4, I)$
$\delta(q_4, I) = (q_5, \#, R)$
(*state must be changed otherwise, all I's will be changed to #'s*)

$\delta(q_5, I) = (q_5, I, R)$
$\delta(q_5, \#) = (q_6, \#, R)$
(*the middle # is being crossed while moving from left to right*)

$\delta(q_6, I) = (q_6, I, R)$
$\delta(q^6, \#) = (q_0, \#, N)$
(the left-most # on right side is scanned in q_6 to reach q_0 so that whole process may be repeated again.)

Summarizing the above moves the transition table for δ function is given by

	I	#
q_0		$(q_1, \#, L)$
q_1	$(q_2, \#, L)$	(halt, #,L)
q_2	(q_2, I, L)	$(q_3, \#, L)$
q_3	(q_3, I, L)	$(q_4, \#, L)$
q_4	$(q_5, \#, R)$	$(q_7, \#, R)$
q_5	(q_5, I, R)	$(q_6, \#, R)$
q_6	(q_6, I, R)	$(q_6, \# R)$
q_7	(q_7, I, R)	(halt, #, N)
Halt	-	-

Ex.8)

By our representation conventions, the initial configuration is as follows:

#	I	. . .	I	#	#	…

↑

q_0

1 44 2 4 43
n I's

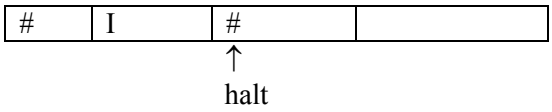
If **n is even,** then f (n) = 0 which further is represented by final configuration

#	#	

↑

halt

If n is odd, then f(x) = 1 which is represented by f (n) = 1 which is represented by a final configuration of the form,



The strategy of reaching from initial configuration to a final configuration is that after scanning even number of I's we enter state q_2 and after scanning odd number of I's, we enter state q_1 and then take appropriate action, leading to the following (partial) definition of transition function δ:

$$\delta(q_0, \#) = (q_2, \#, L)$$
$$\delta(q_2, I) = (q_1, \#, L)$$
$$\delta(q_2, \#) = (\text{halt}, \#, N)$$
$$\delta(q_1, I) = (q_2, \#, L)$$
$$\delta(q_1, \#) = (q_3, \#, R)$$
$$\delta(q_3, \#) = (\text{halt}, I, R)$$

For the transition

$\delta(q_i, a_k) = (q_j, a_l, m)$, the sequence of actions is as follows: First a_l is written in the current cell so far containing a_k. Then movement of tape head is made to left, to right or 'no move' respectively according as the value of m is L, R or N. Finally the state of the control changes to q_j.

The transition function δ for the above computation is

δ	#	I
q_0	$(q_2, \#, L)$	$(q_1, \#, L)$
q_1	$(q_3, \#, R)$	$(q_2, \#, L)$
q_2	(halt, #, N)	$(q_1, \#, L)$
q_3	(halt, I, R)	-
halt	-	-

The students are advised to make transition diagram of the (partial) function defined by the above table.

1.10 FURTHER READINGS

1. *Elements of the Theory of Computation,* H.R. Lewis & C.H.Papadimitriou: PHI, (1981).
2. *Introduction to Automata Theory, Languages, and Computation (II Ed.)* J.E. Hopcroft, R.Motwani & J.D.Ullman: Pearson Education Asia (2001).
3. *Introduction to Automata Theory, Language, and Computation,* J.E. Hopcroft and J.D. Ullman: Narosa Publishing House (1987).
4. *Introduction to Languages and Theory of Computation,* J.C. Martin, Tata-McGraw-Hill (1997).

UNIT 2 ALGORITHMICALLY UNSOLVABLE PROBLEMS

Structure **Page Nos.**

2.0 INTRODUCTION

In this unit, we discuss issues and problems that exhibit the limitations of computing devices in solving problems. *We also prove the undecidability of the halting problem.* It is related to Gödel's Incompleteness Theorem which states that there is no system of logic strong enough to prove all true sentences of number theory.

In addition, we will discuss a number of other problems, which though can be formulated properly, yet are not solvable through any computational means. And we will *prove* that such problems cannot be solved no matter what language is used, what machine is used, and how much computational resources are devoted in attempting to solve the problem etc.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- show that Halting Problem is uncomputable/unsolvable/undecidable;
- to explain the general technique of *Reduction* to establish other problems as uncomputable;
- establish unsolvability of many unsolvable problems using the technique of reduction;
- enumerate large number of unsolvable problems, including those about Turing Machines and about various types of grammars/languages including context-free, context-sensitive and unrestricted etc.

2.2 DECIDABLE AND UNDECIDABLE PROBLEMS

A function g with domain D is said to be computable if there exists some Turing machine

$M = (Q, \Sigma, T, \delta, q_0, F)$ such that
$q_0\, w \mid\!\!-^{*} q_f\, g(w)$, $q_f \in F$, for all $w \in D$.

where,

q_0 ω denotes the initial configuration with left-most symbol of the string ω being scanned in state q_0 **and** q_f g(ω) denotes the final c.

A function is said to be uncomputable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.

For some problems, we are interested in simpler solution in terms of "yes" or "no". For example, we consider problem of context free grammar i.e., for a context free grammar G, Is the language L(G) ambiguous. For some G the answer will be "yes", for others it will be "no", but clearly we must have one or the other. The problem is to decide whether the statement is true for any G we are given. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

Similarly, consider the problem of equivalence of context free grammar i.e., to determine whether two context free grammars are equivalent. Again, given context free grammars G_1 and G_2, the answer may be "yes" or "no". The problem is to decide whether the statement is true for any two given context free grammars G_1 and G_2. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

A class of problems with two outputs "yes" or "no" is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs "yes" or "no". Otherwise, the class of problems is said to be undecidable (unsolvable).

2.3 THE HALTING PROBLEM

There are many problem which are not computable. But, we start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the **Halting problem** can be put as:

Given a Turing machine M and an input *w* to the machine M, determine if the machine M will eventually halt when it is given input *w*.

Trial solution: Just run the machine M with the given input *w*.

- If the machine M halts, we know the machine halts.
- But if the machine doesn't halt in a reasonable amount of time, we cannot conclude that it won't halt. May be we didn't wait long enough.

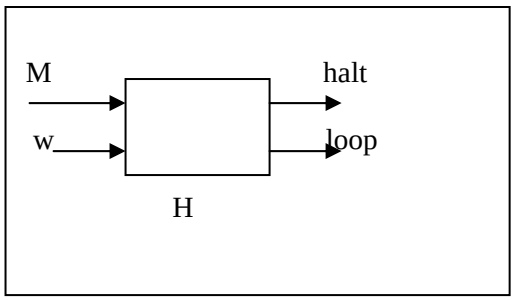
What we need is an algorithm that can determine the correct answer for any M and *w* by performing some analysis on the machine's description and the input. But, we will show that no such algorithm exists.

Let us see first, proof devised by Alan Turing (1936) that halting problem is unsolvable.

Suppose you have a solution to the halting problem in terms of a machine, say, H. H takes two inputs:

1. a program M and
2. an input *w* for the program M.

H generates an output "***halt***" if H determines that M stops on input *w* or it outputs "***loop***" otherwise.

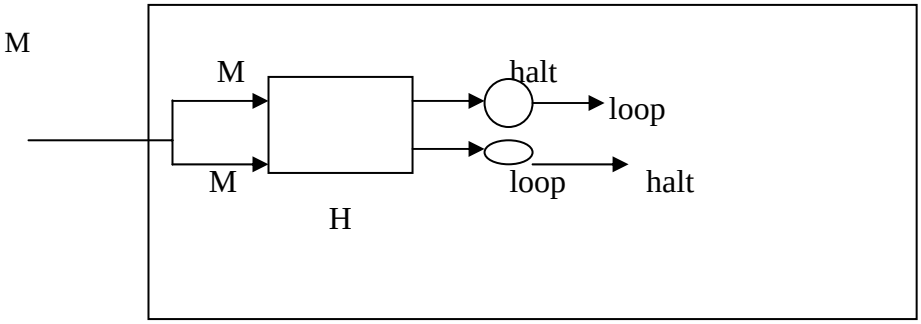


So now H can be revised to take M as both inputs (the program and its input) and H should be able to determine if M will halt on M as its input.

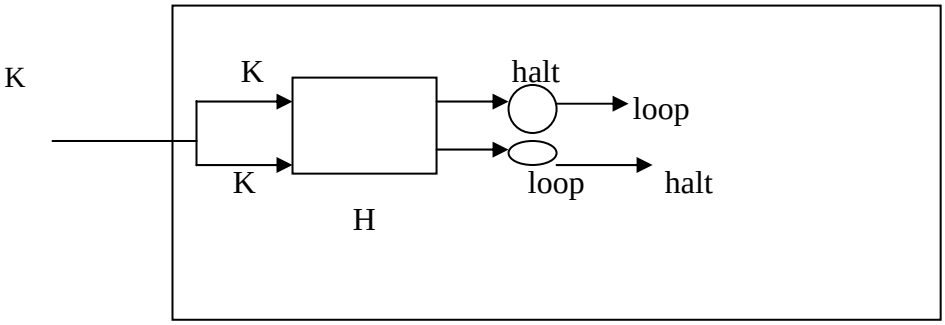
Let us construct a new, simple algorithm K that takes H's output as its input and does the following:

1. if H outputs "***loop***" then K halts,
2. otherwise H's output of "***halt***" causes K to loop forever.

That is, K will do the **opposite** of H's output.



Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it).
If H says that K loops then K will halt.

In either case H gives the wrong answer for K. **Thus H cannot work in all cases.**

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, The halting problem is undecidable.

Now, we formally define what we mean by the halting problem.

Definition 1.1: Let W_M be a string that describes a Turing machine M = (Q, Σ, T, δ, q_0, F), and let *w* be a string in Σ^*. We will assume that W_M and w are encoded as a string of 0's and 1's. A solution of the halting problem is a Turing machine H, which for any W_M and *w*, performs the computation

$q_0\ W_M\ w \mid\!\!-^*\ x_1\ q_y\ x_2$ if M applied to w halts, and

$q_0\ W_M\ w \mid\!\!-^*\ y_1\ q_n\ y_2$ if M applied to w does not halt.

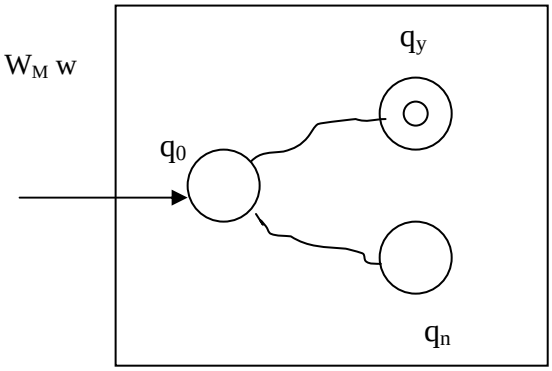
Here q_y and q_n are both final states of H.

Theorem 1.1: There does not exist any Turing machine H that behaves as required by Definition 1.1. The halting problem is therefore undecidable.

Proof: We provide proof by contradiction. Let us assume that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the string W_M *w*. The requirement is then that, the Turing machine H will halt with either a yes or no answer. We capture this by asking that H will halt in one of two corresponding final states, say, q_y or q_n. We want H to operate according to the following rules:

$q_0\ W_M\ w \mid\!\!-^*_M\ x_1\ q_y\ x_2$ if M applied to *w* halts, and
$q_0\ W_M\ w \mid\!\!-^*_M\ y_1\ q_n\ y_2$ if M applied to *w* does not halt.

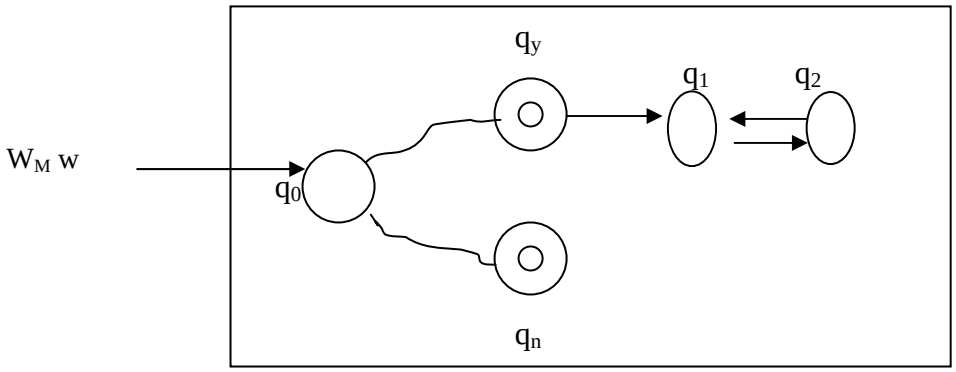
This situation can also be visualized by a block diagram given below:



Next, we modify H to produce H_1 such that

- If H says that it will halt then H_1 itself would loop
- If H says that H will not halt then H_1 will halt.

We can achieve this by adding two more states say, q_1 and q_2. Transitions are defined from q_y to q_1, from q_1 to q_2 and from q_2 to q_1, regardless of the tape symbol, in such a way that the tape remains unchanged. This is shown by another block diagram given below.



Formally, the action of H_1 is described by

$q_0\ W_M\ w \mid\!\!-^*_{H1} \infty$ if M applied to w halts, and

$q_0\ W_M\ w \mid\!\!-^*_{H1} y_1\ q_n\ y_2$ if M applied to w does not halt.

Here, ∞ stands for Turing machine is in infinite loop *i.e*., Turing machine will run forever. Next, we construct another Turing machine H_2 from H_1. This new machine takes as input W_M and copies it, ending in its initial state q_0. After that, it behaves exactly like H_1. The action of H_2 is such that

$q_0\ W_M \mid\!\!-^*_{H2} q_0\ W_M\ W_M \mid\!\!-^*_{H2} \infty$ if M applied to W_M halts, and

$q_0\ W_M \mid\!\!-^*_{H2} y_1\ q_n\ y_2$ if H_2 applied to W_M does not halt.

This clearly contradicts what we assumed. In either case H_2 gives the wrong answer for W_M. Thus H cannot work in all cases.

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, the halting problem is undecidable.

Theorem 2.2: If the halting problem were decidable, then every recursively enumerable language would be recursive. Consequently, the halting problem is undecidable.

Proof: Recall that

1. A language is *recursively enumerable* if there exists a Turing machine that accepts every string in the language and does not accept any string not in the language.

2. A language is *recursive* if there exists a Turing machine that accepts every string in the language and rejects every string not in the language.

Let L be a recursively enumerable language on Σ, and let M be a Turing machine that accepts L. Let us assume H be the Turing machine that solves the halting problem. We construct from this following algorithm:

1. Apply H to W_M w. If H says "no", then by definition w is not in L.

2. If H says "yes", then apply M to w. But M must halt, so it will ultimately tell us whether w is in L or not.

This constitutes a membership algorithm, making L recursive. But, we know that there are recursively enumerable languages that are not recursive. The contradiction implies that H cannot exist *i.e.,* the halting problem is undecidable.

2.4 REDUCTION TO ANOTHER UNDECIDABLE PROBLEM

Once we have shown that the halting problem is undecidable, we can show that a large class of other problems about the input/output behaviour of programs are undecidable.

Examples of undecidable problems

- **About Turing machines**:
  - Is the language accepted by a TM empty, finite, regular, or context-free?
  - Does a TM meet its "specification ?" that is, does it have any "bugs."
- **About Context Free languages**
  - Are two context-free grammars equivalent?
  - Is a context-free grammar ambiguous?

Not so surprising, Although this result is sweeping in scope, may be it is not too surprising. If a simple question such as whether a program halts or not is undecidable, why should one expect that any other property of the input/output behaviour of programs is decidable? Rice's theorem makes it clear that failure to decide halting implies failure to decide any other interesting question about the input/output behaviour of programs. Before we consider Rice's theorem, we need to understand the concept of problem reduction on which its proof is based.

Reducing problem B to problem A means finding a way to convert problem B to problem A, so that a solution to problem A can be used to solve problem B.

One may ask, Why is this important? A reduction of problem B to problem A shows that problem A is at least as difficult to solve as problem B.Also, we can show the following:

- To show that a problem A is undecidable, we reduce another problem that is known to be undecidable to A.
- Having proved that the halting problem is undecidable, we use problem reduction to show that other problems are undecidable.

Example 1: Totality Problem

Decide whether an arbitrary TM halts on all inputs. (If it does, it computes a "total function"). This is equivalent to the problem of whether a program can ever enter an infinite loop, for any input. It differs from the halting problem, which asks whether it enters an infinite loop for a particular input.

Proof: We prove that the halting problem is reducible to the totality problem. That is, if an algorithm can solve the totality problem, it can be used to solve the halting problem. Since no algorithm can solve the halting problem, the totality problem must also be undecidable.
The reduction is as follows. For any TM M and input *w*, we create another TM M_1 that takes an arbitrary input, ignores it, and runs M on *w*. Note that M_1 halts on all inputs if and only if M halts on input *w*. Therefore, an algorithm that tells us whether

M_1 halts on all inputs also tells us whether M halts on input *w*, which would be a solution to the halting problem.

Hence, The totality problem is undecidable.

Example 2: Equivalence problem

Decide whether two TMs accept the same language. This is equivalent to the problem of whether two programs compute the same output for every input.

Proof: We prove that the totality problem is reducible to the equivalence problem. That is, if an algorithm can solve the equivalence problem, it can be used to solve the totality problem. Since no algorithm can solve the totality problem, the equivalence problem must also be unsolvable.

The reduction is as follows. For any TM M, we can construct a TM M_1 that takes any input *w*, runs M on that input, and outputs "yes" if M halts on *w*. We can also construct a TM M_2 that takes any input and simply outputs "yes." If an algorithm can tell us whether M_1 and M_2 are equivalent, it can also tell us whether M_1 halts on all inputs, which would be a solution to the totality problem.

Hence, the equivalence problem is undecidable.

Practical implications

- The fact that the totality problem is undecidable means that we cannot write a program that can find any infinite loop in any program.
- The fact that the equivalence problem is undecidable means that the code optimization phase of a compiler may improve a program, but can never guarantee finding the optimally efficient version of the program. There may be potentially improved versions of the program that it cannot even be sure are equivalent.

  We now describe a more general way of showing that a problem is undecidable i.e., **Rice's theorem**. First we introduce some definitions.

- A *property* of a program (TM) can be viewed as the set of programs that have that property.
- A *functional (or non-trivial) property* of a program (TM) is one that some programs have and some don't.

Rice's theorem (proof is not required)

- "Any functional property of programs is undecidable."
- A functional property is:

  (i) a property of the input/output behaviour of the program, that is, it describes the mathematical function the program computes.

  (ii) nontrivial, in the sense that it is a property of some programs but not all programs.

Examples of functional properties

- The language accepted by a TM contains at least two strings.
- The language accepted by a TM is empty (contains no strings).

- The language accepted by a TM contains two different strings of the same length.

Rice's theorem can be used to show that whether the language accepted by a Turing machine is context-free, regular, or even finite, are undecidable problems.
Not all properties of programs are functional.

2.5 UNDECIDABILITY OF POST CORRESPONDENCE PROBLEM

Undecidable problems arise in language theory also. It is required to develop techniques for proving particular problems undecidable. In 1946, Emil Post proved that the following problem is undecidable:

Let Σ be an alphabet, and let L and M be two lists of nonempty strings over Σ, such that L and M have the same number of strings. We can represent L and M as follows:

$L = (w_1, w_2, w_3, ..., w_k)$
$M = (v_1, v_2, v_3, ..., v_k)$

Does there exist a sequence of one or more integers, which we represent as (i, j, k, ..., m), that meet the following requirements:

- Each of the integers is greater than or equal to one.
- Each of the integers is less than or equal to k. (Recall that each list has k strings).
- The concatenation of $w_i, w_j, w_k, ..., w_m$ is equal to the concatenation of $v_i, v_j, v_k, ..., v_m$.

If there exists the sequence (i, j, k, ..., m) satisfying above conditions then (i, j, k, ..., m) is a solution of PCP.

Let us consider some examples.

Example 3: Consider the following instance of the PCP:

Alphabet Σ = { a, b }
List L = (a, ab)
List M = (aa, b)

We see that (1, 2) is a sequence of integers that solves this PCP instance, since the concatenation of a and ab is equal to the concatenation of aa and b
(i.e ,$w_1 w_2 = v_1 v_2$ = aab). Other solutions include: (1, 2, 1, 2) , (1, 2, 1, 2, 1, 2) and so on.

Example 4: Consider the following instance of the PCP Alphabet Σ = { 0, 1 }

List L = (0, 01000, 01)
List M = (000, 01, 1)

A sequence of integers that solves this problem is (2, 1, 1, 3), since the concatenation of 01000, 0, 0 and 01 is equal to the concatenation of 01, 000, 000 and 1 (i.e., $w_2 w_1 w_1 w_3 = v_2 v_1 v_1 v_3$ =010000001).

2.6 UNDECIDABLE PROBLEMS FOR CONTEXT FREE LANGUAGES

The Post correspondence problem is a convenient tool to study undecidable questions for context free languages. We illustrate this with an example.

Theorem 1.2: There exists no algorithm for deciding whether any given context-free grammar is ambiguous.

Proof : Consider two sequences of strings $A = (u_1, u_2, \dots, u_m)$ and $B = (v_1, v_2, \dots, v_m)$ over some alphabet $\sum$. Choose a new set of distinct symbols a_1, $a_2, \dots, a_m$, such that

$$\{a_1, a_2, \dots, a_m\} \cap \sum = \varnothing,$$

and consider the two languages

$L_A = \{ u_i u_j, \dots u_l u_k a_k a_l \dots, a_j a_i\}$ defined over A and $\{a_1, a_2, \dots, a_m\}$

and

$L_B = \{ v_i v_j, \dots v_l v_k a_k a_l \dots, a_j a_i\}$ defined over B and $\{a_1, a_2, \dots, a_m\}$.

Let G be the context free grammar given by

$$(\{S, S_A, S_B\}, \{a_1, a_2, \dots, a_m\} \cup \sum, P, S)$$

where the set of productions P is the union of the two subsets: the first set P_A consists of

$$S \rightarrow S_A,$$
$$S_A \rightarrow u_i S_A a_i \mid u_i a_i, \qquad i = 1, 2, \dots, n,$$

the second set P_B consists of

$$S \rightarrow S_B,$$
$$S_B \rightarrow v_i S_B a_i \mid v_i a_i, \qquad i = 1, 2, \dots, n.$$

Now take

$$G_A = (\{S, S_A\}, \{a_1, a_2, \dots, a_m\} \cup \sum, P_A, S)$$

and

$$G_B = (\{S, S_B\}, \{a_1, a_2, \dots, a_m\} \cup \sum, P_B, S)$$

Then,

$L_A = L(G_A)$,
$L_B = L(G_B)$,

and

$L(G) = L_A \cup L_B$.

It is easy to see that G_A and G_B by themselves are unambiguous. If a given string in L (G) ends with a_i, then its derivation with grammar G_A must have started with $S \Rightarrow u_i S_A a_i$. Similarly, we can tell at any later stage which rule has to be applied. Thus, If G is ambiguous it must be because there is w for which there are two derivations

$S \Rightarrow S_A \Rightarrow u_i S a_i \Rightarrow^* u_i u_j \ldots u_k a_k \ldots a_j a_i = w$

and

$S \Rightarrow S_B \Rightarrow v_i S a_i \Rightarrow^* v_i v_j \ldots v_k a_k \ldots a_j a_i = w.$

Consequently, if G is ambiguous, then the Post correspondence problem with the pair (A, B) has a solution. Conversely, If G is unambiguous, then the Post correspondence problem cannot have solution.

If there existed an algorithm for solving the ambiguity problem, we could adapt it to solve the Post correspondence problem. But, since there is no algorithm for the Post correspondence problem, we conclude that the ambiguity problem is undecidable.

2.7 OTHER UNDECIDABLE PROBLEMS

- Does a given Turing machine M halt on all inputs?
- Does Turing machine M halt for *any* input? (That is, is L(M)=∅?)
- Do two Turing machines M_1 and M_2 accept the same language?
- Is the language L(M) finite?
- Does L(M) contain any two strings of the same length?
- Does L(M) contain a string of length k, for some given k?
- If G is a unrestricted grammar.
- Does L(G) = ∅ ?
- Does L(G) infinite ?
- If G is a context sensitive grammar.
- Does L(G) = ∅ ?
- Does L(G) infinite ?
- If L_1 and L_2 are any context free languages over Σ.
- Does $L_1 \cap L_2 = \emptyset$?
- Does $L_1 = L_2$?
- Does $L_1 \subseteq L_2$?
- If L is recursively enumerable language over Σ.
- Does L empty ?
- Does L finite ?

Ex. 1) Show that the state-entry problem is undecidable.

Hint: The problem is described as follows: Given any Turing machine M = (Q, Σ, T, δ, q_0, F) and any $q \in Q$, $w \in \Sigma^+$, to determine whether Turing machine M, when given input w, ever enters state q.

Ex. 2) Show that the blank tape halting problem is undecidable.

Hint: The problem is described as follows: Given a Turing machine M, Does Turing machine M halts when given a blank input tape?

Ex. 3) Consider the following instance of the PCP:

Alphabet $\Sigma = \{ 0, 1, 2 \}$
List L = (0, 1, 2)
List M = (00, 11, 22)
Does PCP have a solution ?

Ex. 4) Consider the following instance of the PCP:

Alphabet $\Sigma = \{ a, b \}$
List L = (ba, abb, bab)
List M = (bab, bb, abb)
Does PCP have a solution ?

Ex. 5) Does PCP with two lists A = (b, babbb, ba) and B = (bbb, ba, a) have a solution ?

Ex. 6) Does PCP with two lists A = (ab, b, b) and (abb, ba, bb) have a solution ?

Ex.7) Show that there does not exist algorithm for deciding whether or not

$L(G_A) \cap L(G_B) = \varnothing$ for arbitrary context free grammars G_A and G_B.

2.8 SUMMARY

- A decision problem is a problem that requires a yes or no answer. A decision problem that admits no algorithmic solution is said to be undecidable.
- No undecidable problem can ever be solved by a computer or computer program of any kind. In particular, there is no Turing machine to solve an undecidable problem.
- We have not said that undecidable means we don't know of a solution today but might find one tomorrow. It means we can never find an algorithm for the problem.
- We can show no solution can exist for a problem A if we can reduce it into another problem B and problem B is undecidable.

2.9 SOLUTIONS/ANSWERS

Ex. 1)

The problem is described as follows: Given any Turing machine $M = (Q, \Sigma, T, \delta, q_0, F)$ and any $q \in Q$, $w \in \Sigma^+$, to determine whether Turing machine M, when given input w, ever enters state q.

The problem is to determine whether Turing machine M, when given input w, ever enters state q.

The only way a Turing machine M halts is if it enters a state q for which some transition function $\delta(q_i, a_i)$ is undefined. Add a new final state Z to the Turing machine, and add all these missing transitions to lead to state Z. Now use the (assumed) state-entry procedure to test whether state Z is ever entered when M is given input w. This will reveal whether the original machine M halts. We conclude that it must not be possible to build the assumed state-entry procedure.

Ex. 2)

It is another problem which is undecidable. The problem is described as follows: Given a Turing machine M, does Turing machine M halts when given a blank input tape?

Here, we will reduce the blank tape halting problem to the halting problem. Given M and w, we first construct from M a new machine M_w that starts with a blank tape, writes w on it, then positions itself in configuration q_0w. After that, M_w acts exactly like M. Hence, M_w will halt on a blank tape if and only if M halts on w.

Suppose that the blank tape halting problem were decidable. Given any M and w, we first construct M_w, then apply the blank tape halting problem algorithm to it. The conclusion tells us whether M applied to w will halt. Since this can be done for any M and w, an algorithm for the blank tape halting problem can be converted into an algorithm for the halting problem. Since the halting problem is undecidable, the same must be true for the blank tape halting problem.

Ex. 3)

There is no solution to this problem, since for any potential solution, the concatenation of the strings from list L will contain half as many letters as the concatenation of the corresponding strings from list M.

Ex. 4)

We can not have string beginning with w_2 = abb as the counterpart v_2 = bb exists in another sequence and first character does not match. Similarly, no string can begin with w_3 = bab as the counterpart v_3 = abb exists in another sequence and first character does not match. The next choice left with us is start the string with w_1 = ba from L and the counterpart v_1 = bab from M. So, we have

ba

bab

The next choice from L must begin with b. Thus, either we choose w_1 or w_3 as their string starts with symbol b. But, the choice of w_1 will make two string look like:

baba

babbab

While the choice of w_3 direct to make choice of v_3 and the string will look like:

babab

bababb

Since the string from list M again exceeds the string from list L by the single symbol 1, a similar argument shows that we should pick up w_3 from list L and v_3 from list M. Thus, there is only one sequence of choices that generates compatible strings, and for this sequence string M is always one character longer. Thus, this instance of PCP has no solution.

Ex. 5)

We see that (2, 1, 1, 3) is a sequence of integers that solves this PCP instance, since the concatenation of babbb, b, b and ba is equal to the concatenation of ba, bbb, bbb and a (i.e., $w_2\ w_1\ w_1\ w_3 = v_2\ v_1\ v_1\ v_3$ = babbbbbba).

Ex. 6)

For each string in A and corresponding string in B, the length of string of A is less than counterpart string of B for the same sequence number. Hence, the string generated by a sequence of strings from A is shorter than the string generated by the sequence of corresponding strings of B. Therefore, the PCP has no solution.

Ex. 7)

Proof : Consider two grammars

$G_A = (\{ S_A \}, \{a_1, a_2, \dots , a_m\} \cup \sum, P_A, S_A)$

and

$G_B = (\{S_B \}, \{a_1, a_2, \dots , a_m\} \cup \sum, P_B, S_B)$.

where the set of productions P_A consists of
$S_A \rightarrow u_i S_A a_i \mid u_i a_i, \quad i = 1, 2,\dots, n,$

and the set of productions P_B consists of

$S_B \rightarrow v_i S_B a_i \mid v_i a_i, \quad i = 1, 2,\dots, n.$

where consider two sequences of strings $A = (u_1, u_2, \dots , u_m)$ and $B = (v_1, v_2, \dots , v_m)$ over some alphabet $\sum$. Choose a new set of distinct symbols $a_1, a_2, \dots , a_m$, such that

$\{a_1, a_2, \dots , a_m\} \cap \sum = \varnothing,$

Suppose that $L(G_A)$ and $L(G_B)$ have a common element, i.e.,

$S_A \Rightarrow u_i S a_i \Rightarrow^* u_i\ u_j \dots u_k a_k \dots a_j a_i$

and

$S_B \Rightarrow v_i S a_i \Rightarrow^* v_i\ v_j \dots v_k a_k \dots a_j a_i.$

Then the pair (A, B) has a PC-solution. Conversely, if the pair does not have a PC- solution, then $L(G_A)$ and $L(G_B)$ cannot have a common element. We conclude that $L(G_A) \cap L(G_B)$ is nonempty if and only if (A, B) has a PC- solution.

2.10 FURTHER READINGS

1. *Elements of the Theory of Computation*, H.R. Lewis & C.H.Papadimitriou: PHI, (1981).

2. *Introduction to Automata Theory, Languages, and Computation* (II Ed.), J.E. Hopcroft, R.Motwani & J.D.Ullman: Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language,* J.E. Hopcroft and J.D. Ullman: *and Computation,* Narosa Publishing House (1987).

4. *Introduction to Languages and Theory of Computation*, J.C. Martin: Tata-Mc Graw-Hill (1997).

5. *Computers and Intractability– A Guide to the Theory of NP-Completeness*, M.R. Garey & D.S. Johnson, W.H. Freeman and Company (1979).

UNIT 3 COMPLEXITY OF ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

In unit 2 of the block, we discussed a number of problems which cannot be solved by algorithmic means and also discussed a number of issues about such problems.

In this unit, we will discuss the issue of efficiency of computation of an algorithm in terms of the *amount of time* used in its execution. On the basis of analysis of an algorithm, the amount of time that is estimated to be required in executing an algorithm, will be referred to as the time complexity of the algorithm. The time complexity of an algorithm is measured in terms of some (basic) *time unit* (not second or nano-second). Generally, time taken in executing one move of a TM, is taken as (basic) time unit for the purpose. Or, alternatively, time taken in executing some elementary operation like addition, is taken as one unit. More complex operations like multiplication etc, are assumed to require an *integral* number of basic units. As mentioned earlier, given many algorithms (solutions) for solving a problem, we would like to choose the most efficient algorithm from amongst the available ones. For *comparing efficiencies of algorithms*, that solve a particular problem, *time complexities of algorithms are* considered as *functions of the sizes of the problems* (to be discussed). The *time complexity functions of the algorithms are compared in terms of their* growth rates (*to be defined*) as growth rates are considered important measures of *comparative efficiencies*.

The concept of the **size of a problem,** though a fundamental one, yet is difficult to define precisely. Generally, the size of a *problem*, is measured in terms of the size of the *input*. The concept of the size of an input of a problem may be explained informally through examples. In the case of multiplication of two nxn (squares) matrices, the size of the problem may be taken as n^2, i.e, the number of elements in each matrix to be multiplied. For problems involving polynomials, the degrees of the polynomials may be taken as measure of the sizes of the problems.

For a problem, a solution with time complexity which can be expressed as a polynomial of the size of the problem, is considered to have an **efficient solution**. However, not many problems that arise in practice, admit any efficient algorithms, as these problems can be solved, if at all, by only non-polynomial time algorithms. A problem which does not have any (*known*) polynomial time algorithm is called an **intractable** problem.

We may note that the term *solution* in its general form *need not* be an *algorithm*. If by tossing a coin, we get the correct answer to each instance of a problem, then the process of tossing the coin and getting answers constitutes a solution. But, the process is not an algorithm. Similarly, we solve problems based on **heuristics**, i.e., good guesses which, generally but not necessarily always, lead to solutions. All such cases of solutions are not algorithms, or algorithmic solutions. To be more explicit, by an algorithmic solution A of a problem L (*considered as a language*) from a problem domain Σ^*, we mean that among other conditions, the following are satisfied. A is a step-by-step method in which for each instance of the problem, there is a definite sequence of execution steps (*not involving any guess work*). A *terminates* for each $x \in \Sigma^*$, irrespective of whether $x \in L$ or $x \notin L$.

In this sense of algorithmic solution, only a ***solution by a Deterministic TM*** *is called an* ***algorithm.*** *A solution by a* ***Non-Deterministic TM may not be an algorithm.***

(i) However, for every NTM solution, there is a Deterministic TM (DTM) solution of a problem. Therefore, if there is an NTM solution of a problem, then there is an algorithmic solution of the problem. *However, the symmetry may end here*.

The *computational equivalence* of Deterministic and Non-Deterministic TMs does not state or guarantee any *equivalence in respect of requirement of resources* like time and space by the Deterministic and Non-Deterministic models of TM, for solving a (solvable) problem. To be more precise, if a problem is solvable in polynomial-time by a Non-Deterministic Turing Machine, then it is, of course, *guaranteed* that there is a deterministic TM that solves the problem, but it is *not guaranteed* that there exists a Deterministic TM that solves the problem *in polynomial time.* Rather, this fact forms the basis for one of the deepest open questions of Mathematics, which is stated as ***'whether*** P = NP?'(P and NP to be defined soon).

The question put in simpler language means: Is it possible to design a Deterministic TM to solve a problem in polynomial time, for which, a Non-Deterministic TM that solves the problem in polynomial time, has already been designed?

We summarize the above discussion from the intractable problem's definition onward. Let us begin with definitions of the notions of P and NP.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

Thus starting with two distinct classes of problems, viz.**, tractable** problems and **intractable** problems, we introduced two classes of problems called **P and NP**. Some interesting relations known about these classes are:

(i) P = set of tractable problems
(ii) $P \subseteq NP$.

(*The relation (ii) above* simply follows from the fact that every Deterministic TM is a special case of a Non-Deterministic TM).

However, it is not known whether P=NP or $P \subset NP$. This forms the basis for the subject matter of the rest of the chapter. As a first step, we introduce some notations to facilitate the discussion of the concept of computational complexity.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the concepts of time complexity, size of a problem, growth rate of a function;
- define and explain the well-known notations for growth rates of functions, viz., O, Ω, Θ, o, ω;
- explain criteria for classification of problems into undefinable defineable but not solvable, solvable but not feasible, P, NP, NP-hard and NP-Complete etc.;
- define a number of problems which are known to be NP-complete problems;
- explain polynomial-reduction as a technique of establishing problems as NP-hard; and
- establish NP-completeness of a number of problems.

3.2 NOTATIONS FOR GROWTH RATES OF FUNCTIONS

The time required by a solution or an algorithm for solving a (solvable) problem, depends *not only* on the size of the problem/input and the number of operations that the algorithm/solution uses, *but also* on the hardware and software used to execute the solution. However, the effect of change/improvement in hardware and software on the time required may be closely approximated by a *constant*.

Suppose, a supercomputer executes *instructions* one million times faster than another computer. Then irrespective of the size of a (solvable) problem and the solution used to solve it, the supercomputer solves the *problem* roughly million times faster than the computer, if the same solution is used on both the machines to solve the problem. Thus we conclude that the time requirement for execution of a solution, changes roughly by a *constant factor* on change in hardware, software and environmental factors.

3.2.1 The Constant Factor in Complexity Measure

An **important consequence of the above discussion** is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem. *Thus, functions differing from each other by constant factors, when treated as time complexities should not be treated as different, i.e., should be treated as complexity-wise equivalent.*

3.2.2 Asymptotic Considerations

Computers are generally used to solve problems involving *complex* solutions. The complexity of solutions may be either because of the large number of involved computational steps and/or large size of input data. The plausibility of the claim apparently follows from the fact that, when required, computers are used *generally not to* find the product of two 2×2 matrices *but to find* the product of two $n \times n$ matrices for large n running into hundreds or even thousands.

Similarly, computers, when required, are generally used *not to find roots* of quadratic equations but for finding roots of complex equations including polynomial equations of *degrees more than hundreds or sometimes even thousands.*

The above discussion leads to the conclusion that when considering time complexities $f_1(n)$ and $f_2(n)$ of (computer) solutions of a problem of size n, we need to consider and compare the behaviours of the two functions only for large values of n. If the relative behaviours of two functions for smaller values conflict with the relative behaviours for larger values, then we may ignore the conflicting behaviour for smaller values. For example, if the earlier considered two functions

$$f_1(n) = 1000\, n^2 \qquad \text{and}$$
$$f_2(n) = 5n^4$$

represent time complexities of two solutions of a problem of size n, then despite the fact that

$$f_1(n) \geq f_2(n) \qquad \text{for } n \leq 14,$$

we would still prefer the solution having $f_1(n)$ as time complexity because

$$f_1(n) \leq f_2(n) \qquad \text{for all } n \geq 15.$$

This explains the reason for the presence of the phrase 'n ≥ k' in the definitions of the various measures of complexities discussed below:

3.2.3 Well Known Asymptotic Growth Rate Notations

In the following we discuss some well-known growth rate notations. These notations denote relations from functions to functions.

For example, if functions

f, g: $N \rightarrow N$ are given by

$f(n) = n^2 - 5n$ and

$g(n) = n^2$

then

$O(f(n)) = g(n)$ or $O(n^2 - 5n) = n^2$

(the notation O to be defined soon).

To be more precise, each of these notations is a mapping that associates a *set of* functions to each function. For example, if f (n) is a polynomial of degree k then the set O (f (n)) includes all polynomials of degree less than or equal to k.

The five well-known notations and how these are pronounced:

(i) O (O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) Ω (Ω (n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) Θ (Θ (n^2) is pronounced as 'theta of n^2')

(iv) o (o (n^2) is pronounced as 'little-oh of n^2')

(v) ω (ω (n^2) is pronounced as 'little- omega of n^2')

Remark 3.2.3.1

In the discussion of any one of the five notations, generally two functions say f and g are involved. The functions have their domains and Codomains as N, the set of natural numbers, i.e.,

$f: N \rightarrow N$
$g: N \rightarrow N$

These functions may also be considered as having domain and codomain as R.

Remark 3.2.3.2

The purpose of these asymptotic growth rate notations and functions denoted by these notations, is to facilitate the recognition of *essential character of a complexity function* through some simpler functions delivered by these notations. For example, a complexity function $f(n) = 5004\ n^3 + 83\ n^2 + 19\ n + 408$, has essentially the same behaviour as that of $g(n) = n^3$ as the problem size n becomes larger and larger. But $g(n) = n^3$ is much more comprehensible than the function f(n). Let us discuss the notations, starting with the notation **O.**

3.2.4 The Notation O

Provides asymptotic *upper bound* for a given function. Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \leq C\ g(x)$ for all $x \geq k$ (A)

(*The restriction of being positive on integers/reals is justified as all complexities are positive numbers*).

Example 3.2.4.1: For the function defined by

$f(x) = 2x^3 + 3x^2 + 1$
show that

(i) $f(x) = O(x^3)$
(ii) $f(x) = O(x^4)$
(iii) $x^3 = O(f(x))$
(iv) $x^4 \neq O(f(x))$
(v) $f(x) \neq O(x^2)$

Solutions

Part (i)

Consider

$f(x) = 2x^3 + 3x^2 + 1$
$\leq 2x^3 + 3x^3 + 1 \quad x^3 = 6x^3$ for all $x \geq 1$

(by replacing each term x^i by the highest degree term x^3)

$\therefore$ there exist $C = 6$ and $k = 1$ such that
$f(x) \leq C.\ x^3$ for all $x \geq k$

Thus, we have found the required constants C and k. Hence f(x) is $O(x^3)$.

Part (ii)

As above, we can show that

$$f(x) \leq 6 \quad x^4 \quad \text{for all } x \geq 1.$$

However, we may also, by computing some values of f(x) and x^4, find C and k as follows:

$$f(1) = 2+3+1 = 6 \quad ; \quad (1)^4 = 1$$
$$f(2) = 2.2^3 + 3.2^2 + 1 = 29 \quad ; \quad (2)^4 = 16$$
$$f(3) = 2.3^3 + 3.3^2 + 1 = 82 \quad ; \quad (3)^4 = 81$$

for C = 2 and k = 3 we have
$f(x) \leq 2.\ x^4$ for all $x \geq k$

Hence, f(x) is $O(x^4)$.

Part (iii)

for C = 1 and k = 1 we get
$x^3 \leq C\ (2x^3 + 3x^2 + 1)$ for all $x \geq k$

Part (iv)

We prove the result by contradiction. Let there exist positive constants C and k such that

$$x^4 \leq C\ (2x^3 + 3x^2 + 1) \text{ for all } x \geq k$$
$$\therefore x^4 \leq C\ (2x^3 + 3x^3 + x^3) = 6Cx^3 \text{ for } x \geq k$$
$$\therefore\ x^4 \leq 6\ C\ x^3 \quad \text{for all } x \geq k.$$

implying $x \leq 6C$ for all $x \geq k$

But for x = max of { 6 C + 1, k}, the previous statement is not true.
Hence the proof.

Part (v)

Again we establish the result by contradiction.
Let $O\ (2\ x^3+3x^2+1) = x^2$
Then for some positive numbers C and k
$2x^3 + 3x^2+1 \leq C\ x^2$ for all $x \geq k$,
implying
$x^3 \leq C\ x^2$ for all $x \geq k$ *(Θ $x^3 \leq 2x^3+3x^2+1$ for all $x \geq 1$)*
implying
$x \leq C$ for $x \geq k$
Again for x = max $\{C+1, k\}$
The last imaquality does not hold. Hence the result.

Example: The big-oh notation can be used to estimate S_n, the sum of first n positive integers

Hint: S_n=1+2+3+…….+n ≤ n+n +……+ n = n^2
Therefore, $S_n = O\ (n^2)$.

Remark 3.2.4.2

It can be easily seen that for given functions f(x) and g(x), if there exists one pair of C and k with $f(x) \le C.g(x)$ for all $x \ge k$, then there exist infinitely many pairs (C_i, k_i) which satisfy

$f(x) \le C_i\, g(x)$ for all $x \ge k_i$.

Because for **any** $\mathbf{C_i} \ge C$ and **any** $\mathbf{k_i} \ge k$, the above inequality is true, if $f(x) \le c.g(x)$ for all $x \ge k$.

3.2.5 The Notation Ω

Provides an asymptotic *lower bound* for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \ge C\,(g(x))$ whenever $x \ge k$

Example 3.2.5.1: For the functions

$f(x) = 2x^3 + 3x^2 + 1$ and $h(x) = 2x^3 - 3x^2 + 2$

show that

(i) $f(x) = \Omega(x^3)$
(ii) $h(x) = \Omega(x^3)$
(iii) $h(x) = \Omega(x^2)$
(iv) $x^3 = \Omega(h(x))$
(v) $x^2 \ne \Omega(h(x))$

Solutions:

Part (i)

For C =1, we have
$f(x) \ge C\, x^3$ for all $x \ge 1$

Part (ii)

$h(x) = 2x^3 - 3x^2 + 2$
Let C and $k > 0$ be such that
$2x^3 - 3x^2 + 2 \ge C\, x^3$ for all $x \ge k$
i.e., $(2 - C)\, x^3 - 3x^2 + 2 \ge 0$ for all $x \ge k$

Then C = 1 and $k \ge 3$ satisfy the last inequality.

Part (iii)

$2x^3 - 3x^2 + 2 = \Omega(x^2)$
Let the above equation be true.
Then there exists positive numbers C and k
s.t.

$2x^3 - 3x^2 + 2 \ge C\, x^2$ for all $x \ge k$
$2\, x^3 - (3 + C)\, x^2 + 2 \ge 0$

It can be easily seen that lesser the value of C, better the chances of the above inequality being true. So, to begin with, let us take C = 1 and try to find a value of k s.t

$$2x^3 - 4x^2 + 2 \geq 0.$$

For $x \geq 2$, the above inequality holds

$\therefore$ k=2 is such that

$$2x^3 - 4x^2 + 2 \geq 0 \text{ for all } x \geq k$$

Part (iv)

Let the equality

$$x^3 = \Omega (2x^3 - 3x^2 + 2)$$

be true. Therefore, let C>0 and k > 0 be such that

$$x^3 \geq C(2(x^3 - 3/2\, x^2 + 1))$$

For C = ½ and k = 1, the above inequality is true.

Part (v)

We prove the result by contradiction.

Let $x^2 = \Omega (3x^3 - 2x^2 + 2)$

Then, there exist positive constants C and k such that

$x^2 \geq C (3x^3 - 2x^2 + 2)$ for all $x \geq k$

i.e., $(2C + 1)\, x^2 \geq 3C\, x^3 + 2 \geq C\, x^3$ for all $x \geq k$

$\frac{2C+1}{C} \geq x$ for all $x \geq k$

But for any $x \geq 2\ \frac{(2C+1)}{C}$,

The above inequality can not hold. Hence contradiction.

3.2.6 The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

(*Note the last inequalities represent two conditions to be satisfied simultaneously viz., $C_2\, g(x) \leq f(x)$ and $f(x) \leq C_1\, g(x)$).*

We state the following theorem without proof, which relates the three functions O, Ω, Θ

Theorem: For any two functions f(x) and g(x), f(x) = Θ (g(x)) if and only if f(x) = O (g(x)) and f(x) = Ω (g(x)).

Examples 3.2.6.1: For the function
$f(x) = 2x^3 + 3x^2 + 1$, show that

(i) $f(x) = \Theta(x^3)$

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

(iii) $f(x) \neq \Theta(x^4)$

Solutions

Part (i)

for $C_1 = 3$, $C_2 = 1$ and $k = 4$

1. $C_2 x^3 \leq f(x) \leq C_1 x^3$ for all $x \geq k$

Part (ii)

We can show by contradiction that no C_1 exists.

Let, if possible for some positive integers k and C_1, we have $2x^3+3x^2+1 \leq C_1 . x^2$ for all $x \geq k$

Then
$x^3 \leq C_1 x^2$ for all $x \geq k$
i.e.,
$x \leq C_1$ for all $x \geq k$
But for
$x = \max \{C_1 + 1, k\}$
The last inequality is not true

Part (iii)

$f(x) \neq \Theta(x^4)$

We can show by contradiction that there does not exist C_2
s.t

$$C_2 x^4 \leq (2x^3 + 3x^2 + 1)$$

If such a C_2 exists for some k then $C_2 x^4 \leq 2x^3 + 3x^2 + 1 \leq 6x^3$ for all $x \geq k \geq 1$,

implying

$$C_2 x \leq 6 \text{ for all } x \geq k$$

But for $x = \left(\frac{6}{C_2} + 1\right)$
the above inequality is false. Hence, proof of the claim by contradiction.

3.2.7 The Notation o

The asymptotic upper bound provided by big-oh notation may or may not be tight in the sense that if $f(x) = 2x^3 + 3x^2 + 1$

Then for $f(x) = O(x^3)$, though there exist C and k such that
$f(x) \leq C(x^3)$ for all $x \geq k$
yet there may also be some values for which the following equality also holds

$f(x) = C(x^3)$ for $x \geq k$

However, if we consider

$f(x) = O(x^4)$

then there can not exist positive integer C s.t

$f(x) = C\,x^4$ for all $x \geq k$

The case of $f(x) = O(x^4)$, provides an example for the next notation of small-oh.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let C > 0 **be any number**, then f(x) = o(g(x)) (pronounced as little oh of g of x) if there exists natural number k satisfying

$f(x) < C\,g(x)$ for all $x \geq k \geq 1$ (B)

Here we may note the following points:

(i) In the case of little-oh the constant C does not depend on the two functions f (x) and g (x). Rather, we can *arbitrarily* choose C >0.

(ii) The inequality (B) is strict whereas the inequality (A) of big-oh is not necessarily strict.

Example 3.2.7.1: For $f(x) = 2x^3 + 3x^2 + 1$, we have

(i) $f(x) = o(x^n)$ for any $n \geq 4$.
(ii) $f(x) \neq o(x^n)$ for $n \leq 3$

Solution

Let C > 0 **be given and** to find out k satisfying the requirement of little-oh. Consider

$2x^3 + 3x^2 + 1 < C\,x^n$

$= 2 + \frac{3}{x} + \frac{1}{x^3} < C\,x^{n-3}$

Case when n = 4
Then above inequality becomes

$2 + \frac{3}{x} + \frac{1}{x^3} < C\,x$

if we take $k = \max\left\{\frac{7}{C}, 1\right\}$

then
$2x^3 + 3x^2 + 1 < C\,x^4$ for $x \geq k$.

In general, as $x^n > x^4$ for $n \geq 4$,

therefore
$2x^3 + 3x^2 + 1 < C\,x^n$ for $n \geq 4$
for all $x \geq k$
with $k = \max\left\{\frac{7}{c}, 1\right\}$

Part (ii)

We prove the result by contradiction. Let, if possible, $f(x) = 0(x^n)$ for $n \leq 3$.

Then there exist positive constants C and k such that $2x^3+3x^2+1 < C\, x^n$ for all $x \geq k$.

Dividing by x^3 throughout, we get

$$2+ \frac{3}{x}+\frac{1}{x^2} < C\, x^{n-3}$$

$n \leq 3$ and $x \geq k$

As C is arbitrary, we take
C = 1, then the above inequality reduces to

$$2+ \frac{3}{x}+\frac{1}{x^2} < C.\, x^{n-3} \quad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

Also, it can be easily seen that

$$x^{n-3} \leq 1 \qquad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

$$\therefore\ 2+ \frac{3}{x}+\frac{1}{x^2} \leq 1 \qquad \text{for } n \leq 3$$

However, the last inequality is not true. Therefore, the proof by contradiction.

Generalizing the above example, we get the

Example 3.2.7.2: If f(x) is a polynomial of degree m and g(x) is a polynomial of degree n. Then
f(x) = o(g(x)) if and only if n>m.

we state (without proof) below two results which can be useful in finding small-oh upper bound for a given function

More generally, we have

Theorem 3.2.7.3: Let f(x) and g(x) be functions in definition of small-oh notation.

Then f(x) = o(g(x) if and only if

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

Next, we introduce the last asymptotic notation, namely, small-omega. The relation of small-omega to big-omega is similar to what is the relation of small-oh to big-oh.

3.2.8 The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then

$f(x) = \omega(g(x))$
if there exist a positive integer k s.t
$f(x) > C \quad g(x) \qquad$ for all $x \geq k$

Example 3.2.8.1: If $f(x) = 2x^3 + 3x^2 + 1$
then
$f(x) = \omega(x)$
and also
$f(x) = \omega(x^2)$

Solution:

Let C be any positive constant.

Consider

$$2x^3 + 3x^2 + 1 > C\,x$$

To find out $k \geq 1$ satisfying the conditions of the bound ω.

$$2x^2 + 3x + \frac{1}{x} > C \quad \textit{(dividing throughout by x)}$$

Let k be integer with $k \geq C+1$

Then for all $x \geq k$

$$2x^2 + 3x + \frac{1}{x} \geq 2x^2 + 3x > 2k^2 + 3k > 2C^2 + 3C > C. \quad (\Theta\ k \geq C+1)$$

$\therefore f(x) = \omega(x)$

Again, consider, for any $C > 0$,

$$2x^3 + 3x^2 + 1 > C\,x^2$$

then

$$2x + 3 + \frac{1}{x^2} > C \qquad \text{Let k be integer with } k \geq C+1$$

Then for $x \geq k$ we have

$$2x + 3 + \frac{1}{x^2} \geq 2x + 3 > 2k + 3 > 2C + 3 > C$$

Hence

$$f(x) = \omega(x^2)$$

In general, we have the following two theorems (stated without proof).

Theorem 3.2.8.2: If f(x) is a polynomial of degree n, and g(x) is a polynomial of degree n, then

$f(x) = \omega(g(x))$ if and only if $m > n$.

More generally

Theorem 3.2.8.3: Let f(x) and g(x) be functions in the definitions of little-omega Then $f(x) = \omega\, g(x)$ if and only if

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

or

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

Ex.1) Show that $n! = O(n^n)$.

Ex.2) Show that $n^2 + 3\log n = O(n^2)$.

Ex.3) Show that $2^n = O(5^n)$.

3.3 CLASSIFICATION OF PROBLEMS

The fact of being engaged in solving problems may be the only sure indication of a living entity being alive *(though, the distinction between entities being alive and not being alive is getting fuzzier day by day*). The problems, attempted to be solved, may be due to the need for survival in a hostile and competitive environment or may be because of intellectual curiosity of knowing more and more of the nature. In the previous unit, we studied a number of problems which are not solvable by computational means. **We can go still further and categorize the problems, which we encounter or may encounter, into the following broad classes:**

(I) **Problems which can not even be defined formally.**

By a formal definition of a problem, we mean expressing in terms of mathematical entities like sets, relations and functions etc., the information concerning the problem, in respect of at least

a) Possible inputs
b) Possible outcomes
c) Entitles occurring and operations on these entities in the (dynamic) problem domains.

In this sense of *definition of a problem*, what to talk of solving, most of the problems can not even be defined. Think of the following problems.

a) Why the economy is not doing well?
b) Why there is hunger, illiteracy and suffering despite international efforts to eradicate these?
c) Why some people indulge in corrupt practices despite being economically well?

These are some of problems, the definition of each of which require enumeration of potentially infinite parameters, and hence are almost impossible to define.

(II) Problems which can be formally defined but can not be solved by computational means. We discussed some of these problems in the previous unit.

(III) Problems which, though theoretically can be solved by computational means, yet are infeasible, i.e., these problems require so large amount of computational resources that practically is not feasible to solve these problems by computational means. These problems are called **intractable or infeasible** problems. The distinguishing feature of the problems is that for each of these problems any solution has time complexity which is exponential, or at least non-polynomial, function of the problem size.

(IV) Problems that are called feasible or theoretically not difficult to solve by computational means. The distinguishing feature of the problems is that for each instance of any of these problems, there exists a **Deterministic Turing Machine** that solves the problem having time-complexity as a polynomial function of the size of the problem. The class of problem is denoted by P.

(V) Last, but probably most interesting class include large number of problems, for each of which, it is not known whether it is in P or not in P.
These problems fall somewhere between class III and class IV given above However, for each of the problems in the class, it is known that it is in NP, i.e., each can be solved by at least one Non-Deterministic Turing Machine, the time complexity of which is a polynomial function of the size of the problem.

A problem from the class NP can equivalently but in more intuitive way, be defined as one for which a potential solution, if given, can be **verified** in polynomial time whether the potential solution is actually a solution or not.

The problems in this class, are called **NP-Complete problems** (to be formally defined later). *More explicitly, a problem is NP-complete if it is in NP and for which no polynomial-time Deterministic TM solution is known so far.*

Most interesting aspect of NP-complete problems, is that for each of these problems *neither, so far*, it has been possible to design a Deterministic polynomial-time TM solving the problem *nor* it has been possible to show that Deterministic polynomial - time TM solution *can not* exist.

The idea of NP-completeness was introduced by Stephen Cook* in 1971 and the **satisfiability problem** defined below is the first problem that was proved to be NP-complete, of course, by S. Cook.

Next, we enumerate some of the NP-complete problems without justifying why these problems have been placed in the class. Justification for some of these problems will be provided in later sections.

A good source for the study of NP-complete problems and of related topics is Garey & Johnson+

Problem 1: Satisfiability problem (or, for short, SAT) states: Given a Boolean expression, is it satisfiable?

Explanation: A Boolean expression involves

(i) Boolean variables $x_1, x_2,..., x_i, \ldots$, each of which can assume a value either TRUE (generally denoted by 1) or FALSE (generally denoted by 0) and

(ii) Boolean/logical operations: NOT(x_1) (generally denoted by x_i or $\neg x_i$), AND (denoted generally by $\wedge$), and OR (denoted by $\vee$). Other logical operators like $\rightarrow$ and $\leftrightarrow$ can be equivalently replaced by some combinations of $\vee$, $\wedge$ and $\neg$.

(iii) Pair of parentheses

(iv) A set of syntax rules, which are otherwise well known.

---

* *Cook S.A: The complexity of Theorem providing procedures, proceedings of the third annual ACM symposium on the Theory of Computing*, New York: Association of Computing Machinery, 1971, pp. 151-158.

+ Garey M.R. and Johnson D.S. : Computers and Intractability: A guide to the Theory of NP-Completeness, *H.Freeman, New York, 1979.*

For example

$((x_1 \wedge x_2) \vee \neg x_3)$ is (legal) Boolean expression.

Next, we explain other concepts involved in SAT.

Truth Assignment: For each of the variables involved in a given Boolean expression, associating a value of either 0 or 1, gives a truth assignment, which in turn gives a truth-value to the Boolean expression.

For example: Let $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$ be one of the eight possible assignments to a Boolean expression involving x_1, x_2 and x_3

Truth-value of a Boolean expression.

Truth value of $((x_1 \wedge x_2) \vee \neg x_3)$ for the truth–assignment $x_1=0$, $x_2=1$ and $x_3=1$ is $((0 \wedge 1) \vee \neg 1) = (0 \vee 0) = 0$

Satisfiable Boolean expression: A Boolean expression is said to be satisfiable if at least one truth assignment makes the Boolean expression True.

For example: $x_1=1$, $x_2=0$ and $x_3=0$ is one assignment that makes the Boolean expression $((x_1 \wedge x_2) \vee \neg x_3)$ True. Therefore, $((x_1 \wedge x_2) \vee \neg x_3)$ is satisfiable.

Problem 2: CSAT or CNFSAT Problem: given a Boolean expression in CNF, is it satisfiable?

Explanation: A Boolean formula FR is said to be in Conjunctive Normal Form (i.e., CNF) if it is expressed as $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ where each C_i is a disjunction of the form

$$x_{i1} \vee x_{i2} \vee \ldots \vee x_{im}$$

where each x_{ij} is a literal. A **literal** is either a variable x_i or negation $\overline{x_i}$ of variable x_i.

Each C_i is called a **conjunct.** It can be easily shown that every logical expression can equivalently be expressed in CNF

Problem 3: Satisfiability (or for short, 3SAT) Problem: given a Boolean expression in 3-CNF, is it satisfiable?

Further Explanation: If each conjunct in the CNF of a given Boolean expression contains exactly three distinct literals, then the CNF is called 3-CNF.

Problem 4: Primality problem: given a positive integer n, is n prime?

Problem 5: Traveing salesman Problem (TSP)

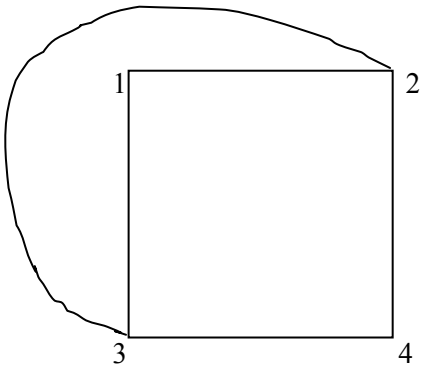
Given a set of cities $C = \{C_1, C_2, \ldots C_n\}$ with $n > 1$, and a function d which assigns to each pair of cities (C_i, C_j) some cost of travelling from C_i to C_j. Further, a positive integer/real number B is given. The problem is to find a route (covering each city exactly once) with cost at most B.

Problem 6: Hamiltonian circuit problem (H C P) given an undirected graph G = (V, E), does G contain a Hamiltonian circuit?

Further Explanation: A Hamiltonian circuit of a graph G = (V, E) is a set of edges that connects the nodes into a single cycle, with each node appearing exactly once. *We may note that the number of edges on a Hamiltonian circuit must equal the number of nodes in the graph.*

Further, it may be noted that HCP is a special case of TSP in which the cost between pairs of nodes is the same, say 1.

Example: **Consider the graph**



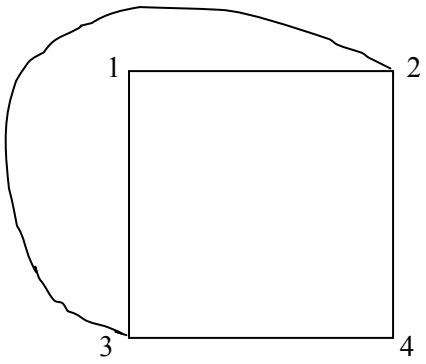
Then the above graph has one Hamiltonian circuit viz., (1, 2, 4, 3, 1)

Problem 7: **The vertex cover problem (V C P) (also known as Node cover problem): Given a graph G = (V,E) and an integer k, is there a vertex cover for G with k vertices?**

Explanation: A vertex cover for a graph G is a set C of vertices so that each edge of G has an endpoint in G. For example, for the graph shown above, {1, 2, 3} is a vertex cover. It can be easily seen that every superset of a vertex cover of a graph is also a vertex cover of the graph.

Problem 8: **K-Colourability Problem: Given a graph G and a positive integer k, is there a k-colouring of G?**

Explanation: A k-colouring of G is an assignment to each vertex of one of the k colours so that no two adjacent vertices have the same color. It may be recalled that two vertices in a graph are adjacent if there is an edge between the two vertices.



As the vertices 1, 2, 3 are mutually adjacent therefore, we require at least three colours for k-colouring problem.

Problem 9: **The complete subgraph problem (CSP Complete Sub) or clique problem: Given a graph G and positive integer k, does G have a complete subgraph with k vertices?**

Explanation: For a given graph G = (V, E), two vertices v_1 and v_2 are said to be **adjacent** if there is an edge connecting the two vertices in the graph. A **subgraph** H= (V_1, E_1) of a graph G = (V, E) is a graph such that

$V_1 \subseteq V$ and $E_1 \subseteq E$. In other words, each vertex of the subgraph is a vertex of the graph and each edge of the subgraph is an edge of the graph**.**

Complete Subgraph of a given graph G is a subgraph in which every pair of vertices is adjacent in the graph.

For example in the above graph, the subgraph containing the vertices {1, 2, 3} and the edges (1, 2), (1, 3), (2, 3) is a complete subgraph or a clique of the graph. However, the whole graph is not a clique as there is no edge between vertices 1 and 4.

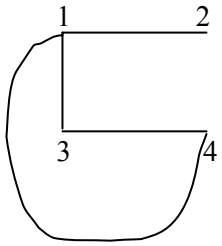
Problem 10: Independent set problem: Given a graph G = (V, E) and a positive integer k, is there an independent set of vertices with at least k elements?

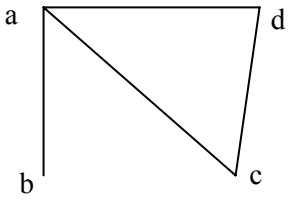
Explanation: A subset V_1 of the set of vertices V of graph G is said to be independent, if *no two* distinct vertices in V_1 are adjacent. For example, in the above graph $V_1 = \{1, 4\}$ is an independent set.

Problem 11: The subgraph isomorphism problem: Given graph G_1 and G_2, does G_1 contain a copy of G_2 as a subgraph?

Explanation: Two graphs $H_1 = (V_1, E_1)$ and $H_2 = (V_2, E_2)$ are said to be **isomorphic** if we can rename the vertices in V_2 in such a manner that after renaming, the graph H_1 and H_2 look identical *(not necessarily pictorially, but as ordered pairs of sets)*

For Example



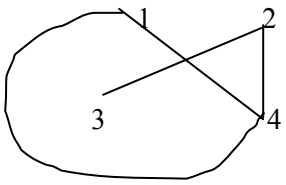


are isomorphic graph because after mapping $1 \rightarrow a$, $2 \rightarrow b$, $3 \rightarrow c$ and $4 \rightarrow d$, the two graphs become identical.

Problem 12: Given a graph g and a positive integer k, does G have an "edge cover" of k edges?

Explanation: For a given graph G = (V,E), a subset E_1 of the set of edges E of the graph, is said to be an edge cover of G, if every vertex is an end of at least one of the edges in E_1.

For Example, for the graph



The two-edge set {(1, 4), (2, 3)} is an edge cover for the graph.

Problem 13: Exact cover problem: For a given set $\mathcal{P} = \{S_1, S_2, \ldots, S_m\}$, where each S_i is a subset of a given set S, is there a subset Q of $\mathcal{P}$ *such that for each x* in S, there is exactly one S_i in Q for which x is in S_i ?

Example: Let S = {1, 2, …,10}

and $\mathcal{P}$ = { S_1, S_2, S_3, S_4, S_5} s.t
S_1 = {1, 3, 5}
S_2 = {2, 4, 6}
S_3 = {1, 2, 3, 4}
S_4 = {5, 6, 7, 9, 10}
S_5 = {7, 8, 9, 10 }
Then Q = { S_1, S_2, S_5} is a set cover for S.

Problem 14: The knapsack problem: Given a list of k integers n_1, n_2… n_k, can we partition these integers into two sets, such that sum of integers in each of the two sets is equal to the same integer?

3.4 REDUCTION, NP-COMPLETE AND NP-HARD PROBLEMS

Earlier we (informally) explained that a problem is called NP-Complete if P has at least one Non-Deterministic polynomial-time solution and further, so far, no polynomial-time Deterministic TM is known that solves the problem.

In this section, we formally define the concept and then describe a general technique of establishing the NP-Completeness of problems and finally apply the technique to show some of the problems as NP-complete. We have already explained how a problem can be thought of as a language L over some alphabet Σ. Thus the terms *problem and language* may be interchangeably used.

For the formal definition of NP-completeness, *polynomial-time reduction*, as defined below, plays a very important role.

In the previous unit, we discussed *reduction technique* to establish some of the problems as undecidable. The method *that was used for establishing undecidability* of a language using the technique of reduction, may be briefly described as follows:

Let P_1 be a problem which is *already known* to be undecidable. We want to check whether a problem P_2 is undecidable or not. If we are able to design an algorithm which transforms or constructs an instance of P_2 for each instance of P_1, then P_2 is also undecidable.

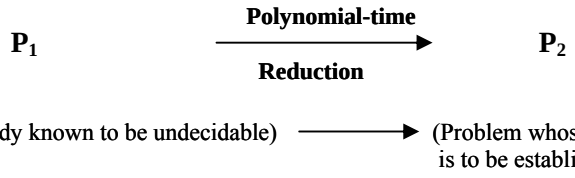
The process of transformation of the instances of the problem **already known to the undecidable to** instances of the problem, the undecidability is to checked, is called **reduction.**

Some-what similar, but, slightly different, rather special, reduction called **polynomial-time reduction** is used to establish NP-Completeness of problems.

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1.

A method of establishing the NP-Completeness (*to be formally defined later*) of a problem P_2 constitutes of designing a *polynomial time reduction* that constructs an instance of P_2 for each instance of P_1, where P_1 is already known to be NP-Complete.

The direction of the mapping must be clearly understood as shown below.



Though we have already explained the concept of NP-Completeness, yet for the sake of completeness, we give below the formal definition of NP-Compleness

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

(i) The problem L_2 is in the class NP
(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2.

In this context, we introduce below another closely related and useful concept**.**

Definition: NP-Hard Problem: A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L:

In other words, a *problem L is hard if* only condition (ii) of NP-Completeness is satisfied. But the problem has may be so hard that establishing L as an NP-class problem is so far not possible.

However, from the above definitions, it is clear that every NP-complete problem L must be NP-Hard and additionally should satisfy the condition that L is an NP-class problem.

In the next section, we discuss NP-completeness of some of problems discussed in the previous section.

3.5 ESTABLISHING NP-COMPLETENESS OF PROBLEMS

In general, the process of establishing a problem as NP-Complete is a two-step process. **The first step**, which in most of the cases is quite simple, constitutes of **guessing** possible solutions of the instances, one instance at a time, of the problem and then **verifying** whether the guess actually is a solution or not.

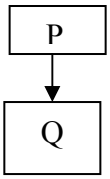
The second step involves designing a polynomial-time algorithm which reduces instances of an already known NP-Complete problem to instances of the problem, which is intended to be shown as NP-Complete.

However, to begin with, there is a major hurdle in execution of the second step. The above technique of reduction can not be applied unless we already have established at least one problem as NP-Complete. Therefore, for the first NP-Complete problem, the NP-Completeness has to be established in a different manner.

As mentioned earlier, Stephen Cook (1971) established Satisfiability as the first NP-Complete problem. The proof was based on explicit reduction of the language of any non-deterministic, polynomial-time TM to the satisfiability problem.

The proof of Satisfiability problem as the first NP-Complete problem, is quite lengthy and we skip the proof. Interested readers may consult any of the text given in the reference.

Assuming the satisfiality problem as NP-complete, the rest of the problems that we establish as NP-complete, are established by reduction method as explained above. A diagrammatic notation of the form.



Indicates: *Assuming P is already established as NP-Complete, the NP-Completeness of Q is established by through a polynomial-time reduction from P to Q.*

A scheme for establishing NP-Completeness of some the problems mentioned in Section 2.2, is suggested by *Figure. 3.1* given below:

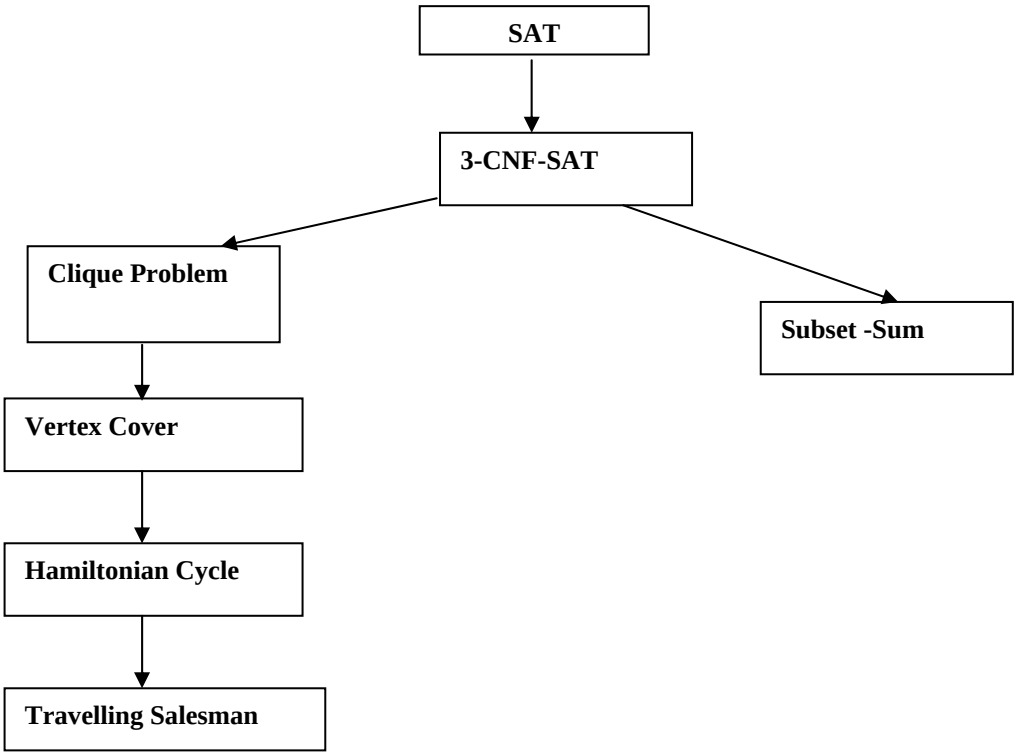


Figure: 3.1

Example 3.4.1: Show that the Clique problem is an **NP-complete** problem.

Proof : The verification of whether every pairs of vertices is connected by an edge in E, is done for different paris of vertices by a Non-deterministic TM, i.e., in parallel. Hence, it takes only polynomial time because for each of n vertices we need to verify at most n (n+1) /2 edges, the maximum number of edges in a graph with n vertices.

We next show that 3- CNF-SAT problem can be transformed to clique problem in polynomial time.

Take an instance of 3-CNF-SAT. An instance of 3CNF-SAT consists of a set of *n* clauses, each consisting of exactly 3 literals, each being either a variable or negated variable. It is satisfiable if we can choose literals in such a way that:

- at least one literal from each clause is chosen
- if literal of form *x* is chosen, no literal of form ¬x is considered.

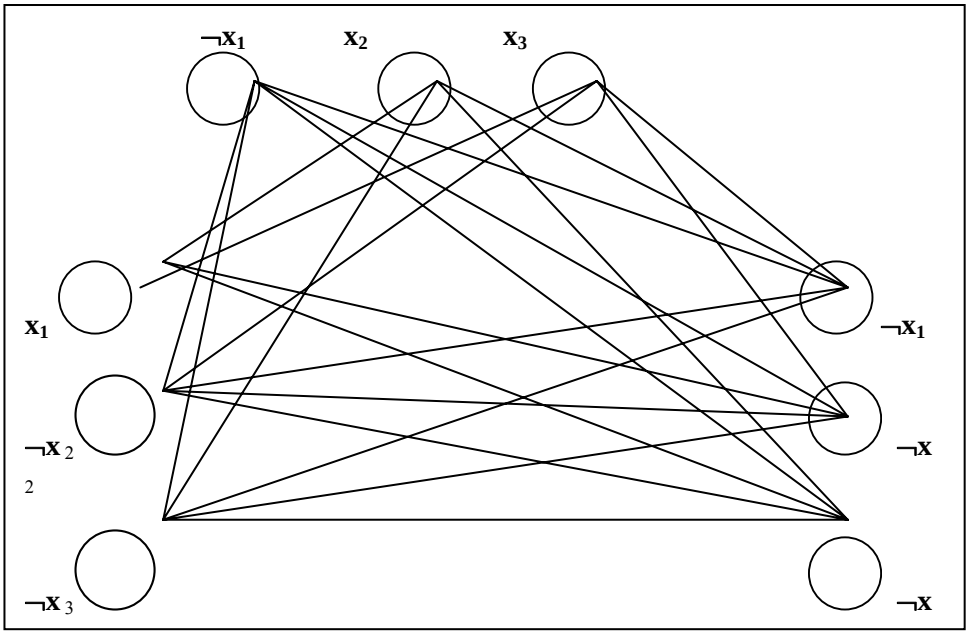


Figure: 3.2

For each of the literals, create a graph node, and connect each node to every node in other clauses, except those with the same variable but different sign. This graph can be easily computed from a boolean formula ∅ in 3-CNF-SAT in polynomial time. Consider an example, if we have–

$$\varnothing = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

then G is the graph shown in *Figure 3.2* above.

In the given example, a satisfying assignment of ∅ is ($x_1 = 0$, $x_2 = 0$, $x_3 = 1$). A corresponding clique of size k = 3 consists of the vertices corresponding to x_2 from the first clause, $\neg x_3$ from the second clause, and $\neg x_3$ from the third clause.

The problem of finding *n*-element clique is equivalent to finding a set of literals satisfying SAT. Because there are no edges between literals of the same clause, such a clique must contain exactly one literal from each clause. And because there are no edges between literals of the same variable but different sign, if node of literal *x* is in the clique, no node of literal of form ¬x is.

This proves that finding *n*-element clique in 3*n*-element graph is **NP-Complete**.

Example 5: Show that the Vertex cover problem is an **NP- complete**.

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset V' of the vertices of the graph which contains at least one of the two endpoints of each edge.

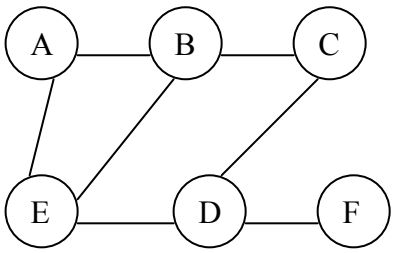


Figure: 3.3

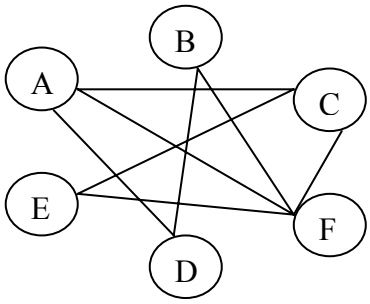


Figure: 3.4

The vertex cover problem is the optimization problem of finding a vertex cover of minimum size in a graph. The problem can also be stated as a decision problem :

VERTEX-COVER = {<*G*, *k*>| graph *G* has a vertex cover of size k }.

A deterministic algorithm to find a vertex cover in a graph is to list all subsets of vertices of size *k* and check each one to see whether it forms a vertex cover. This algorithm is exponential in *k.*

Proof : To show that Vertex cover problem $\in$ **NP**, for a given graph G = (V, E), we take V'$\subseteq$ V and verifies to see if it forms a vertex cover. Verification can be done by checking for each edge (u, v) $\in$ E whether u $\in$ V' or v $\in$ V'. This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to vertex cover problem in polynomial time. This transformation is based on the notion of the complement of a graph G. Given an undirected graph G = (V, E), we define the complement of G as G' = (V, E'), where E' = { (u, v) | (u, v) $\notin$ E}. *i.e.,* G' is the graph containing exactly those edges that are not in G. The transformation takes a graph G and *k* of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size *k* if and only if the graph G' has a vertex cover of size |V| – *k.*
Suppose that G has a clique V' $\subseteq$ V with |V'| = *k*. We claim that V – V' is a vertex cover in G'. Let (u, v) be any edge in E'. Then, (u, v) $\notin$ E, which implies that atleast one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, atleast one of u or v is in V – V', which means that edge (u, v) is covered by V – V'. Since (u, v) was chosen arbitrarily from E', every edge of E' is covered by a vertex in V – V'. Hence, the set V – V', which has size |V| – *k*, forms a vertex cover for G'.

Conversely, suppose that G' has a vertex cover V' $\subseteq$ V , where |V'| = |V| - *k*. Then, for all u, v $\in$ V, if (u, v) $\in$ E', then u $\in$ V' or v $\in$ V' or both. The contrapositive of this implication is that for all u, v $\in$ V, if u $\notin$ V' and v $\notin$ V', then (u, v) $\in$ E. In other words, V – V' is a clique, and it has size |V| – |V'| = *k.*

For example, The graph G(V,E) has a clique {A, B, E}. The complement of graph G is given by G' and have independent set given by {C, D, F}.

This proves that finding the vertex cover is **NP-Complete**.

Ex.4) Show that the Partition problem is **NP**.

Ex.5) Show that the k-colorability problem is **NP**.

Ex.6) Show that the Independent Set problem is **NP- complete**.

Ex.7) Show that the Travelling salesman problem is **NP- complete**.

3.6 SUMMARY

In this unit in number of concepts are defined.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e., to a polynomial time DTM.

Next, five **Well Known Asymptotic Growth Rate Notations are defined.**

The notation O provides asymptotic *upper bound* for a given function.
Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

$$f(x) \leq C\, g(x) \quad \text{for all } x \geq k$$

The Ω notation provides an asymptolic *lower bound* for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

$$f(x) \geq C\,(g(x)) \quad \text{whenever } x \geq k$$

The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let C > 0 **be any number**, then f(x) = o(g(x)) (pronounced as little oh of g of x) if there exists natural number k satisfying–

$$f(x) < C\, g(x) \quad \text{for all } x \geq k \geq 1$$

The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.
Further

Let C > 0 **be any number**, then

$f(x) = \omega (g(x))$

if there exist a positive integer k s.t

$f(x) > C \quad g(x) \qquad$ for all $x \geq k$

In Section 3.2 in defined, 14 well known problems, which are known to be NP-Complete.

In Section 3.3 we defined the following concepts:

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1.

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

(i) The problem L_2 is in the class NP
(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2.

Definition: NP-Hard Problem: A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L.

Finally in Section 3.4, we discussed how some of the problems defined in Section 3.2 are established as NP-Complete.

3.7 SOLUTIONS/ANSWERS

Ex.1)

$n!/n^n = (n/n)\ ((n-1)/n)\ ((n-2)/n)\ ((n-3)/n)\ldots(2/n)(1/n)$
$= 1(1-(1/n))\ (1-(2/n))\ (1-(3/n))\ldots(2/n)(1/n)$

Each factor on the right hand side is less than equal to 1 for all value of n. Hence, The right hand side expression is always less than one.

Therefore, $n!/n^n \leq 1$
or, $n! \leq n^n$
Therefore, $n! = O(n^n)$

Ex. 2)

For large value of n, $3\log n << n^2$

Therefore, $3\log n/ n^2 << 1$
$(n^2 + 3\log n)/ n^2 = 1 + 3\log n/ n^2$
or, $(n^2 + 3\log n)/ n^2 < 2$
or, $n^2 + 3\log n = O(n^2)$.

Ex.3)

We have, $2^n/5^n < 1$
or, $2^n < 5^n$
Therefore, $2^n = O(5^n)$.

Ex. 4)

Given a set of integers, we have to divide the set in to two disjoint sets such that their sum value is equal .

A deterministic algorithm to find two disjoint sets is to list all possible combination of two subsets such that one set contain *k* elements and other contains remaining (n–k) elements. Then to check if the sum of elements of one set is equal to the sum of elments of another set. Here, the possible number of combination is C(n, *k*). This algorithm is exponential in n.

To show that the partition problem $\in$ **NP**, for a given set S, we take $S_1 \subseteq S$, $S_2 \subseteq S$ and $S_1 \cap S_2 = \varnothing$ and verify to see if the sum of all elements of set S_1 is equal to the sum of all elements of set S_2. This verification can be done in polynomial time.

Hence, the partition problem is **NP**.

Ex. 5)

The graph coloring problem is to determine the minimum number of colors needed to color given graph G(V, E) vertices such that no two adjacent vetices has the same color. A deterministic algorithm for this requires exponential time.

If we cast the graph–coloring problem as a decision problem *i.e*., can we color the graph G with k-colors such that no two adjacent vertices have same color ? We can verify that if this is possible then it is possible in polynomial time.

Hence, The graph –coloring problem is **NP**.

Ex. 6)

An independent set is defined as a subset of a vertices in a graph such that no two vertices are adjacent.

The independent set problem is the optimization problem of finding an independent set of maximum size in a graph. The problem can also be stated as a decision problem :

INDEPENDENT-SET = {<*G*, *k*>| *G* has an independent set of atleast size k}.

A deterministic algorithm to find an independent set in a graph is to list all subsets of vertices of size *k* and check each one to see whether it forms an independent set. This algorithm is exponential in *k*.

Proof : To show that the independent set problem $\in$ **NP**, for a given graph G = (V, E), we take $V' \subseteq V$ and verifies to see if it forms an independent set. Verification can be done by checking for $u \in V'$ and $v \in V'$, does $(u,v) \in E$. This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to independent set problem in polynomial time.The transformation is similar clique to vertex

cover. This transformation is based on the notion of the complement of a graph G. Given an undirected graph G = (V, E), we define the complement of G as G' = (V, E'), where E' = { (u, v) | (u, v) ∉ E}. *i.e.,* G' is the graph containing exactly those edges that are not in G. The transformation takes a graph G and *k* of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size *k* if and only if the graph G' has an independent set of size |V| - *k*.

Suppose that G has a clique V' ⊆ V with |V'| = *k*. We claim that V – V' is an independent set in G'. Let (u, v) be any edge in E'. Then, (u, v) ∉ E, which implies that atleast one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, atleast one of u or v is in V – V', which means that edge (u, v) is covered by V – V'. Since (u, v) was chosen arbitrarily from E', every edge of E' is covered by a vertex in V – V'. So, either u or v is in V – V' and no two adjacent vertices are in V – V'. Hence, the set V – V', which has size |V| - *k*, forms an independent set for G'.

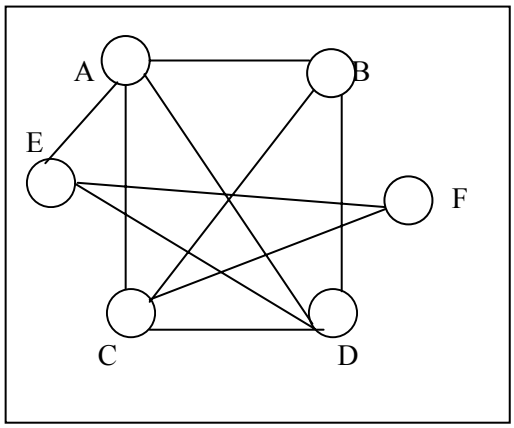


Figure: 3.5

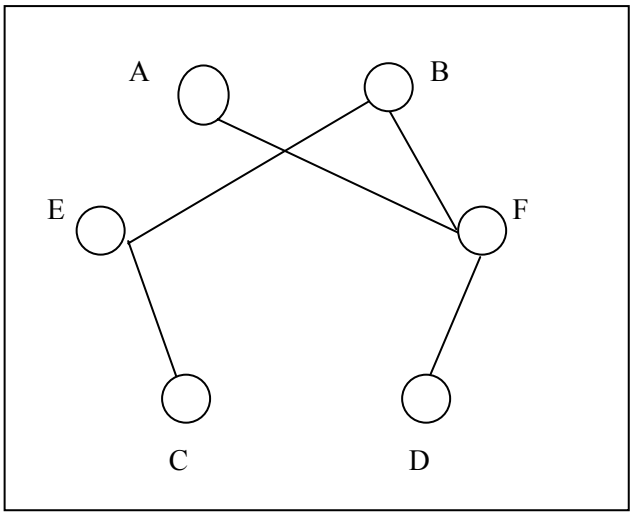


Figure: 3.6

For example, The graph G(V,E) has a clique {A, B, C, D} given by *Figure 3.5.* The complement of graph G is given by G'and have independent set given by {E F}.

This transformation can be performed in polynomial time. This proves that finding the independent set problem is **NP-Complete**.

Ex.7)

Proof : To show that travelling salesman problem $\in$ **NP**, we show that verification of the problem can be done in polynomial time. Given a constant M and a closed circuit path of a weighted graph G = (V, E) . Does such path exists in graph G and total weight of such path is less than M ?, Verification can be done by checking, does (u,v) $\in$ E and the sum of weights of these edges is less than M. This verification can be done in polynomial time.

Now, We show that Hamiltonian circuit problem can be transformed to travelling problem in polynomial time. It can be shown that , Hamiltonian circuit problem is a special case of the travelling salesman problem. Towards this goal, given any Graph G(V, E), we construct an instance of the |V|-city Travelling salesman by letting $d_{ij} = 1$ if $(v_i, v_j) \in E$, and 2 otherwise. We let the cost of travel M equal to |V|. It is immediate that there is a tour of length M or less if and only if there exists a Hamiltonian circuit in G.

Hence, The travelling salesman is **NP-complete**.

3.8 FURTHER READINGS

1. *Elements of the Theory of Computation*, *and Computation* H.R. Lewis & C.H.Papadimitriou, PHI, (1981).

2. *Introduction to Automata Theory, Languages and Computation*, J.E. Hopcroft, R.Motwani & J.D.Ullman, (II Ed.) Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language, and Computation,* J.E. Hopcroft and J.D. Ullman: Narosa Publishing House (1987).

4. *Introduction to Languages and Theory of Computation*, J.C. Martin: Tata-Mc Graw-Hill (1997).

5. *Discrete Mathematics and Its Applications* (*Fifth Edition*)K.N. Rosen: Tata McGraw-Hill (2003).

6. *Introduction to Alogrithms* (*Second Edition*) T.H. Coremen, C.E. Leiserson & C. Stein, Prentice-Hall of India (2002).