# Master of Computer Application

**(Open and Distance Learning Mode)**

**Semester – I**

# Mathematical Foundation for Computer Application

**Centre for Distance and Online Education (CDOE)**

**DEVI AHILYA VISHWAVIDYALAYA, INDORE**

**“A+” Grade Accredited by NAAC**

IET Campus, Khandwa Road, Indore - 452001

www.cdoedavv.ac.in

www.dde.dauniv.ac.in

**SLM Author(s)**

**Mr. Tapesh Sarsodiya**
B.E., M.E.
IET, Devi Ahilya Vishwavidyalaya, Indore – 452001

**Mrs. Sunita Goud**
M. Tech.
SCS, Devi Ahilya Vishwavidyalaya, Indore – 452001

**Copyright** : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
**Edition** : 2022 (Restricted Circulation)
**Published by** : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
**Printed at** : University Press, Devi Ahilya Vishwavidyalaya, Indore – 452001

# Mathematical Foundation for Computer Application

# CONTENTS

# INTRODUCTION

NOTES

Mathematics is, perhaps, the most important subject for achieving excellence in any field of science or commerce. The book has been structured to define the key mathematical concepts and its formulations by providing helpful and relevant material in lucid, self-explanatory and simple language to help you to understand the basic concepts and achieve your goals. It can be used by you as an introduction to the underlying ideas of mathematics that are applicable to computer science as well.

This book, *Mathematical Foundations of Computer Science*, is divided into five units. The first unit introduces the concept of algorithms, its properties and characteristics. It also discusses the advantage of logarithmic algorithms over linear algorithms. The next unit covers the various features of graphs, its types and operations. The third unit deals with the types of tree structures and the various situations in which they are applied. The next unit explains the concept of recursion and the recursive procedures. The last unit discusses the basics of the number theory.

The topics are logically organized and explained with related mathematical theorems, analysis and formulations to provide a background for statistical thinking and analysis with good knowledge of calculus. The interactive examples have also been carefully designed so that you can gradually build up your knowledge and understanding.

The key features of this book are:

- It balances theory with applications.
- The theorems and proofs are followed by solved exercises.
- Simplified notations and techniques of mathematical methods to make the text easy to understand.
- The book motivates new concepts with the extensive use of examples.
- The mathematical applications provided will help you to understand mathematics in action and to contextualize what you are actually learning.
- The text facilitates understanding of key mathematical concepts and their application in solving problems.

The book follows the self-instructional mode wherein each unit begins with an Introduction to the topic. The Unit Objectives are then outlined before going on to the presentation of the detailed content in a simple and structured format. Check Your Progress questions are provided at regular intervals to test the student's understanding of the subject. A Summary, a list of Key Terms and a set of Questions and Exercises are provided at the end of each unit for recapitulation.

# UNIT 1 ALGORITHMS

NOTES

**Structure**

## 1.0 INTRODUCTION

NOTES

Informally, an algorithm refers to any well-defined computational procedure that takes some values as 'input' and produces some value or set of values as 'output'. It is composed of a finite set of steps, each of which may require one or more operations. Every operation may be characterized as either simple or complex. Operations performed on scalar quantities are termed simple, while those performed on vector data are normally termed as complex. An algorithm can also be viewed as a tool for solving a well-specified 'computational problem'. The statement of the problem specifies, in general terms, the desired input/output relationship.

In simple terms, an algorithm can be defined as a step-by-step procedure for performing some task in a finite amount of time.

For a given problem, there are several ways of designing an algorithm, but the best way is the one that executes the algorithm fast. The most commonly used design approaches include, incremental approach, divide and conquer approach, dynamic programming, greedy strategy, branch and bound, backtracking and randomized algorithms.

Algorithms are used in a broad spectrum of computer applications. Algorithms to sort, search and process text, solve graph problems and basic geometric problems, display graphics and perform common mathematical calculations are extensively studied and are considered a necessary component of computer science.

## 1.1 UNIT OBJECTIVES

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

- Describe the basic features of algorithms
- Compute exponentiation
- Describe the meaning and implementation of linear search
- Describe the meaning and implementation of binary search
- Understand the functions of Big O notation
- Define the best-case, worst-case and average-case situations in an algorithm
- Describe the advantages of logarithmic algorithms over linear algorithms
- Understand the various types of complexities, such as space complexity, time complexity, practical complexity, etc
- Represent an algorithm through a pseudo code
- Describe the various techniques used in amortized analysis

## 1.2 ALGORITHMS: AN INTRODUCTION

Informally, an algorithm refers to any well-defined computational procedure that takes some values as 'input' and produces some value or set of values as 'output'. An algorithm is thus a sequence of computational steps that transform the input into the output.

You can also view an algorithm as a tool for solving a well-specified 'computational problem'. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

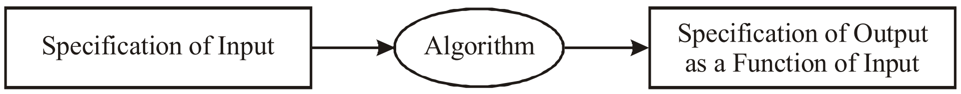
In simple terms, you can say that 'Aan algorithm is a step-by-step procedure for performing some task in a finite amount of time'.

An algorithm is composed of a finite set of steps each of which may require one or more operations. Every operation may be characterized as either a simple or complex. Operations performed on scalar quantities are termed simple, while operations on vector data normally termed as complex.

### 1.2.1 Definition, Characteristics and Properties of Algorithms

The following are some of the established definitions of algorithms:

- An algorithm is any well-defined computational procedure that takes some value, or set of values as input and provides some value, or set of values as output.
- An algorithm is a set of instructions for solving a problem.
- An algorithm is a sequence of computational steps that transforms inputs into one or more output.
- An algorithm is the essence of a computational procedure in form of step-by-step instructions.
- An algorithm is a finite set of instructions that accomplishes a particular task.



**Characteristics of Algorithms**

The following are the major features of algorithms:

- **Input:** Each algorithm should have zero or more (but only finite) data items which are supplied externally.
- **Output:** An algorithm must provide at least one data item to explain its purpose.
- **Finiteness:** An algorithm must terminate after a finite number of steps which were executed in a finite amount of time.

- **Definiteness:** Each step must be unambiguously specified and clear, i.e., each step must be precisely defined.
- **Effectiveness:** Each step should be sufficiently simple and basic.

NOTES

Algorithms that are definite and effective are also termed as computational procedures.

***What is a Good Algorithm?***

A good algorithm should be efficient in terms of the running time as well as the space utilized. An algorithm is said to be efficient, if it takes less amount of time to execute and also utilizes less amount of memory.

***Efficiency as a Function of Input Size***

Efficiency can be measured in terms of the number of bits in an input number as well as the number of data elements (numbers, points).

**Properties of Algorithm**

The following are the five important properties (features) of algorithm:

- Finiteness
- Definitiveness
- Input
- Output
- Effectiveness
- **Finiteness:** An algorithm must always terminate after a finite number of steps. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Definitiveness:** Each operation must have a definite meaning and it must be perfectly clear. All steps of an algorithm need to be precisely defined. The actions to be executed in each case should be rigorously and clearly specified.
- **Inputs:** An algorithm may have zero or more 'input' quantities. These inputs are given to the algorithm either prior to its beginning or dynamically as it runs. An input is taken from a specified set of objects. Also, it is externally supplied to the algorithm.
- **Output:** An algorithm has one or more 'output' quantities. These quantities have specified relations to the inputs. An algorithm produces at least one output.
- **Effectiveness:** Each operation should be effective, i.e., the operation must terminate in a finite amount of time.

An algorithm is usually supposed to be 'effective' in the sense that all its operations need to be sufficiently basic so that they can in principle be executed exactly the same way in a finite length of time by someone using pencil and paper.

### 1.2.2 Types of Algorithms

(i) Approximate algorithm
(ii) Probabilistic algorithm
(iii) Infinite algorithm
(iv) Heuristic algorithm

**(i) Approximate Algorithm**

An algorithm is said to approximate if it is infinite and repeating.

For example, $\sqrt{2} = 1.414$

$\sqrt{3} = 1.732$

$\pi = 3.14$, etc.

**(ii) Probabilistic Algorithm**

If the solution of a problem is uncertain, then it is called a probabilistic algorithm.

For example, Tossing of a coin

**(iii) Infinite Algorithm**

An algorithm, which is not finite, is called infinite algorithm.

For example, a complete solution of a chessboard, division by zero

**(iv) Heuristic Algorithm**

Giving less inputs and getting more outputs is called heuristic algorithm.

### 1.2.3 Areas of Research in the Study of Algorithms

Several active areas of research are included in the study of algorithms. The following four distinct areas can be identified:

1. **Devising Algorithms**

   The creation of an algorithm is an art. It may never be fully automated. A few design techniques are especially useful in fields other than computer science, such as operations research and electrical engineering. All of the approaches we consider have application in diverse areas, including computer science. But some important design techniques such as linear, non-linear and integer programming are not covered here as they are traditionally covered in other courses.

2. **Validating Algorithms**

   Once you have devised an algorithm, you need to show that it computes the correct answer for all possible legal inputs. This process is referred to as 'algorithm validation'. It is not necessary to express the algorithm as a program. If it is stated in a precise way, it will do. The objective of the validation is to assure the user that the algorithm will work correctly and independently of the issues concerning the programming language, in which it will eventually be written. After validity of the method gets checked, is shown, it is possible to write the program. On completion of program writing the second phase begins. This phase is called 'program providing' or 'program verification'. A proof of correctness requires the solution to be stated in two forms. One form is usually a program, which is annotated by a set of assertions about the input and output variables of the program. The second form is called specification and this may also be expressed in the predicate calculus. A proof shows that these two forms are equivalent for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language is precisely defined and all basic operations are proved correct. All these details may cause a proof to be very much longer than the program.

3. **Analysing Algorithms**

   As an algorithm is executed, it uses computer's central processing unit (CPU) for performing operations. It also uses the memory for holding the program and its data. Analysis of algorithm is the process of determining the computing time and storage required by an algorithm.

4. **Testing a Program**

   Testing of a program comprises of two phases: (i) Debugging and (ii) Profiling.

   (i) Debugging refers to the process of carrying out programs on sample data sets with the objective of finding faulty results. If any faulty result occurs, it is corrected by debugging. A proof of correctness is much more valuable than a thousand tests, since it guarantees that the program will work correctly for a possible input.

   (ii) Profiling refers to the process of executing a correct program on data sets and the measurement of the time and space it takes in computing the results. It is useful in the sense that it confirms a previously done analysis and points out logical places for performing useful optimization.

For example,

If you wish to measure the worst-case performance of the sequential search algorithm, we need to do the following:

- Decide the values of n for which computing time has be obtained
- Determine for each of the above value of n the data that exhibits the worst-case behaviour

### 1.2.4 Algorithm for Sequential Search

```
Algorithm seqsearch (a, x, n)
//search for x in a[l: n] . a[0] is used as additional
space
{
i := n; a[0] := x;
while(a[i] * x) do i := i  + 1;
return i;
}
```

The decision on which the values of *n* to be used is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for algorithm, our interest is to simply predict how long it will take, in the worst case, to search for `x`, given the size `n` of `a`.

NOTES

### 1.2.5 Algorithms as Technology

If computers were infinitely fast and computer memory was free, you would be in a position to adopt any correct method to solve a problem. In all likelihood, you would like your implementation to be adhering to good software engineering practice. However, you would use the method which is the easiest to implement.

However, computers may be fast, but they cannot be infinitely fast. Similarly, memory may be cheap, but it cannot be free. Thus, computing time and space in memory are bounded resources . You need to use these resources wisely. Such algorithms which are efficient in terms of time or space will be helpful.

**Efficiency**

It has been found that algorithm devices used for solving the same problem usually differ considerably in their efficiency. These differences are more significant than those due to hardware and software.

### 1.2.6 Algorithms and Other Technologies

Algorithms are important on contemporary computers which have advanced technologies, such as

- Hardware with high clock rates, pipelining and super scalar architectures
- Easy to use, intuitive Graphical User Interfaces (GUIs)
- Local Area Networking (LAN) and Wide Area Networking (WAN)

A truly skilled programmer possesses a solid algorithmic knowledge and technique. It separates him/her from a novice. It is true that with modern computing technology, you can perform some tasks even if you do not have much knowledge of algorithms. However, if you have a good background in algorithms, you can perform much better.

### 1.2.7 Measuring the Running Time of an Algorithm

**Experimental Study**

The following steps need to be carried out:

(*i*) A program should be written in a language which will implement the algorithm.

(*ii*) This program should be run with input data which is of varying size and composition.

(*iii*) Methods such as **`getTime()`** or **`System.currentTime millis()`** should be employed for obtaining an accurate value of the actual running time required by the algorithm for execution.

**Limitations of Experimental Study**

The experimental studies have the following limitations:

(i) In order to calculate the running time of the algorithm, it must be implemented and tested.

(ii) Since the experiments are carried out only with a few set of inputs, the running time calculated need not be representative of other inputs which were not part of the experimental study.

(iii) The same hardware and software platforms should be used for comparing two algorithms.

**Theoretical Analysis**

The general methodology for analysing the running time of algorithms:

- Uses a high-level description of the algorithm instead of testing one of its implementations
- Takes into account all possible inputs
- Enables evaluation of efficiency of any algorithm in a way that is independent of the hardware and software environment.

### 1.2.8 Algorithm Design Strategies

For a given problem, there are several ways to design algorithms, but the best way is the one which executes the algorithm fast, such that it operates quickly on inputs.

The following are the descriptions of several design approaches which yield good algorithms:

**Incremental Approach**

This is one of the simplest approaches of algorithm designing. In this case, whenever a new element is inserted into its appropriate place, the index is increased. You start moving from the first step executing each step one by one till you reach the end. Here, you do not split your problem.

Example includes insertion sort designed using incremental approach.

**Divide and Conquer Approach**

Some algorithms have recursion and they call themselves one or more times to deal with sub-problems in order to reach the solution. These types of algorithms follow the Divide and Conquer approach.

In this approach, you break the original problem into several sub-problems which are similar to the original problem in structure but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution of the original problem.

Traditionally, an algorithm is referred to as the 'divide and conquer' type, only if it contains at least two recursive calls.

The following are the three steps involved in this approach:

1. **Divide:** The given problem is divided into several sub-problems
2. **Conquer:** The sub-problems are solved recursively
3. **Combine:** The solutions of the sub-problems are combined to create a solution of the original problem

Examples include quick sort, merge sort, binary search, etc.

### Dynamic Programming

Dynamic programming is the most powerful design technique for optimization problems. The divide and conquer approach is applicable where sub-problems are independent. On the other hand, dynamic programming is applicable where sub-problems share sub-problems. A dynamic programming algorithm remembers past results and uses them to find new results.

Dynamic programming is generally used for optimization problems. In these problems multiple solutions exist, but we need to find the 'best' solution. This requires 'optimal sub-structure' and 'overlapping sub-problems'.

- **Optimal sub-structure:** Optimal solution contains optimal solutions to sub-problems.
- **Overlapping sub-problems:** Solutions to sub-problems can be stored and reused in a bottom-up fashion.

Examples include assembly line scheduling, matrix chain multiplication and longest common sub-sequence.

### Greedy Strategy

Greedy algorithms typically applies to optimization problems such as dynamic programming algorithms, where a set of choices must be made in order to arrive at an optimal solution. The main idea behind greedy algorithm is to make each choice in a locally optimal manner, i.e., choose the solution which looks best at the moment without considering the future results. Greedy approach provides an optimal solution for many problems much more quickly than a dynamic programming approach. In greedy algorithms, you use optimal sub-structure in a top-down fashion. Instead of first finding optimal solutions to sub-problems and then making a choice, greedy algorithms first make a choice—the choice that looks best at the time—and then solve the resulting sub-problems.

Greedy algorithms do not always guarantee optimal solutions, however, they generally produce solutions that are very close in value to the optimal.

Examples include activity selection problem, Huffman algorithm, fractional knapsack problem.

### Branch and Bound

Branch and Bound algorithm is used for finding optimal solutions of various optimization problems, especially discrete and combinational types. In Branch and Bound algorithm, a given problem which cannot be bounded has to be divided into two new restricted problems. Branch and Bound algorithms can be slow, and in worst cases they grow exponentially as the input size grows, but in some cases these algorithms perform well.

Examples include Knapsack problem, non-linear programming, maximum satisfiability problem, least cost search, 15-puzzle, and so on.

**Backtracking**

The term backtrack was first coined by D.H. Lehmer in the 1950s. If a problem has several possible choices at any stage, then you select any choice and start moving by considering that choice. If it choice solves your problem then it is good, otherwise, you backtrack, i.e., move backwards and choose some other choice and repeat the same procedure until the solution is obtained. Some sequence of choices may be a solution to your problem.

Examples include N-Queens problem, sum of subsets problem, Hamiltonian circuit problem, graph colouring, etc.

**Randomized Algorithms**

An algorithm whose input is determined by the values produced by a random number generator is a randomized or probabilistic algorithm. Such an algorithm employs a degree of randomness as part of its logic. Various decisions made in the algorithm depend on the output of the random number generator. As random number generator produces different outputs from run to run, so the output of a randomized algorithm could also differ from run to run for the same input.

The following are the two types of randomized algorithms:

*(i)* Las Vegas algorithms

*(ii)* Monte Carlo algorithms

Example includes randomized quicksort.

## CHECK YOUR PROGRESS

1. How can efficiency be measured as a function of input size?
2. What is the 'Incremental approach' to design algorithms?
3. What are the two main types of randomized algorithms?

### 1.2.9 Analysis of Algorithms

During analysis, performance of an algorithm should be evaluated by predicting how much resources the algorithm requires. You usually concentrate on determining the running time (worst case) without considering the space requirements, unless stated. So, to predict the resource requirements, you need a computational model. Popular computational models include RAM (Random Access Model), PARAM, Message Passing Model, Turing Machine, etc.

In RAM model, you have to deal with instructions which are executed one after the other and there should also be no concurrent operations. Instructions include the following:

- **Arithmetic:** Add, multiply, substract, floor, ceiling, divide
- **Shift left and shift right**
- **Data movement:** Assignment, load, copy, store
- **Logical:** Comparison
- **Control:** Conditional/unconditional branching, subroutine call, return

NOTES

These instructions are called the primitive operations. Primitive operations are low-level operations which are independent of the programming language. They can be identified in the pseudocode.

There is no generally accepted set of rules for the analysis of algorithms. You can perform analysis by counting the number of primitive operations in the algorithm.

By analysing the pseudocode, you are able to count the number of primitive operations executed by an algorithm.

In Example 1.1 the algorithm which determines the maximum elements from a set of elements given in an array of size n is given. Determine the number of primitive operations required.

For example,

```
MAXIMUM (A, n)
    1. current_max ¬ A[0]
    2. for i ¬ 1 to n - 1 do
    3. if current_max < A[i]
    4. then current_max ¬ A[i]
       {increment counter i}
    5. return current_max
```

No. of primitive operations $= 2 + 1 + n + 4(n - 1) + 1 = 5n$ (At least)

$= 2 + 1 + n + 6(n - 1) + 1 = 7n - 2$ (At most)

Consider the following example for insertion sort, which is a very efficient algorithm for sorting a small number of elements.

**Insertion sort:** It is a very good algorithm for sorting or arranging, either in the increasing or decreasing order for small number of elements.

In this case the sequence of numbers which are to be sorted and output is a sorted sequence

```
 Insertion-Sort (A)
1.for i ← 2 to length A [i] do
2.item ← A [i]
3.//Insert A [i] into the sorted sequence A [1... i - 1]
4.j ← i -1
```

```
5. while j > 0 and A [j] > item do
6. A [j + 1] ← A [j]
7. j ← j - 1
8. A [j + 1] ← item
```

NOTES

In this algorithm, Steps 2 to 8 are under for loop construct which indicates indentation. Similarly, Steps 6 and 7 are under while loop construct. We have taken `i, j` items as local variables in this procedure. The input to this algorithm is an array which is shown in brackets after the name of the algorithm, and here the input is an array of some numbers which are to be sorted.

Consider the following example to understand how insertion sort works:

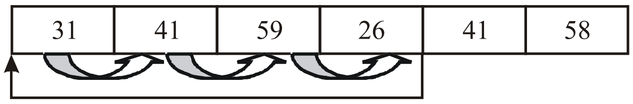
| | | | | | |
|---|---|---|---|---|---|
| 31 | 41 | 59 | 26 | 41 | 58 |

**Dry run:** The length of the array is 6 since there are six elements.

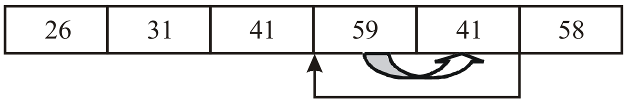
| i | item | j | A[j] | |
|---|---|---|---|---|
| 2 | 41 | 1 | 31 | Exits from while loop |

| | | | | | |
|---|---|---|---|---|---|
| 31 | 41 | 59 | 26 | 41 | 58 |

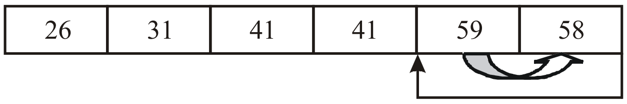
| | | | | |
|---|---|---|---|---|
| 3 | 59 | 2 | 41 | Exits from while loop |



| | | | | |
|---|---|---|---|---|
| 4 | 26 | 3 | 59 | Enters into while loop |
| | | 2 | 41 | Enters into while loop |
| | | 1 | 31 | Enters into while loop |
| | | 0 | | Enters from while loop |



| | | | | |
|---|---|---|---|---|
| 5 | 41 | 4 | 59 | Enters into while loop |
| | | 3 | 41 | Exits from while loop |



| | | | | |
|---|---|---|---|---|
| 6 | 58 | 5 | 59 | Enters into while loop |
| | | 4 | 41 | Exits from while loop |

So, the final sorted array is,

| | | | | | |
|---|---|---|---|---|---|
| 26 | 31 | 41 | 41 | 58 | 59 |

Each round of iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input.

**Analysis of insertion sort:** Each step is associated with two factors, namely, cost and frequency.

**Cost:** The amount of time a particular step takes during execution which is a constant quantity denoted by $c_1, c_2, c_3, c_4 \ldots\ldots.$

**Frequency:** The number of times a particular step executes

***Note:*** The main step of the looping constructs executes one time more than its internal statements. As in the above example, Step 1 executes seven times, Steps 2 to 4 and Step 8 executes six times. Likewise, Step 5 executes one more time than Steps 6 and 7 because at last it checks for the condition which becomes false.

Consider that the number of elements in an array is n. So, `length [A] = n.`

Let $t_i$ be the number of times the **`while`** loop test in line 5 is executed for that value of **`i`**.

| | Insertion Sort(A) | Cost | Frequency |
|---|---|---|---|
| 1 | **for** i ← 2 **to** length[A] | $c_1$ | $n$ |
| 2 | **do** item ← A[i] | $c_2$ | $n-1$ |
| 3 | //Insert A[i] into the sorted sequence A [1..i – 1] | 0 | $n-1$ |
| 4 | j ← i – 1 | $c_4$ | $n-1$ |
| 5 | **while** j > 0 and A[j] > item | $c_5$ | $\sum_{i=2}^{n} t_i$ |
| 6 | **do** A[j + 1] ← A[j] | $c_6$ | $\sum_{i=2}^{n} t_i-1$ |
| 7 | j ← j – 1 | $c_7$ | $\sum_{i=2}^{n} t_i-1$ |
| 8 | A[j + 1] ← item | $c_8$ | $n-1$ |

The running time of the algorithm denoted by ***T*(*n*)** is the sum of the running times for each step. A statement whose cost is $c_i$ and frequency is ***n*** will contribute $c_i n$ to the total running time. To compute ***T*(*n*)**, the running time of **`INSERTION-SORT`**, we sum the products of the Cost and Frequency columns obtaining,

$$\mathbf{T}(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$$

**Best case:** It is the function defined by the minimum number of steps taken on any input of size ***n***. It gives the minimum value of ***T(n)*** for any possible input data. Best case provides a lower bound on the running time for any input and occurs when minimum number of steps are executed, i.e., the while loop condition is always false or the array is already sorted. For that case, $\boldsymbol{t_i = 1}$.

$$\sum_{i=2}^{n} t_i = n-1$$

So,

$$\mathrm{T}(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$\mathrm{T}(n) = (c_1 + c_2 + c_4 + c_5 + c_8)\, n - (c_2 + c_4 + c_5 + c_8)$$
$$\mathrm{T}(n) = an + b$$

$$\boxed{\mathrm{T}(n) = \mathrm{O}(n)}$$

For best case, the running time of insertion sort is a linear function in ***n***.

**Worst case:** It is the function defined by the maximum number of steps taken on any input size of size ***n***. It gives the maximum value of ***T(n)*** for any possible input data. Worst case provides an upper bound on the running time for any input and gives you a guarantee that the algorithm will never take longer time.

You usually concentrate on finding the worst case running time, i.e., in searching algorithms worst case occurs when you try to find a number but the number is not present. Likewise, worst case occurs when the maximum number of steps are executed, i.e., the while loop condition always leads to true or the array elements are given in decreasing order. For that case, $\boldsymbol{t_i = j}$.

Hence,

$$\begin{aligned}
\mathrm{T}(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} j + c_6 \sum_{i=2}^{n} (j-1) \\
&\quad + c_7 \sum_{i=2}^{n} (j-1) + c_8(n-1) \\
\mathrm{T}(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5((n(n+1)/2) - 1) \\
&\quad + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1) \\
\mathrm{T}(n) &= (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 \\
&\quad - c_7/2 + c_8)\mathrm{n} - (c_2 + c_4 + c_5 + c_8) \\
\mathrm{T}(n) &= an^2 + bn + c
\end{aligned}$$

$$\boxed{\mathrm{T}(n) = \mathrm{O}(n^2)}$$

For worst case, the running time of insertion sort is a quadratic function in **n**.

**Average case:** It is the function defined by the average number of steps taken on any input of size ***n***. It gives the expected value of **T(*n*)**. Generally, you do not analyse the average case, because it is often as bad as the worst case. This case lies between the best and worst cases. Hence, in this case, $t_i = i/2$ or $(i + 1)/2$ or $(i - 1)/2$.

$$\boxed{T(n) = O(n^2)}$$

NOTES

***Note:*** During analysis we drop the lower contributing terms and the coefficients to do analysis for large ***n***.

**Analysis of Some Other Algorithms:**

| **MATRIX-ADD (A, B, C, m, n)** | **Cost** | **Frequency (Times)** |
|---|---|---|
| 1. **for** i ← 1 to m | $c_1$ | $m + 1$ |
| 2. **do** j ← 1 to n | $c_2$ | $m(n + 1)$ |
| 3. **do** C[i, j] ← A[i, j] + B[i, j] | $c_3$ | $m.n$ |

So, $T(n) = c_1(m + 1) + c_2(m(n + 1)) + c_3.mn$

$T(n) = c_1m + c_1 + c_2.mn + c_2m + c_3.mn$

$T(n) = (c_2 + c_3).mn + (c_1 + c_2)m + c_1$

$T(n) = O(mn)$

| **SUM (A, n)** | **Cost** | **Frequency (Times)** |
|---|---|---|
| 1. **sum** ← 0 | $c_1$ | 1 |
| 2. **for** i ← 1 to n | $c_2$ | $n + 1$ |
| 3. **do** sum ← sum + A[i] | $c_3$ | $n$ |
| 4. **return** sum | $c_4$ | 1 |

So, $T(n) = c_1.1 + c_2.(n + 1) + c_3.n + c_4.1$

$T(n) = (c_2 + c_3).n + (c_1 + c_2 + c_4)$

$T(n) = an + b$

$T(n) = O(n)$

To have a good best case running time, the algorithm should be modified so that it tests whether the input satisfies some special case condition, and if it does so, then outputs a pre-computed answer. The best case running time is generally not a good measure of an algorithm.

### 1.2.10 Merits and Demerits of Algorithm

Many algorithms are used in a broad spectrum of computer applications. Such elementary algorithms are extensively studied and are considered a necessary component of computer science. Examples of these algorithms include algorithms to sort, search and process text, solve graph problems and basic geometric problems, and display graphics and perform common mathematical calculations.

Sorting is useful in arranging data objects in a specific order, e.g., in numerically ascending or descending order. Sorting may be internal or external.

NOTES

Using internal sorting, you can arrange data stored internally in a computer's memory. Simple algorithms for sorting by selection, exchange or insertion are easy to understand and straightforward to code. However, in case the number of objects to be sorted is large, simple algorithms would not be helpful as they are usually very slow. In such cases, you need a more sophisticated algorithm, such as heap sort or quick sort, to achieve acceptable performance. Using external sorting, you can arrange data records that are stored.

Searching for data means looking for a desired data object in a group of data objects. Elementary searching algorithms comprise of linear search and binary search. In linear search, a sequence of data objects is examined one by one. In binary search, on the other hand, a more sophisticated strategy for searching data is adopted. While searching a large array, binary search works faster than linear search. You can also store the collection of data objects as a tree that need to be searched frequently. If such a tree is properly structured, searching the tree would be very efficient.

A sequence of characters is termed as a text string. In a word processing system, efficient algorithms for manipulating text strings, such as algorithms for organizing text data into lines and paragraphs and searching for occurrences of a given pattern in a document, are necessary. A source program in a high-level programming language is a text string. Text processing is one of the essential tasks of a compiler. A compiler uses efficient algorithms to perform lexical analysis and parsing. When individual characters are grouped into meaningful words or symbols, it is termed as lexical analysis. When the syntactical structure of a source program is recognized, it is termed as parsing.

A graph is used in modelling a group of interconnected objects. A graph representing a set of locations connected by routes for transportation is a good example. Graph algorithms are used to solve such problems which deal with objects and their connections, such as determining whether or not all locations are connected, visiting all locations that are accessible from a given location, or determining the shortest path from one location to another.

Mathematical algorithms are widely applied in science and engineering. Algorithms to generate random numbers, perform operations on matrices, solve simultaneous equations and numerical integration, etc., are examples of basic algorithms for mathematical computations. In the modern programming languages, predefined functions are usually provided for many common computations, such as random number generation, logarithm, exponentiation and trigonometric functions.

There are applications in which a computer program has to adapt to a change in its environment so as to continue performing well. Using a self-organizing data structure, which gets reorganized at regular intervals, such that those components which are most likely to be accessed are placed where they can be accessed most efficiently, is a common approach to make a computer program adaptive. A self-modifying algorithm that adapts itself is also conceivable. In order to develop adaptive computer programs, biological evolution has given impetus to evolutionary computation methods, such as genetic algorithms.

Some applications need a large amount of computations in a timely manner. For saving time, you need to develop a parallel algorithm which uses many processors simultaneously and thus quickly solves a given problem. The basic idea is that the given problem is divided into sub-problems and each processor is used to solve a sub-problem. The processors usually have to communicate among themselves so as to facilitate cooperation. For communicating with one another, the processors may share memory. Alternatively, they may be connected by communication links into some type of network, such as a hypercube.

### 1.2.11 Flowchart and Algorithms

In the beginning, the use of flowcharts was restricted to electronic data processing for representing the conditional logic of computer programs. The1980s witnessed the emergence of structured programming and structured design. As a result of this, in database programming, data flow and structure charts began to replace flowcharts. With the widespread adoption of such ALGOL-like computer languages as Pascal, textual models like pseudocode are being used frequently for representing algorithms. Unified Modeling Language (UML) started the synthesis and codification these modelling techniques in the 1990s.

A flowchart refers to a graphical representation of a process which depicts inputs, outputs and units of activity. It represents the whole process at a high or detailed (depending on your use) level of observation. It serves as an instruction manual or a tool to facilitate a detailed analysis and optimization of workflow as well as service delivery.

Flowcharts have been in use since long. Nobody can be specified as the 'father of the flowchart'. It is possible to customize a flowchart according to need or purpose. This is why flowcharts are considered a very unique quality improvement method for representing data.

#### Symbols

A typical flowchart has the following types of symbols:

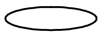
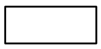
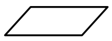
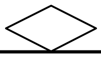
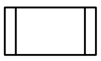
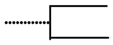
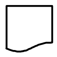
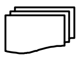
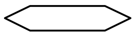
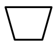
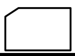
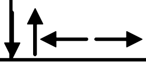
- **Start and end symbols**: They are represented as ovals or rounded rectangles, normally having the word 'Start' or 'End'.
- **Arrows:** They show the 'flow of control' in computer science. An arrow coming from one symbol and ending at another symbol shows the transmission of control to the symbol the arrow is pointing to.
- **Processing steps**: They are represented as rectangles.

  **Example:** Add 1 to X.
- **Input/Output symbol**: It is represented as a parallelogram.

  **Examples:** Get X from the user; display X.

- **Conditional symbol**: It is represented as a diamond (rhombus). It has a Yes/No question or True/False test. It contains two arrows coming out of it, normally from the bottom and right points. One of the arrows corresponds to Yes or True, while the other corresponds to No or False. These two arrows make it unique.

There are also other symbols in flowcharts may contain, e.g., connectors. Connectors are normally represented as circles. They represent converging paths in the flowchart. Circles contain more than one arrow. However, only one arrow goes out. Some flowcharts may just have an arrow point to another arrow instead. Such flowcharts are useful in representing an iterative process, what is known as a loop in terms of computer science. A loop, for example, comprises a connector where control first enters processing steps, a conditional with one arrow exiting the loop, and another going back to the connector.

| **Shape** | **Symbol** | **Symbol Name** | **Purpose** |
|---|---|---|---|
| Oval |  | Terminator | To represent the begin/end or start/stop of a flow chart |
| Rectangle |  | Process | To represent calculations and data manipulations |
| Parallelogram |  | Data | To represent Input/Output data |
| Diamond |  | Decision | To represent a decision or comparison control flow |
| Double sided Rectangle |  | Predefined Process | To represent Modules or set of operations or a function |
| Bracket with broken line |  | Annotation | To represent descriptive comments or explanations |
| Document |  | Print out | To represent output data in the form a document |
| Multiple documents |  | Print outs | To represent output data in the form of multiple documents |
| Circle | | Connector | To connect different parts of the flow chart |
| Hexagon |  | Repetition/ Looping | To represent a group of repetitive statements |
| Trapezoid |  | Manual Operation | To represent an operation which is done manually |
| Card |  | Card | To represent a card. E.g., punched card |
| Arrows |  | Flows of control | To represent the flow of the execution |

It is now used at the beginning of the next line or page with the same number. Thus, a reader of the chart is able to follow the path.

**Instructions**

NOTES

The following is the step-by-step process for developing a flowchart:

**Step 1:** Information on how the process flows is gathered. For this, the following tools are used:

- Conservation
- Experience
- Product development codes

**Step 2:** The trial of process flow is undertaken.

**Step 3:** Other more familiar personnel are allowed to check for accuracy.

**Step 4:** If necessary, changes are made.

**Step 5:** The final actual flow is compared with the best possible flow.

**Construction/Interpretation tips for a flowchart**

- The boundaries of the process should be defined unambiguously.
- The simplest symbols should be used.
- It should be ensured that each feedback loop contains an escape.
- It should be ensured that there is only one output arrow out of a process box. Otherwise, it would require a decision diamond.

**Types of Flowcharts**

A flowchart is common type of chart representing an algorithm or a process and showing the steps as boxes of different kinds and their order by connecting these with arrows. We use flowcharts to analyse, design, document or manage a process or program in different fields.

There are many different types of flowcharts. On the one hand, there are different types for different users, such as analysts, designers, engineers, managers or programmers. On the other hand, those flowcharts can represent different types of objects. Sterneckert (2003) divides four more general types of flowcharts:

- Document flowcharts showing a document flow through system
- Data flowcharts showing data flows in a system
- System flowcharts showing controls at a physical or resource level
- Program flowchart showing the controls in a program within a system

However, there are several of these classifications. For example, Andrew Veronis named three basic types of flowcharts: the system flowchart, the general flowchart, and the detailed flowchart. Marilyn Bohl (1978) stated ‘in practice, two kinds of flowcharts are used in solution planning: system flowcharts and program flowcharts...'. More recently, Mark A. Fryman (2001) stated that there are more differences. Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts and process flowcharts are just a few of the different types of flowcharts that are used in business and government.

**Interpretation**

- Analyse flowchart of the actual process
- Analyse flowchart of the best process
- Compare both charts looking for areas where they are different. Most of the time, the stages where differences occur are considered to be the problem area or process.
- Take appropriate in-house steps to correct the differences between the two separate flows.

**Example:** Process flowchart—Finding the best way home

This is a simple case of processes and decisions in finding the best route home at the end of the working day.

A flowchart provides the following:

- **Communication:** Flowcharts are excellent means of communication. They quickly and clearly impart ideas and descriptions of algorithms to other programmers, students, computer operators and users.
- **An overview:** Flowcharts provide a clear overview of the entire problem and its algorithm for solution. They show all major elements and their relationships.
- **Algorithm development and experimentation:** Flowcharts are a quick method of illustrating program flow. It is much easier and faster to try an idea with a flowchart than to write a program and test it on a computer.
- **Check program logic:** Flowcharts show all major parts of a program. All details of program logic must be classified and specified. This is a valuable check for maintaining accuracy in logic flow.
- **Facilitate coding:** A programmer can code the programming instructions in a computer language with more ease with a comprehensive flowchart as a guide. A flowchart specifies all the steps to be coded and helps to prevent errors.
- **Program documentation:** A flowchart provides a permanent recording of program logic. It documents the steps followed in an algorithm.

**Advantages of Flowcharts**

- Clarify the program logic.
- Before coding begins, a flowchart assists the programmer in determining the type of logic control to be used in a program.

- Serve as documentation.
- Serve as a guide for program coding of program writing.
- A flowchart is a pictorial representation that may be useful to the businessperson or user who wishes to examine some facts of the logic used in a program.
- Help to detect deficiencies in the problem statement.

**Limitations of Flowcharts**

- Program flowcharts are bulky for the programmer to write. As a result many programmers do not write the chart until after the program has been completed. This defeats one of its main purposes.
- It is sometimes difficult for a business person or user to understand the logic depicted in a flowchart.
- Flowcharts are no longer completely standardized tools. The newer structured programming techniques have changed the traditional format of a flowchart.

**Differences between Flowcharts and Algorithms**

***Flowchart***

- It is the graphical representation of the solution to a problem.
- It is connected with the shape of each box indicating the type of operation being performed. The actual operation, which is to be performed, is written inside the symbol. The arrow coming out of symbol indicates which operation to perform next.

***Algorithm***

- It is a process for solving a problem.
- It is constructed without boxes in a succession of steps.

## Ways to Write an Algorithm

An algorithm can be written in the following three ways:

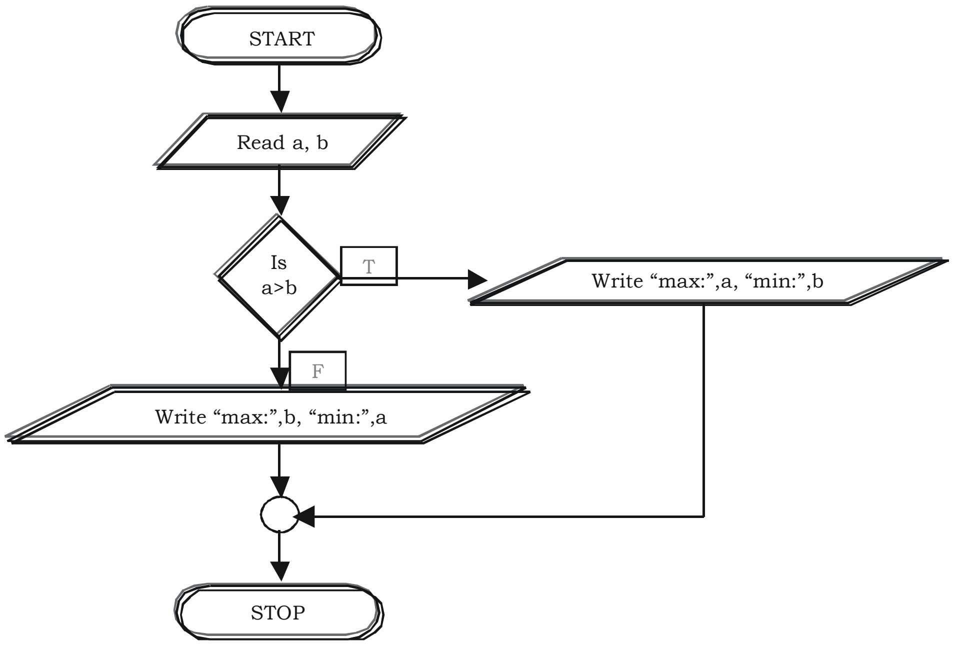
- **Straight Sequential:** A series of steps that can be performed one after the other
- **Selection or Transfer of Control:** Making a selection of a choice from two alternatives of a group of alternatives
- **Iteration or Looping:** Performing repeated operations

The following are the examples of algorithms and flowcharts for some different problems:

## Examples of Straight Sequential Execution

**Example 1.1:** Write a flowchart to find the maximum and minimum of given numbers.

NOTES



**Example 1.2:** Write the various steps involved in executing a 'C' program and illustrate it with the help of a flowchart.

**Solution:** Executing a program written in C involves a series of steps. They are as follows:

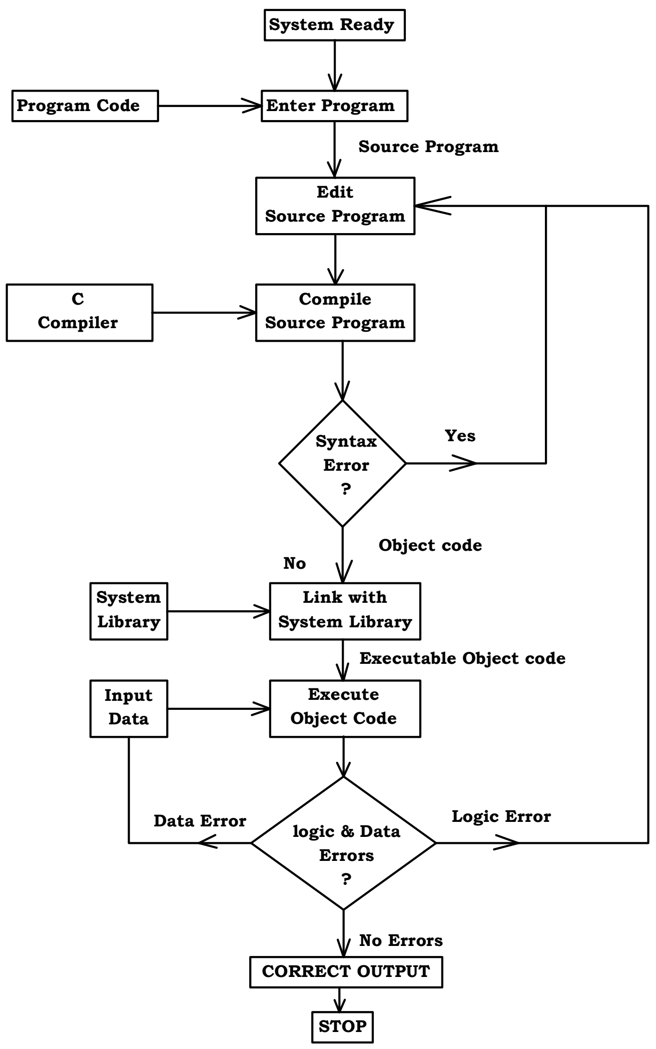
- Creating the program
- Compiling the program
- Linking the program with functions that are needed from the C library.
- Executing the program

Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming files may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channelled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

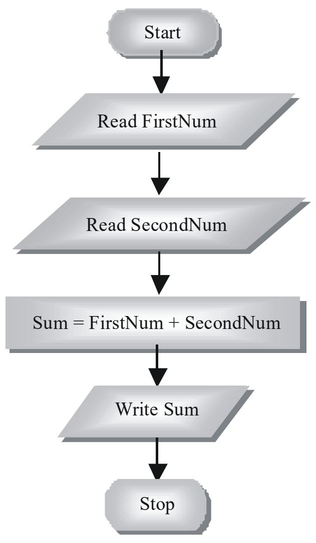
The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers).

NOTES



## Examples for Flowcharts with Algorithms

a. Draw a flowchart for adding two numbers and write an algorithm for it.

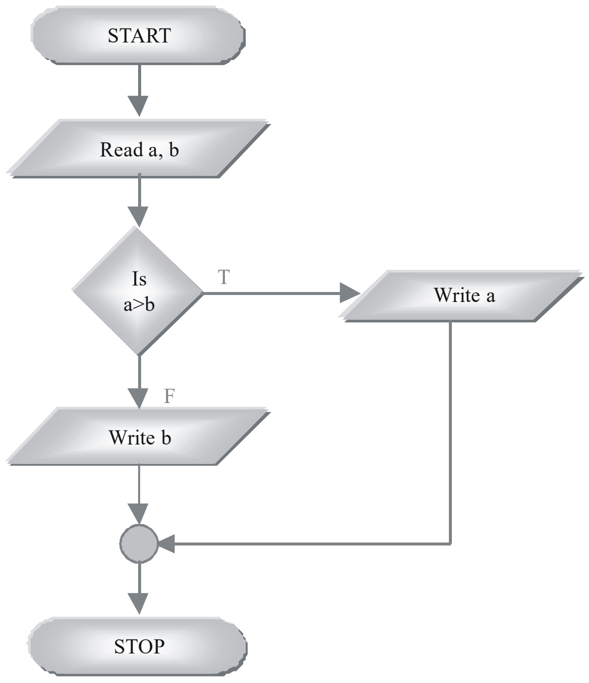


NOTES

**Algorithm for addition of two numbers:**

```
Step 1: Start
Step 2: Read FirstNumber
Step 3: Read SecondNumber
Step 4: Sum= FirstNumber + SecondNumber
Step 5: Write (Sum)
Step 6: Exit
```

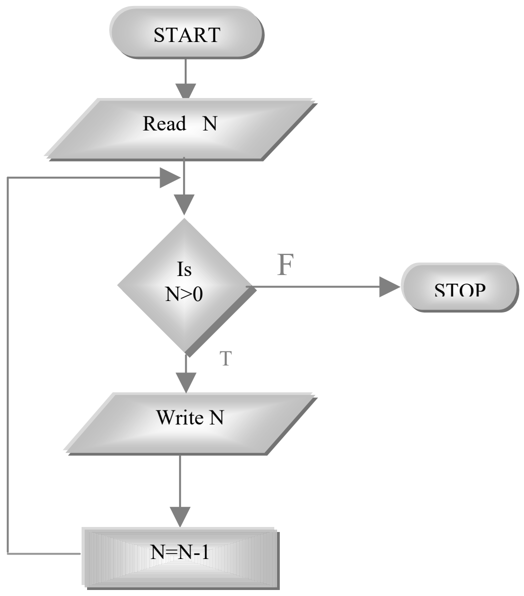
b. Draw a flowchart to find the larger number between two numbers and write an algorithm for it.



**Algorithm for finding large number between two numbers**

```
Step 1: Start
Step 2: Read a and b
Step 3: IF a > b THEN Write (a)
        ELSE Write(b)
Step 5: Stop
```

c. Draw a flowchart to display natural numbers between 1 and N in reverse order.



NOTES

d. Draw a flowchart to display umber of odd digits in a given number.

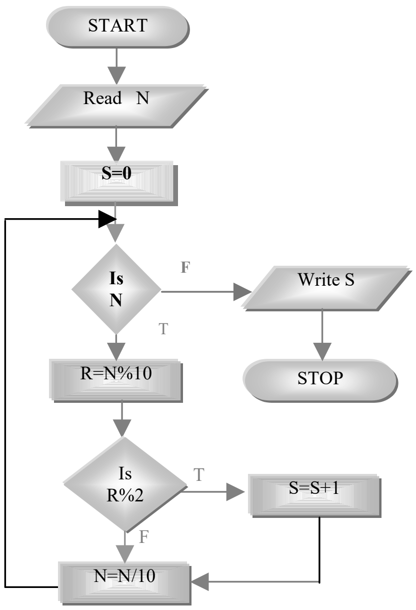
**Algorithm for displaying Natural numbers between 1 and N in Reverse Order.**

```
Step 1: Start
Step 2: Read  N
Step 3: Repeat while N>0
                Write (N)
                 N=N-1
Step 4:Exit
```

**Algorithm to display number of odd digits exist in a given number.**

```
Step 1: Start
Step 2: Read  N
Step 3: S=0
Step 4: REPEAT while N>0
                R=N mod 10
                IF  R mod 2 THEN
                       S=S+1
                N =N/10
Step 5. Write(s)
Step 6: Exit
```

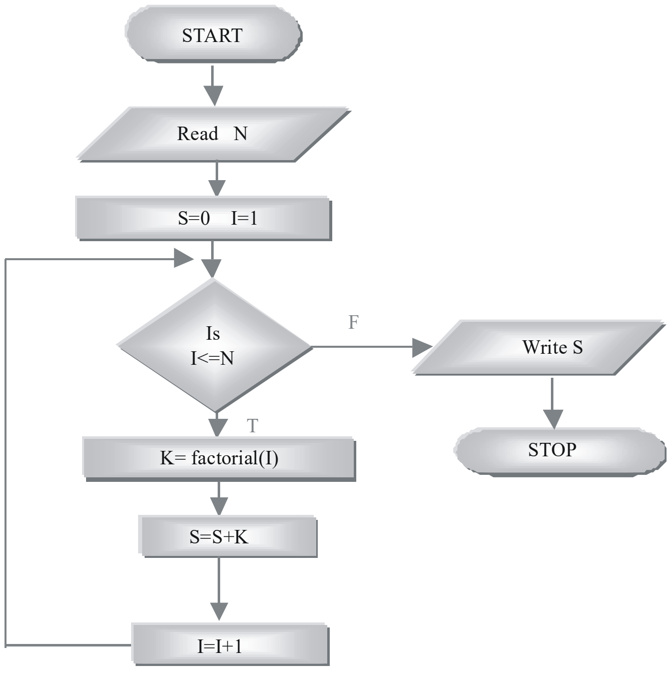
**NOTES**



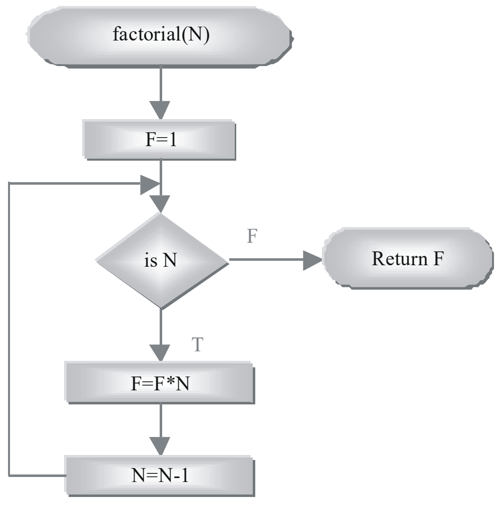
e. Draw a flowchart to evaluate the series 1! + 2!+ 3!+ .......+N!

**Algorithm for evaluating the series 1!+2!+.....+N!**

```
Step 1: Start
Step 2: Read  N
Step 3: S=0,I=1
Step 4: Repeat while I<=N
                K=factorial(I)
                S=S+K
                I=I+1
Step 5. Write(S)
Step 6: Exit
```



f. Flowchart to evaluate N!

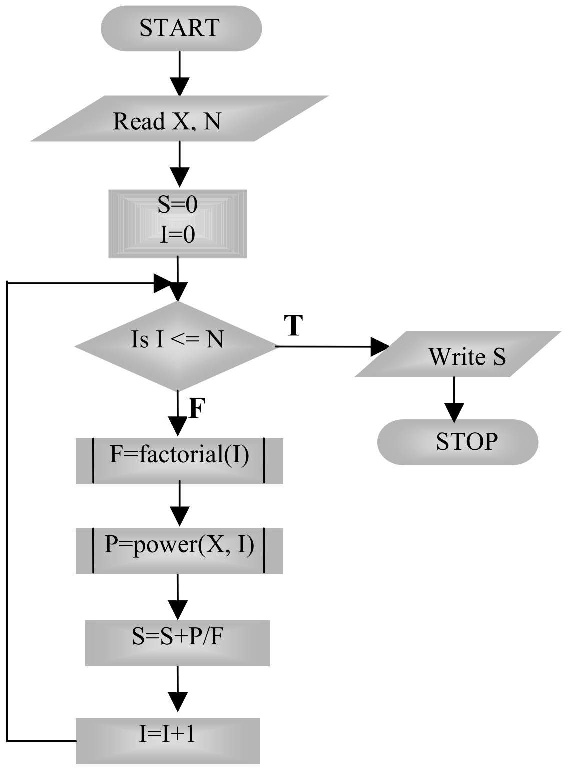


**Algorithm to find factorial(N).**
**Where N is a value and function returns**
**Factorial value for N**

```
Step 1: F=1
Step 2: Repeat while N <> 0
               F=F*N
               N=N-1
Step 3: Return F
```

NOTES

g. Draw a flowchart to evaluate the series $1+x+ x^2/2! + ...+x^n /N!$

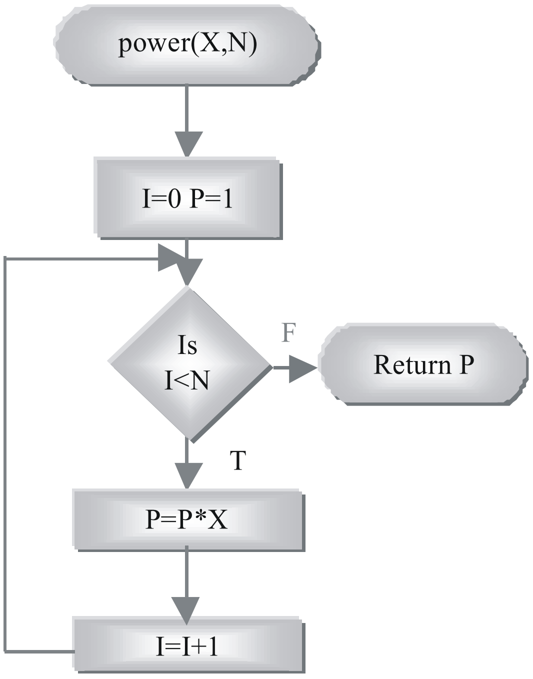


NOTES

**Algorithm to evaluate $1 + X + X^2/2! + \ldots + X^n/N!$**

```
Step 1: Read X,N
Step 2: S=0,I=0
Step 3: Repeat while I<=N
                F=factorial(I)
                P=power(X,I)
                S=S+P/F
                I=I+1
Step 4: Writes(S)
Step 5: Exit
```

h. Flowchart to evaluate Power(X, N)



**Algorithm to evaluate Power(X, N). Where X and N are values**

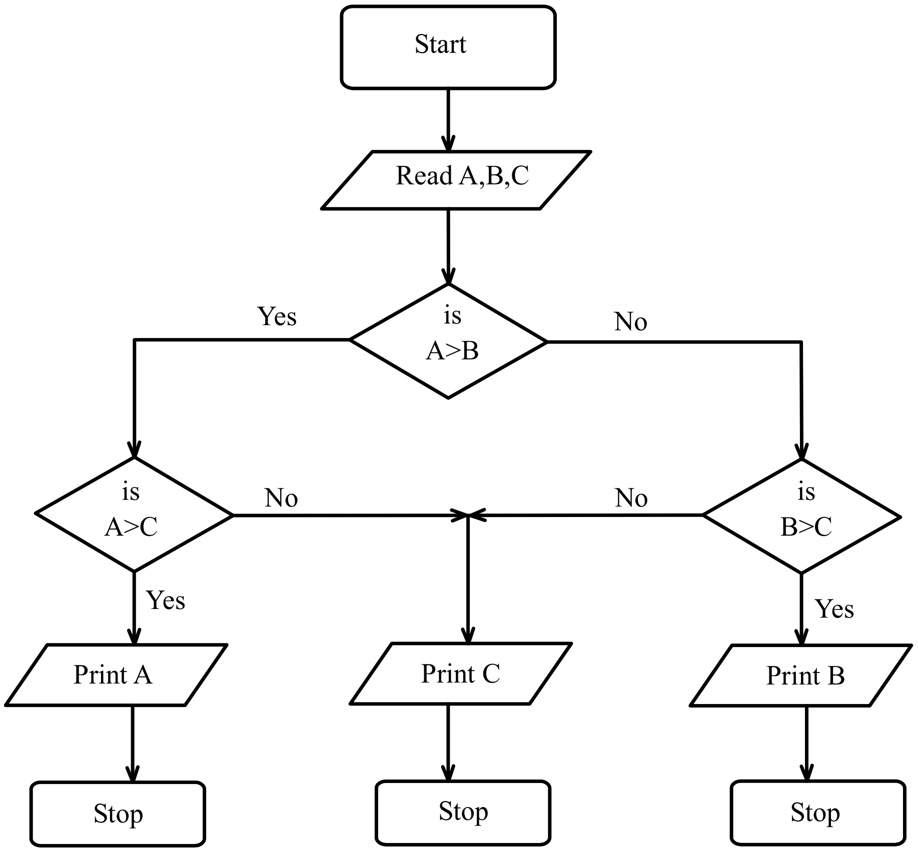
```
Step 1: I=0,P=1
Step 2: Repeat while I<N
                P=P*X
                I=I+1
Step 3: Return P
```

## 1.2.12 Designing an Algorithm using Flowcharts

NOTES

**Example 1.3:** Algorithm to pick the largest of three numbers.

```
Step 1: Read A, B, C.
Step 2: If A > B, go to Step 3.
        Else go to Step 5.
Step 3: If A > C
        Print A as the largest number.
        Else
        Print C as the largest number.
Step 4: Stop.
Step 5: If B > C
        Print B as the largest number.
        Else
        Print C as the largest number.
Step 6: Stop.
```

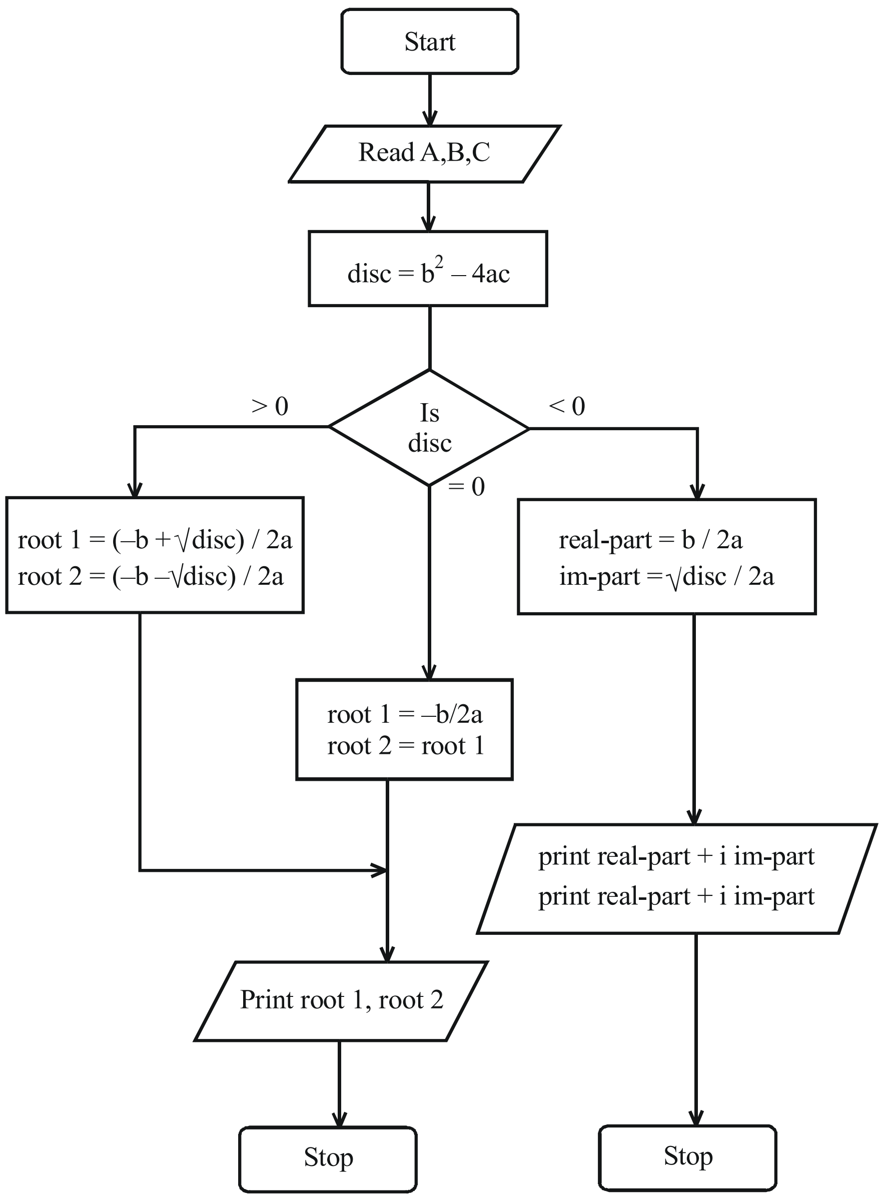


NOTES

**Explanation:** Read the three numbers A, B and C. A is compared with B. If A is larger, then it is compared with C. If A turns out to be the largest number again, then A is the largest number; otherwise, C is the largest number. If in the second step, A is less than or equal to be B, then B is compared with C. If B is larger, then B is the largest number; otherwise, C is the largest number.

**Example 1.4:** Algorithm to find the roots of a quadratic equation $ax^2 + bx + c = 0$ for all cases.

```
Step 1: Read a, b, c.
Step 2: disc ← $b^2$ – 4ac.
Step 3: If disc ← 0, go to Step 4.
            Else, if disc > 0, go to Step 5.
            Else, go to Step 6.
Step 4: root l ← – b/2a.
            root 2 ← rootl.
            go to Step 7.
Step 5: Root l ← (–b + sqrt (disc)) / 2a.
            Root 2 ← (–b – sqrt (disc)) / 2a.
            go to Step 7.
Step 6: real-part ← –b / 2a.
            im-part ← sqrt(–disc) / 2a.
            Print real-part + i im-part.
            Print real-part – i im-part.
            Stop.
Step 7: Print root l, root 2.
            Stop.
```



**NOTES**

**Example 1.5:** Algorithm for finding maximum and minimum numbers.

```
Step 1: Read number.
Step 2: Maximum ← number. Minimum ← number.
Step 3: If (another number) go to Step 4.
            Else go to Step 7.
Step 4: Read number.
Step 5: If number > Maximum Maximum = number.
            Else if number < Minimum
            Minimum = number.
```

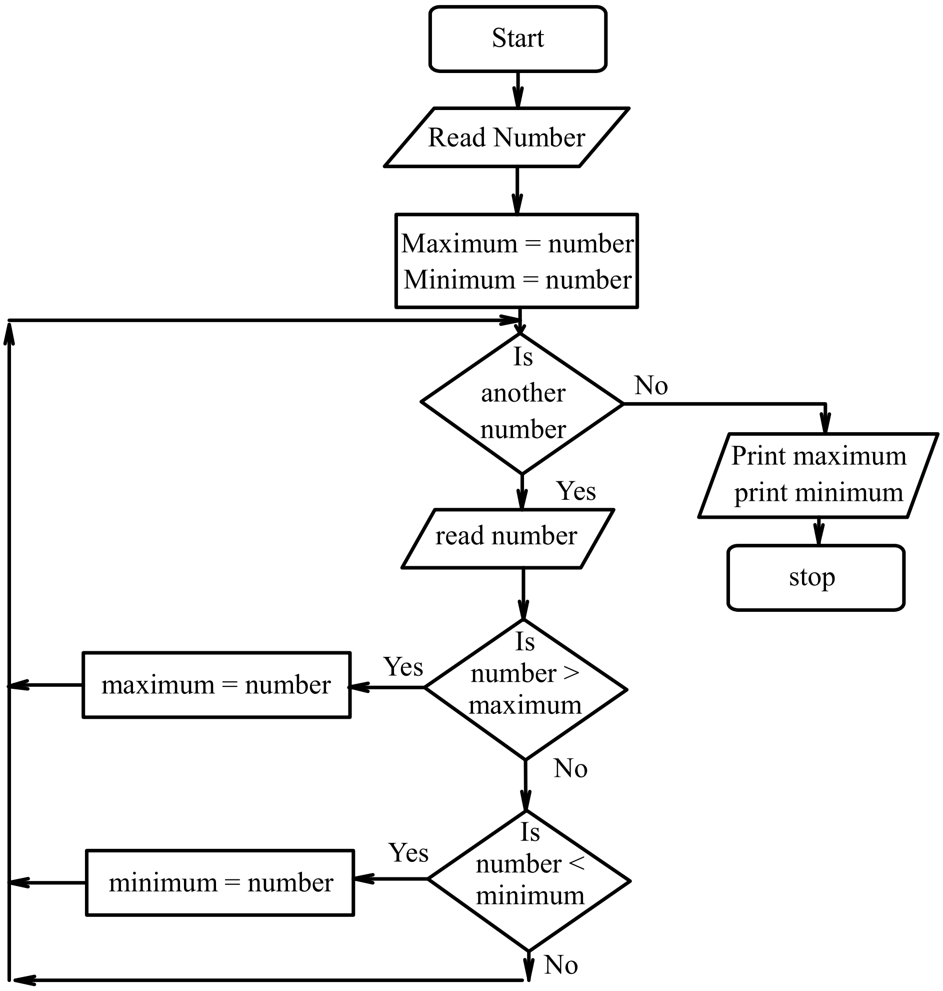
**Step 6:** go to Step 3.

**Step 7:** Print Maximum.

Print Minimum.

NOTES

**Step 8:** Stop.



**Example 1.6:** Algorithm for finding maximum and minimum of given n numbers.

**Step 1:** Read N.

**Step 2:** Counter ← 1.

Read number.

Maximum ← number.

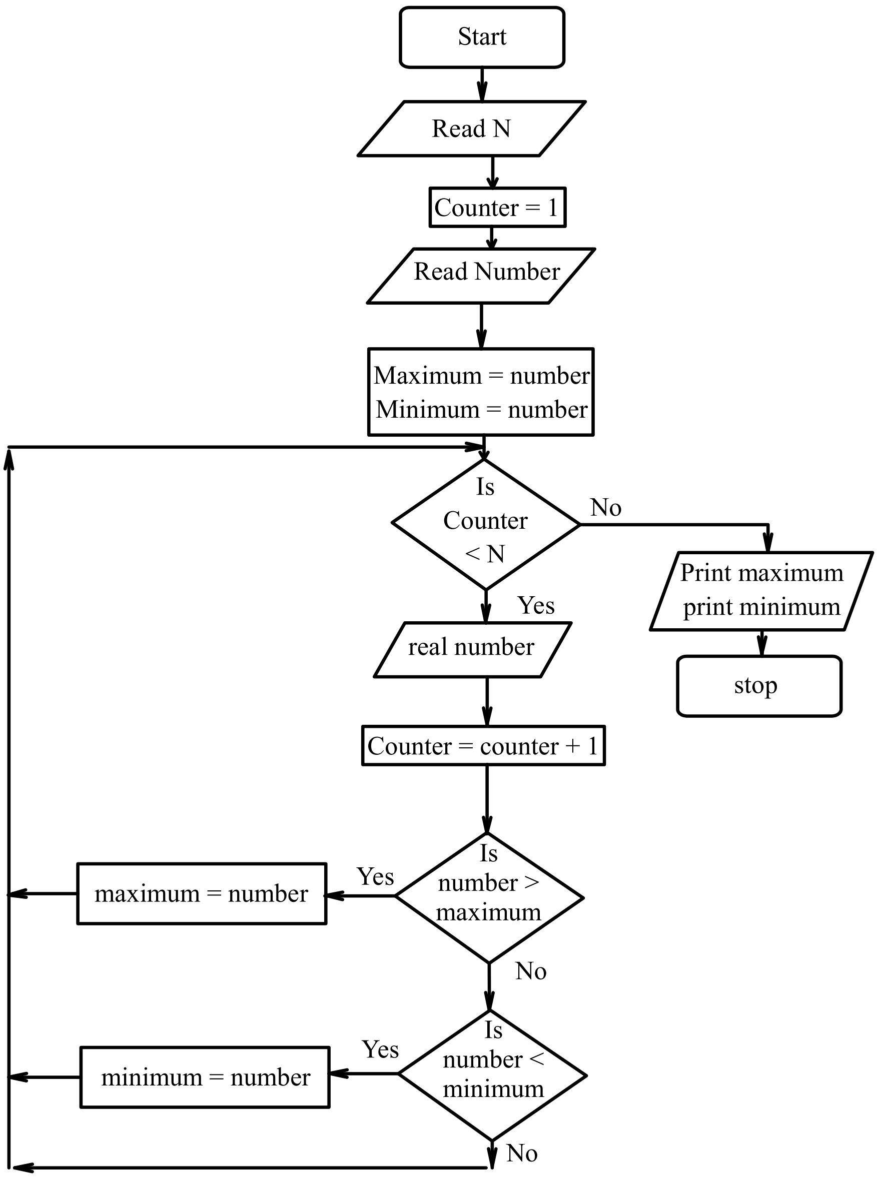
Minimum ← number.

**Step 3:** If Counter < N go to Step 4.

Else go to Step 7.

```
Step 4: Read number.
        Counter ← Counter + 1.
Step 5: If number > Maximum
Step 6: Maximum ← number.
Step 7: Else If number < Minimum
Step 8: Minimum ← number.
Step 9: go to Step 3.
Step 10: Print Maximum. Print Minimum.
Step 11: Stop.
```

NOTES

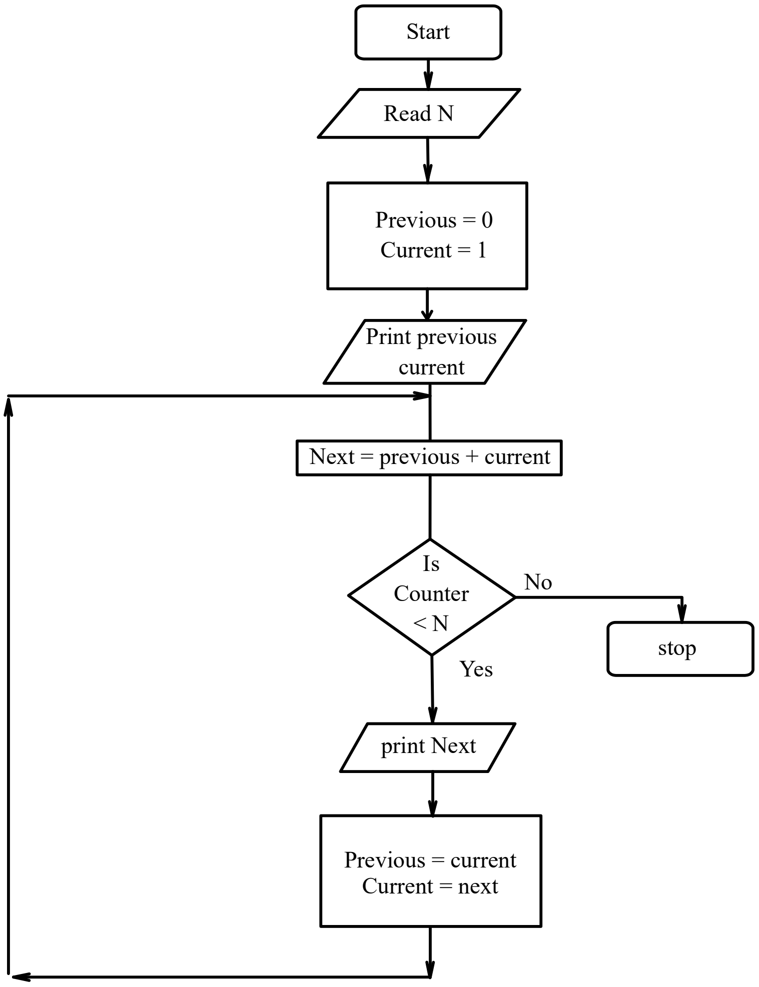


NOTES

**Example 1.7:** Algorithm for generating Fibonacci numbers up to n.

The first and second terms in the Fibonacci series are 0 and 1. The third and subsequent terms in the sequence are found by adding the preceding two terms in the series. The Fibonacci series is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ......

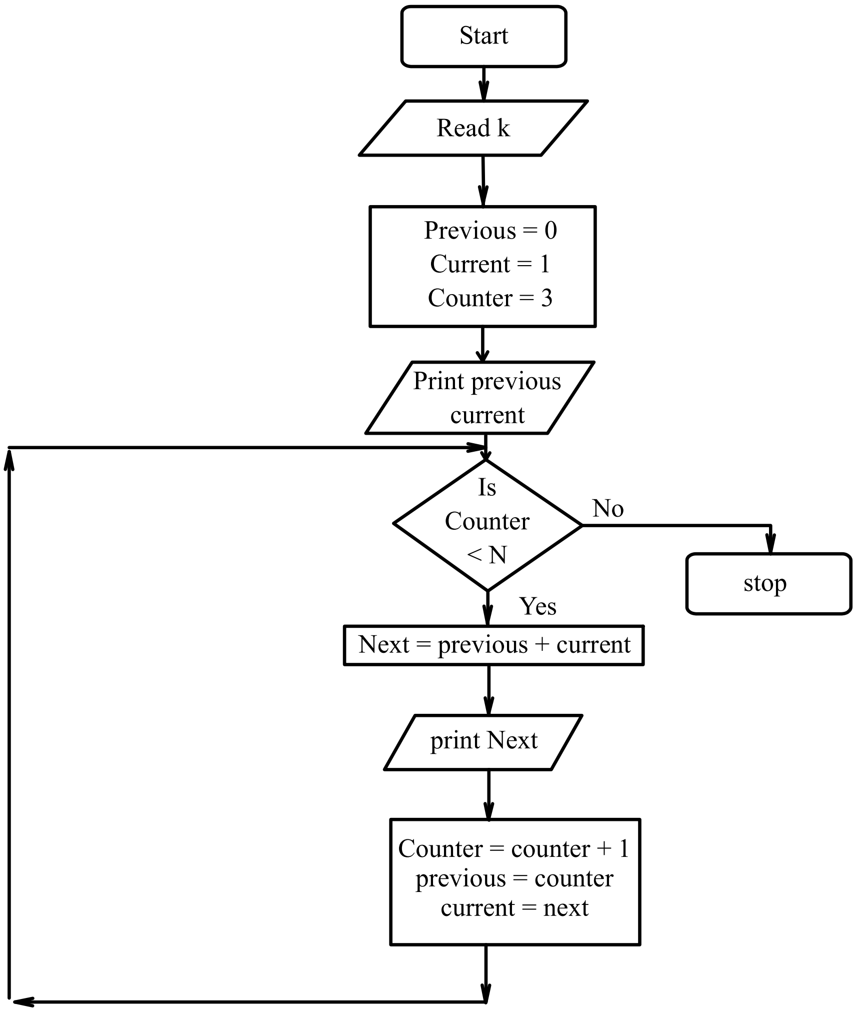
```
Step 1: Read N.
Step 2: Previous ← 0. Current ← 1.
        Print Previous, Current.
Step 3: Next ← Previous + Current.
Step 4: If Next < N
        Print Next.
        Previous ← Current.
        Current ← Next,
        go to Step 3.
        Else go to Step 5.
Step 5: Stop.
```



**Example 1.8:** Algorithm for generating first k Fibonacci numbers.

```
Step 1: Read k.
Step 2: Counter ← 2.
        Previous ← 0.
        Current ← 1.
        Print 'First k Fibonacci numbers are:'
        Print Previous, Current.
Step 3: Next ← Previous + Current.
        Counter ← Counter + 1.
        Print Next. Previous ← Current.
        Current ← Next.
Step 4: If (Counter < k) go to Step 3.
        Else go to Step 5.
Step 5: Stop.
```

NOTES



NOTES

**Example 1.9:** Sum of first n Factorials

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n and is denoted by n! It is defined as follows:

```
N! = n(n-1) … 2*1
```

For example, 5! =5*4*3*2*1. Its older notation was $\underline{n|}$. In factorial number system where the denominations are 1, 2, 6, 24, 120,…, etc. the nth digit is in the range 0 to n. This identity works as shown below in the example:

```
1*1!+2*2!+3*3!+ … +k*k! = (k+1)! - 1
Sum of 2!+3! = (2*1) + (3*2*1) = (2) + (6) = 8
```

The following algorithm is used to find the sum of n factorials:

Algorithm of Sum of n Factorials

```
Step 1: integer n, factorial, i, j, sum;
Step 2: sum←0;
Step 3: print 'Enter the number';
Step 4: read n;
Step 5: for i←1 to n //Running outer loop till n value
{
factorial←1
for j←1 to i
//Inner loop to calculate the sum of n factorial values
{
factorial ←factorial*j;
//Calculating n factorial values
}
sum←sum+factorial;
//Calculating sum of n factorial values
}
Step 6: print 'Sum of n Factorials';
Step 7: print sum;
//Print the sum value of n factorials
```

Implementation to find the Sum of first n Factorials

```
/*——————— START OF PROGRAM ————————*/
#include <stdio.h> //Declaration of Header files
#include <conio.h>
/*——-1/1! + 2/2! + 3/3! + 4/4! ...-——*/
void main()
{
int factorial,sum=0,i,j,n;
//Declaring and assigning the variables
printf("Enter a value for [n] value = ");
scanf("%d", &n);
//Accept input value for n term
for(i=1;i<=n;i++) //For outer loop till n value
{
factorial=1;

for(j=1;j<=i;j++)
//Using inner for loop to calculate the sum of n factorial
{
factorial = factorial *j;
}
sum=sum+ factorial;
}
Printf("\n sum of %d factorial = %d", n, sum);
}
getch();
}
```

The result of the above program is as follows:

```
Enter a value for [n] value = 4
sum of 4 factorial = 33
```

How the above program works it is explained step-wise-step in the following ways:

```
1!+2!+3!+4! = 33
The value of 1! = 1
The value of 2! = 2
The value of 3! = 6
The value of 4! = 24
1+2+6+24= 33
```

**Example 1.10:** To Find Largest Value and Second Largest Value of the List

The largest and second largest values in the given list are determined by array implementation. Array can contain the various elements of the list. The algorithm to find the largest and second largest of given list is as follows:

Algorithm to find the largest value and second largest value of the given list

```
Step 1: integer M, a[M], i, largest, t, second_largest;
Step 2: print 'Enter a value for array';
Step 3: read M;
Step 4: for i←1 to M
print 'Enter values:';
```

```
read a[i];
Step 5: if i==1
largest←t←second_largest←a[i];
Step 6: else if a[i]>largest
second_largest←largest;
largest←a[i];
Step 7: else if a[i]>second_largest && a[i]<largest
second_largest←a[i];
Step 8: else if a[i]<t
t←a[i];
Step 9: print 'Largest value in the given list =';
Step 10: print largest;
Step 11: 'Second largest value in the given list =';
Step 12: print second_largest;
```

The result of the above algorithm is as follows:

```
Enter a value for array
5
```

Then `array A[M]` is assigned a value 5 as `A[5]`.

The input values are entered in the following way:

```
Enter values
45
90
112
4
35
Largest value in the given list = 112
Second largest value in the given list = 90
```

The first element of the array is `45` which is assumed to be the largest value and it is kept in the temporary location where it is temporarily stored in variable `t`. All the remaining values are checked from this number. Now, the `A[i]` value is assigned as `45`. At second step, the condition is satisfied so largest value is `90`. Now, `90` is checked with the next entered value `112`. Because the condition is not satisfied so `112` is assumed as greater value. The values `4` and `35` are less than `90`, so the condition for less than largest is not satisfied. The checking process of second largest value `'45'` is done after checking the rest four values and declaring `112` as first largest value. Further, the statement `'second_largest=largest'` is used. The first element of the array is again taken as largest among the four values. Now, `45` is checked step-by-step in `if else if` conditional statement to find the second largest value.

Program to find the largest value and second largest value in a given list

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
#define M 5 //Define preprocessor directive that assigns
M = 5
void main()
{
int a[M], i, largest, t, second_largest;
clrscr(); //Clear the screen of previous
for(i=1; i<=M); i+)
{
printf("Enter %d value");
scanf("%d", &a[i]);
if(i==1)
largest=t=second_largest=a[i];
//The largest, t, second_largest values are assigned as
the value of a[i].
if(a[i]>largest)
{
second_largest=largest;
largest=a[i];
}
if(a[i]>second_largest && a[i]<largest)
second_largest=a[i];
if(a[i])<t)
t=a[i];
}
printf("\nLargest Vvalue in the given list = %d", largest);
printf("\nSecond largest value in the given list = %d",
second_largest);
getch();
}
```

NOTES

The result of the above program is as follows:

```
Enter 1 value = 45
Enter 2 value = 90
Enter 3 value = 112
Enter 4 value = 4
Enter 5 value = 35
Largest value in the given list =112
Second largest value in the given list=90
```

In the above program, `#define M 5` statement defines preprocessor directive that works as a macro. It means wherever `M` comes in the program, its value `5` is changed automatically. The `#define` statement can not be terminated by a semicolon (;) because the preprocessor is a program that comes before `main()` statement.

**Example 1.11:** Determining nth root of a number.

The nth root of a number a is a number where n is positive integer. The nth roots are taken with the following iteration where a is the input values and n is the root value to be taken. The equation is arranged in the following ways:

$$b = \sqrt[n]{a}$$

where a is the number, `n` is the nth root and b is the value that retains the nth root of number a. For example, nth root is equal to 3 and number `a` is equal to 2 can be written as because $2^3 = 8$. The following algorithm is used to find out the nth root of a given value:

Algorithm to find the nth root of a number

```
Step 1:   double calculate_root(double,double);
//Declare a calculate_root function having two parameters
Step 2:   double Find_nth_Root(double,double,double);
//Declare a calculate_root function having three parameters
Step 3: double number(double,double);
//Declare a number function having two parameters
Step 4: double x←1, NUMBER_OF_ITERATIONS←40, n;
//Assign value 1 to x variable and NUMBER_OF_ITERATIONS=40
Step 5: double N;
//Declare a variable N as double data type
Step 6: double root;
//Declare a variable root as double data type
Step 7: x_label:
//Assign a label named as x_label
Step 8: print 'Enter root do want [2,3, …5] ?'
Step 9: read n;
//Accept input value n
Step 10: if n<=0
//Check the condition where n is less than 0
Step 11: print 'Number should be Greater than 0';
Step 12: print n;
Step 13: goto x_label;
//Go to label on x_label
Step 14: y_label:
//Assign a label named as y_label
```

NOTES

```
Step 15: print 'Enter the number for Root';
Step 16: read N;
Step 17: if N<=0
Step 18: print 'Number should be greater than 0';
Step 19: print 'PRESS ANY KEY TO ENTER AGAIN';
Step 20: goto y_label;
//Go to label on y_label
Step 21: x←calulate_root(n,N);
//x retains the returned value of function calculate_root
Step 22: print 'The first assumed root is',x;
Step 23: root←Find_nth_Root(N,n,x);
//root retains the Find_nth_Root returned value
Step 24: print 'Root of n',n;
Step 25: print N;
Step 26: print root;
Step 27: double calculate_root(double n,double N)
Step 28: integer i,xr;
//integer i and xr are declared
Step 29: xr←1;
//xr is assigned as 1
Step 30: double j←1;
//double j is assigned as 1
Step 31: while(1)
Step 32: for i←0 to n //Running for loop
     {
     xr←xr*j; //xr retains the value of xr*j
     }
Step 33: if xr>N
    Return j-1; //Returns j-1
Step 34: j←j+1; //j value is increased by 1
Step 35: xr←1; //xr value is increased by 1
Step 36: double Find_nth_Root(double NUM,double n,double
X0) //Function Find_nth_Root starts from here.
Step 37: int i;
Step 38: double d←1.0;
Step 39: double first_term, second_term, root←X0;
Step 40: for i←1 to NUMBER_OF_ITERATIONS
//Body of for loop starts that calculates first term and
second term value of enter values of NUMBER_OF_ITERATIONS
Step 41: d←number(root,n);
//d retains the n th value of given number.
```

```
Step 42: first_term←((n-1)/n)*root;
// first_term retains the value of let say 5 (5-1)/5)*
root value
Step 43: second_term←(1/n)*(NUM/d);
Step 44: root←first_term+second_term;
Step 45: print first_term,second_term,root;
Step 46: return root;
Step 47: double number (double x,double n)
Step 48: double d←1;
Step 49: integer i;
Step 50: for i←1 to n-1
Step 51:  d←d*x; //Printing the final nth root value of
given number n
Step 52: return d;
//Returns the resulted value to d
```

The above algorithm can work in the following way:

The odd nth root let say cube root of a real number b can not be identified with the fractional power a^{1/n}, although so has been done in the entries nth root and cube root. The fractional power with a negative base is not uniquely determined therefore, it depends not only on the value of the exponent but also on the form of the exponent; e.g.,

(–1)^{1/3} = the 3rd root of –1, i.e. = –1

(–1)^(2/6) = the 6th root of (–1)^2, i.e. = 1

Implementation to find the nth root of a number

```
/*———————— START OF PROGRAM ————————*/

#include<stdio.h>
#include<conio.h>
#define NUMBER_OF_ITERATIONS 40
//Preprocessor directive where NUMBER_OF_ITERATIONS is
defined as macro
double calculate_root(double,double);
double Find_nth_Root(double,double,double);
double number(double,double);
void main()
{
     double x=1,n;
     double N;
     double root;
x_label:
```

```
printf("\n Enter root value [2,3, …5] ?");
     scanf("%f",&n);
     if(n<=0)
     {
     printf("\nNumber should be Greater than 0");
printf("Press any key to enter again");
     getch();
     goto x_label;
     }
y_label:
printf("\n\rEnter a number = ");
scanf("%f",&N);
     if(N<=0)
     {
     printf("\nNumber should be greater than 0");
printf("\n PRESS ANY KEY TO ENTER AGAIN …");
     getch();
goto y_label;
     }
x = calulate_root(n,N);
printf("\n\nThe  first  assumed  root  is  calculated  as
%f\n",x);
     root=Find_nth_Root(N,n,x);
printf("\n\n%f Root of %f = ",n,N);
     printf("Root value is = %f",root);
     getch();
    }
double calculate_root(double n,double N)
    {
     int i, xr=1;
     double j=1;
     while(1)
     {
     for(i=0;i<n;i=i+1)
     {
     xr=xr*j;
     }
     if(xr>N)
     {
     return(j-1);
     break;
     }
     j=j+1;
     xr=1;
     }
     }
double Find_nth_Root(double NUM,double n,double X0)
{
     int i;
     double d=1.0;
     double first_term,second_term,root=X0;
for(i=1;i<=NUMBER_OF_ITERATIONS;i++)
{
d=number(root,n);
first_term=((n-1)/n)*root;
second_term=(1/n)*(NUM/d);
root=first_term+second_term;
printf("\n%f\t%f\t%f",first_term,second_term,root);
}
     return(root);
}
double number(double x,double n)
{
    double d=1;
    int i;
    for(i=1;i<=n-1;i++)
    d=d*x;
    return(d);
}
```

The result of the above program is as follows:

```
Enter root value [2,3, …5] ? 3
Enter a number = 64
Root value is = 4
```

In the above program, the syntax of `#define` is as follows:

```
#define macro-name replacement-string
```

The `#define` command is used to make substitutions throughout the program file in which it is located. It causes the compiler to go through the file, replacing every occurrence of macro-name with replacement-string. The replacement-string stops at the end of the line. The above program calculates the nth root of any number a. This program uses the NEWTON_RAPTION_ ITERATION method for calculation. For Example, you have to calculate the square root of 16, then n=2 (square root), a=16 (the number). The following examples show how nth root of the given number can be written:

Enter a Number = 32, Enter a Root = 5. The (n th) 5th root of 32 is 2.

Enter a Number = 11, Enter a Root = 4. The 4th root of 11 is 1.82116.

NOTES

**Example 1.12:** Greatest Common Divisor (GCD).

The GCD of two integers is the largest integer value that divides both integer values where both the values are not zero. The basic identities of GCD are as follows:

```
GCD(A,B)=GCD(B,A)
GCD(A,B)=GCD(-A,B)
GCD(A,0)=ABS(A)
```

Both the integer values can be assumed as nonnegative integers. The GCD procedure extracts the greatest common divisor A because the common divisor B divides to get the remainder until finally B divides A. The result A is in fact a greatest common divisor because it contains every other common divisor B.

*GCD Algorithm*

```
Step 1: integer m, n, q, r; //Variables are defined
Step 2: print'Enter two values:';
Step 3: read m,n;
//Input two values for m and n variables
Step 4: if m==0 OR n==0
//Checking the condition whether m is equal to 0 or n is
equal to 0
print'One number is Zero';
else
reach: //Label reach is defined for loop
q←m/n;
//Get the value of q after ding m by n
r←m – q*n; //Gets remainder value
Step 5: if r==0
print 'GCD Value is :'; //Prints message
print n; //Prints GCD value
goto end; //Got to end label
else
m←n←r;
//Assigning m is equal to n that is also equal to r
goto reach; //Go to reach label
end:; //Label end is defined
```

If the two given values are 10, 12 then the greatest common factor is the number that divides both the values 10 and 12.

NOTES

The GCD of two given integers (a and b) is the largest positive integer which divides both integers a and b, for example, gcd (10,12)=2. The following table shows the step-by-step procedure to get resultant GCD value:

Let the two values are, m =15 and n = 18.

| div | quo | %Quo | %div | Resultant value |
|---|---|---|---|---|
| 0 | | | | 1 |
| 1 | 15 | False | True | 1 |
| 2 | 7 | False | False | 1 |
| 3 | 5 | False | True | 3 |
| 4 | 3 | | | |

The loop exits and returns 3. So, the resultant GCD value of the two given values 15 and 18 is 3.

Program to find GCD of given values:

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
void main()
{
int m, n, q, r;
clrscr();
printf("Enter two values:");
scanf("%d%d", &m,&n);
if (m==0||n==0)
printf("One number is Zero");
else
reach:
{
q=m/n;
r=m - q*n;
}
if(r==0)
{
printf("GCD Value is : %d", n);
goto end; //Go to end label
}
else
{
m=n=r;
goto reach;
}
end:;
}
```

The GCD can also be calculated applying Euclidean algorithm. If the integers a and b are two positive integers and n is the remainder, then (a, b) = (b, r).

```
Euclidean_gcd(a,b)
Step 1: integer x, y, f, d;
Step 2: x←f; y←d;
Step 3: if y=0 return x
Step 4: r←x mod y;
Step 5: x←y;
Step 6: y←r;
Step 7: goto Step 2.
```

The above algorithm works in the following way:

Small value (x) = 10, Large value (y) =12.

| Large | Small | Remainder |
|---|---|---|
| 12 | 10 | 2 |
| 10 | 2 | 0 |

Result: **2** is the GCD of 10, 12.

The above algorithm is known as Euclid's GCD algorithm that extracts the greatest common divisor x. The common divisor y divides x and keeps remainder as value n. This process is continued until y divides x finally. Therefore, value assigned for x is the greatest common divisor if it contains every other common divisor y.

**Example 1.13:** Base Conversion (Decimal to Binary).

The base of a binary number is 2 and of decimal number is 10 (denary). Binary numbers have only two numerals (0 and 1), whereas decimal numbers have 10 numerals (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). An example of a binary number is 10011100 and decimal number is 0.012345679012. The decimal numeral system is the one that is the most widely used. Computer operations are performed with number base conversion.

The following algorithm is an example of printing an integer value into binary format:

*Algorithm*

```
Step 1: integer number, binary_val,temp_val,counter,d_val;
Step 2: binary_val←0; //Assigning value 0 to binary_val
Step 3: temp_val←number; // Assigning temp_val is equal
to number
Step 4: counter←0; //Assigning value 0 to counter
Step 5: print 'Enter the number';
Step 6: read number; //Accept input values to number
Step 7: if temp_val>0
{
```

NOTES

```
d_val ← mod(temp_val,2)
binary_val ← binary_val + d_val*10^counter;
//10^counter means power(10,counter)
d_val← d_val + a_val*p_val;
temp_val ← int(temp_val/2)
//Change the fraction values as integer data types.
counter ← counter + 1;
//Increase the counter value by one
}
Step 7: print 'Binary Value';
Step 8: print binary_val; // Prints resultant binary
value
```

How the above algorithm works is explained below:

Let us take a decimal value 6.

```
d_val = 6 mod 2 that returns 0
binary value=0+3*10^0 returns 0
d_val = 3 mod 2 returns 1
counter = 0 +1= 1
```

The decimal number **6** is equal to binary number **110**. This conversion is explained in the following way:

| number | number/2 | number % 2 |
|---|---|---|
| 6 | 3 | 0 |
| 3 | 1 | 1 |
| 1 | 0 | 1 |

Implementation of base conversion (decimal to binary)

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h> //Declaring Header files
#include <conio.h>
#include <math.h>
void main() //Start main() function
{
int number, binary_val, temp_val, counter, d_val, p_val;
binary_val=0;
// Declaring integer data types variables
temp_val=number; //Assigning temp_val is equal to number
counter=0; //Initailizing 0 to counter
printf("\n Enter a number");
scanf("%d", &number); //Accept input value
if (temp_val>0)
{
```

```
d_val = temp_val%2;
//Returns remainder to d_val
p_val=power(10,counter);
binary_val= binary_val+ d_val*p_val; //The value of
binary_val is added to d_val by d_val by 10 'raise to the
power' counter value
temp_val = int(temp_val/2)
//if temp_val contains fraction value, int() function
changes the integer type value
counter = counter +1;
//Counter variable is increased
}
printf("Binary Value = %d", binary_val); //Printing the
binary value
getch();
}
```

**Base Conversion (binary to decimal)**

***Algorithm***

```
Step 1: integer number,d_val,temp_val,counter,a_val;
Step 2: d_val←0; //Assigning 0 to d_val
Step 3: temp_val←number; //Assigning temp_val is equal
to number
Step 4: counter←0; //Assigning 0 to counter
Step 5: print 'Enter the number';
Step 6: read number; //Accept input value for number
Step 7: if temp_val>0 //Body of if control statement
{
a_val ← mod(temp_val,10);
p_val←power(2,counter);
d_val← d_val+ a_val*p_val;
temp_val ← int(temp_val/10);
counter ← counter +1;
}
Step 8: print 'Decimal value';
Step 9: print d_val;
```

How the above algorithm works is explained as follows:

Let us take binary number 1011.

$=1*2^3+0+2^2+1*2^1+1*2^0$

$=8+0+2+1 = 11$

The binary number **1011** is equal to decimal number **11**.

```
/*—————————— START OF PROGRAM ——————————*/
#include <stdio.h> //Declaring Header files
#include <conio.h>
#include <math.h>
void main() //Start main() function
{
int number,d_val,temp_val,counter,a_val;
// Declaring integer data types variables
temp_val=number;
//Assigning temp_val is equal to number
d_val=0;
counter=0; //Initailizing 0 to counter
printf("\n Enter a number");
scanf("%d", &number); //Accept input value
if (temp_val>0)
{
a_val = temp_val%10;
//Returns remainder to d_val
p_val = pow(2,counter);
//Returns counter value raise to the power 2 to p_val
variable
d_val=d_val+ a_val*p_val;
//The value of d_val is added to multiplied value of
a_val and p_val
temp_val = int(temp_val/10)
//if temp_val contains fraction value, int() function
changes the integer type value
counter = counter +1;
//Counter variable is increased
}
printf("Decimal Value = %d", d_val); //Printing the binary
value
getch();
//Pressing key to return the program
}
```

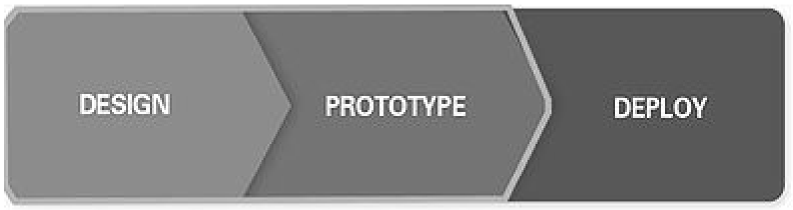
The above program is able to convert the binary number into decimal number. The result of the program is as follows:

```
Enter a number = 1011
Decimal Value = 11
```

When a theoretical algorithm design is combined with the real-world data, it is called **algorithm engineering**. When you take an algorithm and combine it with a hardware device that is connected to the real-world, you can verify and validate the algorithm results and behaviour more precisely and accurately. A simple data acquisition or stimulus device may be considered as the real-world device. Alternatively, you can implement an algorithm on some embedded platform, such as a **field-programmable gate array** (**FPGA)** or microprocessor which can be similar to the final system design.

The first specific use of the term, 'algorithm engineering' was at the inaugural Workshop on Algorithm Engineering (WAE) in 1997.

It has of late been used for describing the steps in a graphical system design: 'A modern approach to design, prototype and deploy the embedded systems which combine open graphical programming with the **commercial off-the-shelf** (**COTS**) hardware for dramatically simplifying development, bringing higher-quality designs with a migration to custom design'.



With the help of algorithm engineering, you can transform a pencil-and-paper algorithm into a robust, efficient, well-tested and easily usable implementation. It covers various topics, from modelling cache behaviour to the principles of good software engineering. However, experimentation is its main focus.

**Check Your Progress**

4. List the instructions that are dealt with in RAM model.
5. What are primitive operations?
6. What is a flowchart?
7. Define algorithm engineering.

## 1.3 EXPONENTIATION

The mathematical operation of the form $x^n$ is known as exponentiation. This involves two numbers, base and exponent. Here, in $x^n$, $x$ is the base and $n$ is the *exponent*. For positive integral values of $n$ exponentiation means repeated multiplication as shown below:

$$x^n = x \times x \times \ldots\ldots.x \times x \qquad (n \text{ times})$$

This can be compared with mathematical operation of multiplying with a positive integer that means repeated addition:

$$xn = x + x + \ldots\ldots.x + x \qquad (n \text{ times})$$

Exponentiation is written as a superscript towards the right of the base. The exponentiation $x^n$ is equivalent to saying '*x raised to the nth power or x raised to the power n or x raised to the power n'*. Some use statements more brief that these and say '*x to the n*.'

NOTES

**Negative Exponentiation**

The exponentiation $x^n$, is also defined when $n$ is a negative integer and $x^1$ 0. There is no natural extension for all real valued $x$ and $n$, but for all positive real values of the base $x$, $x^n$ is defined for real and even complex exponents $n$ by the exponential function $e^y$. Trigonometric functions are also expressed as a combination of complex exponentiation.

Exponentiation is used in many fields such as physics, chemistry, biology, computer science and economics. Applications such as wave behaviour, chemical reaction, kinetics, population growth, public key cryptography and compound interest are used in these fields.

**Exponents One and Zero**

Exponentiation is recursive in nature and one and zero are base cases. $5^1$ mean 5 only and $5^5 = 5 \cdot 5^4$; $5^4 = 5 \cdot 5^3$ and continuing like this, we get $5^1 = 5 \cdot 5^0$.

Another way of saying this is that when $n$, $m$ and $n - m$ are positive (and if $x$ is not equal to zero), one can see by counting the number of occurrences of $x$ that,

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

Extended to the case that $n$ and $m$ are equal, the equation would read,

$$1 = \frac{x^n}{x^n} = x^{n-n} = x^0$$

For equal numerator and denominator this gives the definition of $x^0$. Thus, we **define** $5^0 = 1$ for the equality to hold. This leads to two basic rules of exponentiation:

(i) Any number to the power 1 is the same number.

(ii) Any nonzero number to the power 0 is 1.

**Negative Integer Exponents**

By definition, when base is a nonzero number and exponent is 1 is used give negative, reciprocal of that base:

$x^{-1} = 1/x$ and one may write $x^{-n} = 1/x^n$ for $x^1$ 0 and $n \in I^+$.

Negative integral exponent to a base means repeated division of 1 by the base. For example,

$5^{-1} = 1/5$ and $5^{-3} = ((1/5)/5)/5$

**Identities and Properties**

The most important identity satisfied by integer exponentiation is: $x^{a+b} = x^a.x^b$.

This also leads to $x^{a-b} = x^a/x^b$ for $x0$, and

$$(x^m)^n = x^{mn}$$

There is another basic identity:

$$(x.y)^n = x^n.\ y^n$$

Exponentiation is not commutative, since $2^5 = 32$, but $5^2 = 25$.

Not associative since $2^3$ is base and 4 is exponent. It is $8^4$ or 4096, but when base is 2 and exponent is $3^4$ then it becomes $2^{81}$.

NOTES

## 1.3.1 How to Compute Exponentiation Fast?

Different method can be used to compute fast exponentiation. The major ones include the following:

**(i) Squaring Algorithm**

For fast computations, 'exponentiation by squaring' algorithm is used. This algorithm is good for fast computation of large exponent to a number. Due to nature of its working, it is known as square-and-multiply algorithm. Binary exponentiation is another name given to this algorithm. Double-and-add, is also a name given to this algorithm. It makes use of binary expansion of the exponent and is used in modular arithmetic.

The following recursive algorithm computes $x^n$ for a non-negative integer $n$. Here, $x^n$ is written as Power $(x, n)$ and defined as below.

$$\text{Power}(x, n) = \begin{cases} 1, & \text{if } n = 0 \\ x \times \text{Power}(x, n-1,) & \text{if } n \text{ is odd} \\ \text{Power}(x, n/2)^2, & \text{if } n \text{ is even} \end{cases}$$

Here, normal strategy of $x^n = x.x^{n-1}$ is not adopted rather, '$n$ is even' fact is optimized, and according to this fact:

$$x^n = x^{n/2} \times x^{n/2}$$

Using approach as this, $\log_2 n$ squaring and at maximum of $\log_2 n$ multiplications are performed which is more efficient computationally in comparison to that of multiplying the base with itself in a recursive manner.

Given any $(x,\ n) \in \text{R} \times \text{Z},\ x^n$ ,$x^n$ is calculated by:

```
1. if n < 0 then $x := \frac{1}{x}$ and $n := -n$
2. i: = n, y: = 1, z: = x
3. if i is odd then y: = y.z
```

```
    4. $z := z * z$
    5. $i := \left\lfloor \frac{i}{2} \right\rfloor$, this discards the remainder after
performing division
    6. if $i \neq o$, then go to step 3
    7. give $y$ as result
```

NOTES

There is one problem in this algorithm that this gives $0^0 = 1$ which is mathematically indeterminate.

**(ii) Montgomery's Ladder Technique**

Disadvantages of squaring algorithm lies in doing analysis of the operations performed at every step. This algorithm may become problematic if exponent serves the purpose of a secret key. A variant has been created from squaring algorithm by making use of a technique known as Montgomery's Ladder to solve this problem.

Given an integer $n=(n_{r-1}...n_0)_2$ in base 2 with $n_{r-1}=1$ we can compute $x^n$ as follows:

```
$x_1=x$; $x_2=x^2$
for i=r-2 to 0 do
  if $n_i=0$ then
    $x_2=x_1*x_2$; $x_1=x_1^2$
  else
    $x_1=x_1*x_2$; $x_2=x_2^2$
return $x_1$
```

**(iii) 2*K*-ary Method**

In this algorithm calculation is performed for the value of $x^n$ by way of expansion of the exponent in base $2^k$. This method was proposed for the first time in 1939 by Brauer. In this algorithm the functions used are $f(0) = (k,0)$ and $f(m) = (s,u)$, where $m= 2^s*u$ where $u$ is odd.

**Algorithm:**

**Input**

- An element $x \in G$ and $k > 0$, where $k$ is a parameter
- A non-negative integer $n=(n_{r-1},n_{r-2}, n_0)_{2^k}$
- The pre-computed values $x^3,x^5,...x^{2k-1}$.

**Output**

Element $x^n \in G$

**Steps**

```
1. $y$=1 and $i$=$r$ -1
2. while $i$>0 do
3. ($s$,$u$)=$f(n_i)$
4. for $j$=1 to $k$-$s$ do $y$=$y^2$
5. $y$=$y$*$x^u$
6. for $j$=1 to $s$ do $y$=$y^2$
7. $i$=$i$-1
8. return $y$
```

***Note:*** Optimal efficiency is achieved for small integral value of $k$ satisfying $\log(n)<(k(k+1)*2^{2*k})/(2^{k+1}-k-2)+1$

### (iv) Sliding Window Method

This is a variant of 2$k$-ary method which is more efficient. For example, to calculate exponent 398 having binary equivalent as $(110\ 001\ 110)_2$, a window of length 3 is chosen and then 2$k$-ary algorithm is used for computing $1,x^3,x^6,x^{12},x^{24},x^{48},x^{49},x^{98},x^{196}$, $x^{199},x^{398}$ and also $1,x^3,x^6,x^{12},x^{24},x^{48},x^{96},x^{192},x^{199}$, $x^{398}$. This saves on multiplication and this evaluates $(101\ 00\ 111\ 0)n_{2.}$

General algorithm is given below:

**Input**

- An element $x \in G$
- An integer $n=(n_l,n_{l-1},...,n_0)_2$, $n \in I^+$ and $k$>0, where $k$ is a parameter
- Pre-computed values $x^3$, $x^5$,... $x^{2k-1}$.

**Output**

Element $x^n \in G$

**Steps**

```
1. $y$=1 and $i$=l–1
2. while $i$>–1 do
3. if $n_i$=0 the $y=y^2$ and $i=i-1$
4. else
5. $s$=max{$i$–$k$+1,0}
6. while $n_s$=0 do $s=s+1$
7. for $h$=1 to $i$-$s$+1 do $y=y^2$
8. $u=(n_i,n_{i-1},....,n_s)_2$
9. $y=y*x^u$
```

```
10. $i=s-1$
11. return $y$
```

***Note:*** In the above algorithm, at line 6 the loop has longest string of length $\leq k$ ending in a nonzero value. Computation of all odd powers of 2 up to $2^{2k-1}$ is not required and those specifically involved in computation are considered.

NOTES

**Fixed Base Exponent**

There are several methods which can be employed to calculate $x^n$ when the base is fixed and the exponent varies. Pre-computations play a key role in these algorithms.

**(v) Yao's Method**

Yao's method is orthogonal to the $2k$-ary method where the exponent is expanded in radix $b=2^k$ and the computation is as performed in the algorithm above. Let '$n$', '$n_i$', '$b$', and '$b_i$' be integers. Let the exponent '$n$' be written as

$$n = \sum_{i=0}^{l-1} n_i b_i \text{ where } 0 \leq n_i < h \text{ for all } i \in [0, l-1]$$

Let $x_i = x_i^b$ Then the algorithm uses equality as,

$$x^n = \prod_{i=0}^{l-1} x_i^{n_i} = \prod_{j=1}^{h-1} \left[ \prod_{n_i = j} x_i \right]^j$$

Since $x \in G$, exponent '$n$' is written along with pre-computed values of $x^b_0 .... x^b_{l-1}$ the element $x^n$ is computed by the use of algorithm below:

```
1. $y=1, u=1$ and $j=h-1$
2. while $j > 0$ do
3. for $i=0$ to $l-1$ do
4. if $n_i=j$ then $u=u* x_i^b$
5. $y=y*u$
6. $j=j-1$
7. return $y$
```

By setting $h=2^k$ and $b_i = h^i$ then $n_i$'s are digits of $n$ in base $h$. Yao's method does the collection in $u$, first those $x_i$ appearing for the highest power $h$–1and in the next round also those having h–2 as power get collected in $u$ and like that. The variable $y$ is multiplied $h$–1 times with the initial $u$, $h$–2 times with the next highest powers, and so on. This algorithm makes use of $1+h-2$ multiplications and $l+1$ has to be stored for computing $x^n$.

### (vi) Euclidean Method

The Euclidean method was first introduced in 'efficient exponentiation using precomputation and vector addition chains' by P.D Rooij. The algorithm below computes $x^n$ using the following equality recursively:

$$x_0^{n0} x_1^{n1} = (x_0 x_1^q)^{n_0} \,.\, x_1^{(n_1 \bmod n_0)} \text{ where } q = \lceil n_1 / n_0 \rceil$$

If $x \in G$ and exponent '$n$' is written the way it is written in Yao's method having computed values of $x^b{}_0 \ldots x^b{}_{l-1}$ the element $x^n$ is computed by making use of the algorithm below:

**Steps**

```
1. while true do
2. find $M$ such that $n_M \geq n_i$ for all $i$ in $[0,l-1]$
3. find $N \neq M$ such that $n_N \geq n_i$ for all $i$ in $[0,l-1], i \neq M$
4. if $n_N \neq 0$ then
5. $q = \lfloor (n_M / n_N) \rceil$, $x_N = x_M^{q*} x_N$ and $n_M = n_M \bmod n_N$
6. else break
7. return $x_M^n$
```

The algorithm first finds the largest value amongst the $n_i$ and then the supremum within the set of $\{n_i : i \neq M\}$. It raises $x_M$ to the power $q$, multiplies this value with $x_N$ and then assigns $x_N$ the result of this computation and $n_M$ the value $n_M$ modulo $n_N$.

**Further applications**

Using this idea, speedy calculations for large exponents modulo division of a number is done, that has applications in cryptography. Computation of powers in a ring of integers modulo $q$ is very useful and use can be made of this in computing integer powers in a group. For this following rule is used,

$$\text{Power}(x, -n) = (\text{Power}(x, n))^{-1}.$$

In every semi-group this method works and this is often used for computing powers of matrices.

For example, in evaluating $13789^{722341}$ (mod 2345), if a naïve method is used then it will require very long time and storage space. It involves computing $13789^{722341}$ and taking remainder after dividing by 2345. Even if more effective methods are used it will consume long time as it will first square 13789, then would find remainder after dividing by 2345 and further multiply this result by 13789. This process will continue involving 722340 modular multiplications. The square-and-multiply algorithm is based on $13789^{722341} = 13789(13789^2)^{361170}$. So, when $13789^2$ is computed, full computation would consist of 361170 modular multiplications and there is a gain of a factor of two. As new problem is also the same in its type, same observation can be applied again, and approximately halving the size once more.

Repeated application of this algorithm is equivalent to decomposing the exponent by performing a base conversion of decimal to binary as a sequence of squares and products. This is explained as follows:

$$
\begin{aligned}
x^{11} &= x^{(1011)\text{bin}} \\
&= x^{(1*2\wedge3\,+\,0*2\wedge2\,+\,1*2\wedge1\,+\,1*2\wedge0)} \\
&= x^{1*2\wedge3} * x^{0*2\wedge2} * x^{1*2\wedge1} * x^{1*2\wedge0} \\
&= x^{2\wedge3} * 1 * x^2 * x^{2\wedge0} \\
&= x^8 * x^2 * x^1 \\
&= (x^4)^2 * (x)^2 * x \\
&= (x^4 * x)^2 * x \\
&= ((x^2)^2 * x)^2 * x \\
&= (x^2 * x)^2 * x
\end{aligned}
$$

$\rightarrow$ Thus, algorithm requires only 3 multiplications instead of 10 (11-1)

Some more examples:

- $x^{10} = ((x^2)^2*x)^2$ because $10 = (1{,}010)_2 = 2^3+2^1$, 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.
- $x^{1,000} = ((((((((x^2*x)^2*x)^2*x)^2*x)^2)^2*x)^2)^2)^2$ because $10^3 = (1{,}111{,}101{,}000)_2$, algorithm needs 14 multiplications instead of 999.
- $x^{1,000,000} = ((((((((((((((((((x^2*x)^2*x)^2*x)^2)^2*x)^2)^2)^2)^2)^2*x)^2)^2)^2*x)^2)^2)^2)^2)^2)^2$ because $10^6 = (11{,}110{,}100{,}001{,}001{,}000{,}000)_2$, algorithm needs 25 multiplications instead of 999,999.
- $x^{1,000,000,000} = ((((((((((((((((((((((((((((x^2*x)^2*x)^2)^2*x)^2*x)^2*x)^2)^2)^2*x)^2*x)^2)^2*x)^2)^2*x)^2*x)^2)^2)^2*x)^2)^2*x)^2)^2)^2)^2)^2)^2)^2)^2)^2$ because $10^9 = (111{,}011{,}100{,}110{,}101{,}100{,}101{,}000{,}000{,}000)_2$, algorithm needs 41 multiplications instead of 999,999,999.

## CHECK YOUR PROGRESS

8. What are the two basic rules of exponentiation?
9. List the methods used to compute exponentiation fast.

## 1.4 LINEAR SEARCH

Linear search is the easiest and least efficient searching technique. In this technique, the given list of elements is scanned till either the required element is found or the list is exhausted. This technique is used in direct access media such as magnetic tapes.

Example 1.14 illustrutes a linear search.

**Example 1.14:** Find an element 77 from the given list using linear search. The list of elements is 10, 25, 77, 16, 47 and 98.

Linear search starts by checking the target element (i.e., 77) with the first element of the list, i.e., 10, which is not equal to the target element; search continues with the second element, i.e. 25, which is also not equal to the target element and search continues with the third element, i.e. 77, which is equal to the target element (=77). So, the search is stopped.

### 1.4.1 Algorithm for Linear Search

```
LINEAR_SEARCH (L, N, E)
    1. [Initialization]
      loc = 1
      L[N + 1] = E
    2. [Search the element in the vector]
      REPEAT WHILE ( K[loc]<> E ) DO
         loc = loc + 1
    3. [Check whether the search is successful or not?]
      IF loc = N + 1, THEN WRITE ('UNSUCCESSFUL SEARCH')
         RETURN(0)
      ELSE WRITE('SUCCESSFUL SEARCH')
         RETURN(loc)
```

### 1.4.2 Analysis of Linear Search Algorithm

For N total number of elements, the search time T is proportional to half of N:

```
T = K * N/2 where K is a constant
If K = 2, then T = K*N
```

The average linear search times are proportional to the size of the array, i.e., $O(N)$

***Note:*** If an array is twice as big, it will take twice as long to search.

**Implementation of Linear Search to Find a String from a String Vector/ Array**

**Program for Linear Search of Strings**

```
/*—————————START OF PROGRAM——————————*/
   #include<stdio.h>
   #define MAXROWS 10
```

```
#define MAXCOLS 20
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int LSearch(STRINGS s,STRING target,int n)
{
 int loc=0;
 strcpy(s[n],target);
 while(strcmp(s[loc],target))
   loc++;
 if(loc==n)
   return NOTFOUND;
 else
   return loc;
}
void main()
{
STRINGS a={"MON","TUE","WED","THU","FRI","SAT","SUN"};
int index;
index=LSearch(a,"WED",7);
if(index==NOTFOUND)
      printf("Record not found");
else
      printf("Record found at Location:%d",index+1);
}
/*—————————END OF PROGRAM————————*/
```

**Output:** Record found at Location 3

**Implementation of linear search to find a value in a vector or array**

```
/*———————START OF PROGRAM—————————*/
#include<stdio.h>
#include<conio.h>
#define MAXROWS 10
#define MAXCOLS 20
#define NOTFOUND -1
typedef int VECTOR[MAXCOLS];
int LSearch(VECTOR s,int target,int n)
{
 int loc=0;
 s[n]=target;
 while(s[loc]!=target)
```

```
      loc++;
    if(loc==n)
     return NOTFOUND;
    else
    return loc;
  }
  void main()
  {
  VECTOR a={5,4,3,2,7};
  int index;
  clrscr();
  Program for Linear Search of Numbers
  index=LSearch(a,2,5);
  if(index==NOTFOUND)
  printf("Record not found");
  else
  printf("Record found at Location:%d",index+1);
  }
  /*————————————END OF PROGRAM——————————*/
  Output: Record found at Location:4
```

NOTES

## 1.5 BINARY SEARCH

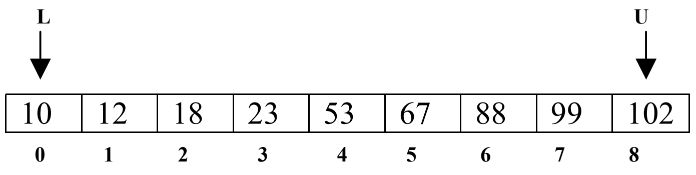
Binary search is used to search for an element in a sorted list.

### 1.5.1 The Search Method

- First compare the key with the item in the middle position of the array.
- If any match is found, return it immediately.
- If the key is less than the middle key, then the item to be found must lie in the lower half of the array; if it is greater, then the item to be found must lie in the upper half of the array.
- Repeat the procedure on the lower (or upper) half of the array.

  Example 1.15 illustrates a binary search

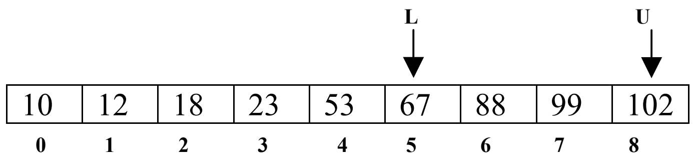
**Example 1.15:** Find an element 88 in an array of elements given below where **L** is Lower bound of the array and **U** is the Upper bound of the array.



Calculate middle by **M**=FLOOR((**L**+**U**)/2), where **L**=0 and **U**=8

**M**=4

Since value in **Vector[M]** is 53, which is less than the target value (=88), search in the second half of the array.



Calculate middle by **M**=FLOOR((**L**+**U**)/2), where **L**=5 and **U**=8

**M**=6

Since value in **Array[M]** is equal to target value (=88), then it is a successful search and record found at location 6.

### 1.5.2 Algorithm for Binary Search

```
BINARY_SEARCH(B,N,E)
1. [Initialization]
   L=1
   H=N
2. [Start the searching process]
   REPEAT THRU STEP 4 WHILE L<=H DO
3. [Get the index of midpoint of interval]
   M=FLOOR(L+H)/2
4.[Comparison to get the element]
  IF E < B[M] THEN
    H=M-1
  ELSE
    IF E > B[M] THEN
       L = M +1
    ELSE
        WRITE('SUCCESSFUL SEARCH')
     RETURN(M)
5. [Unsuccessful search]
  WRITE('UNSUCCESSFUL SEARCH')
6.[Finished]
  RETURN(0)
```

### 1.5.3 Analysis of Binary Search algorithm

For $N$ total number of elements, the search time $T$ is proportional to $log(N)$ $T=K * log_2(N)$.

The average searching time for binary search is $O(log N)$.

**Implementation of Binary Search to Find an Element in a Sorted Vector/ Array**

**Program for Binary Search for Numbers**

NOTES

```
*———————————START OF PROGRAM——————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define NOTFOUND -1
typedef int VECTOR[MAXCOLS];
int BSearch(VECTOR str,int target,int n)
{   int s,e,m,cmp;
s=0;
e=n-1;
while(s<=e)
{   m=(s+e)/2;
if(target<str[m])
          e=m-1;
          else
          if(target>str[m])
             s=m+1;
          else  return m;
 }
 return NOTFOUND;
 }
void main()
{
VECTOR a={1,2,3,4,5};
int index;
clrscr();
index=BSearch(a,4,5);
if(index==NOTFOUND)
printf("Record not found");
else
       printf("Record found at Location:%d",index+1);
}
/*————————————END OF PROGRAM——————————*/
```

**Output:**

```
Record found at Location:5
```

**Implementation to Search a String in a Vector/Array having Strings in Sorted Order**

```
/*——————————START OF PROGRAM——————————*/
#include<stdio.h>
#include<conio.h>
#define MAXROWS 10
#define MAXCOLS 20
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int BSearch(STRINGS str,STRING target,int n)
{
    int s,e,m,cmp;
s=0;
    e=n-1;
while(s<=e)
{
m=(s+e)/2;
cmp=strcmp(target,str[m]);
if(cmp<0)
          e=m-1;
 else
          if(cmp>0)
            s=m+1;
          else
          return m;
 }
 return NOTFOUND;
 }
void main()
{
STRINGS str={"AB","ABC","BB","BCA","CC","CCC"};
int index;
clrscr();
index=BSearch(str,"CC",6);
if(index==NOTFOUND)
printf("Record not found");
else
    printf("Record found at Location:%d",index+1);
}
/*————————END OF PROGRAM——————————*/
```

**Output:**

```
Record found at Location:5
```

***Note:*** In the above two programs, the array should contain sorted values; otherwise, use any sorting algorithm before calling **BSearch.**

NOTES

**Algorithm for Binary Search using Recursive Technique**

**Function BSearch**(Vector,First_Index,Second_Index,Target)

```
1.[Search vector between First_Index and Second_Index
      for target]
 IF First_Index>Second_Index)
     loc=0
 ELSE
   Middle_Index=(First_Index+Second_Index)/2;
   IF Target > Vector[Middle_Index]

  loc=BSearch(Vector,Middle_Index+1,Second_Index,Target)
      ELSE
       IF Target < Vector[Middle_Index]
        loc=BSearch(Vector,First_Index,Middle_Index-1,
Target)
       ELSE
        loc=Middle_Index
2.[Finished]
  RETURN(loc)
```

**Implementation of Binary Search to Find an Element in a Sorted Vector/ Array using Recursion Technique**

**Program for binary search for numbers using recursion**

```
     #include <stdio.h>
     #define MAXCOLS 20
     #define NOTFOUND -1
     typedef int VECTOR[MAXCOLS];
     int BSearch(VECTOR vector,int findex,int sindex,int
target)
     {
      int mindex,loc;
      if(findex>sindex)
         loc=NOTFOUND;
      else
      {
        mindex=(findex+sindex)/2;
```

NOTES

```
           if(target>vector[mindex])
              loc=BSearch(vector,mindex+1,sindex,target);
           else
              if(target<vector[mindex])
                loc=BSearch(vector,findex,mindex-1,target);
              else
                loc=mindex;
      }
    return(loc);
    }
    void main()
    {
     VECTOR a={10,20,30,40,50,60,70,80,90};
     int loc;
     loc=BSearch(a,0,8,40);
     if(loc==NOTFOUND)
     printf("Target string not found");
     else
      printf("Starting from 0th location target is at
location:%d",loc);
    }
```

**Output**

```
Starting from 0th location target is at Location:3
```

## Implementation to Search a String in a Vector/Array having Strings in Sorted Order using Recursion Technique

### Program for binary search for strings using recursion

```
#include <stdio.h>
#include <string.h>
#define MAXCOLS 20
#define MAXROWS 10
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int BSearch(STRINGS str,int findex,int sindex,STRING
target)
{
 int mindex,loc,cmp;
 if(findex>sindex)
    loc=NOTFOUND;
 else
 { mindex=(findex+sindex)/2;
```

```
     cmp=strcmpi(target,str[mindex]);
     if(cmp<0)        /* if target greater than middle
string */
      loc=BSearch(str,mindex+1,sindex,target);
     else
        if(cmp>0)      /* if target less than middle string
*/
           loc=BSearch(str,findex,mindex-1,target);
        else
           loc=mindex;
 }
return(loc);
}
void main()
{
 STRINGS str[]={"aa","bb","cc","dd"};
 int loc;
 loc=BSearch(str,0,3,"bb");
 if(loc==NOTFOUND)
 printf("Target string not found");
 else
  printf("Starting from 0th location target is at
Location:%d",loc);
}
```

**Output**

```
Starting from 0th location target is at Location:1
```

### 1.5.4 Fibonacci Search

The Fibonacci progression is a numeric progression such that $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1}+F_{n-2}$ for $n_2$. The Fibonacci search splits the given list of elements according to the Fibonacci progression unlike splitting in middle as in the binary search.

**Algorithm for Fibonacci search**

```
Function Fibonacci_search(Array, Target, N)
1. [Initialize I with 0]
   I = 0
2. [Check ?]
   WHILE(Fib(I) < N)
     I = I + 1
3. [Assignments]
   A = Fib(I - 2)
   B = Fib(I - 3)
```

```
4. [Calculate middle element]
 Middle=N-A-1
5. [Search process]
 WHILE Array[Middle]<>Target DO
     IF Array[Middle] > Target THEN
          IF b < 0 THEN
             RETURN NOTFOUND
           T = A – B
           Middle = Middle – B
           A = B
           B = T
       ELSE
          IF A < 1 THEN
            RETURN NOTFOUND
           MIDDLE = MIDDLE + B
           A = A – B
           B = B – A
 6.[Finished]
 RETURN Middle
```

## Algorithm for Fibonacci Function

```
Function Fib(N).
1.[Generate number]
 IF N = 0 THEN
  RETURN 0
 ELSE
  IF N = 1 THEN
   RETURN 1
  ELSE
   RETURN Fib(n – 1)+Fib(n – 2)
```

## Analysis of Fibonacci Search Algorithm

Fibonacci numbers grow exponentially, it immediately follows that any node with $N$ descendants that has rank at most $O(logN)$

The average searching time for Fibonacci search is $O(log N)$.

## Implementation to Search a String in a Vector/Array having Strings in Sorted Order using Fibonacci Search

## Program for Fibonacci search for strings

```
/*———————————STARTING THE PROGRAM——————————*/
#include<stdio.h>
```

NOTES

```
#include<conio.h>
#include<string.h>
#define MAXCOLS 20
#define MAXROWS 10
#define NOTFOUND -1
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
int Fib(int n)
{ if(n==0)
  return 0;
 else
  if(n==1)
   return 1;
  else
   return Fib(n – 1) + Fib(n – 2);
}
int Fsearch(STRINGS str, STRING target, int n)
{ int i,a,b,middle,t;
 i = 0;
 while(Fib(i) < n)
       i++;
 a=Fib(i – 2);
 b=Fib(i – 3);
 middle = n – a – 1;
 while(strcmpi(str[middle],target)!=0)
 { if(strcmpi(str[middle],target)>0)
     {
        if(b < 0)
           return NOTFOUND;
        t = a – b;
        middle = middle – b;
        a = b;
        b = t;
     }
   else
       { if(a < 1)
           return – 1;
    middle = middle + b;
        a = a – b;
        b = b – a;
       }
```

NOTES

```
 }
 return(middle);
}
void main()
{
 int i,n;
 STRINGS str[]={"aa","bb","cc","dd","ee","ff","gg"};
 i=Fsearch(str,"gg",7);
 if(i==NOTFOUND)
 printf("\nRecord not found");
 else
  printf("\nStarting from 0th location record found
at:%d",i);
}
```

**Output**

```
    Starting from 0th location record found at:6
```

## 1.6 BIG OH NOTATION (OR BIG O NOTATION)

An algorithm is a step-by-step procedure for performing some task in a finite amount of time. Sometimes we need to know how much time and space (computer memory) a computer algorithm requires, i.e., how efficient it is. This is termed as **time** and **space** complexity. Typically, the complexity refers to a function of the values of the inputs, and we would like to know what is that function. The best, average and worst cases can also be considered.

The big O notation (also known as big OH notation) provides a convenient way to compare the speed of algorithms. This is a mathematical notation used in the priori analysis. If an algorithm is said to have a computing time of O(g(n)), then it implies that if the algorithm is run on some computer on the same type of data put for increasing the values of n, the resulting times will always be less than same constant times |g(n)|.

The best algorithm runs in O(1) times. Good algorithm runs in O(log N) times. Fair algorithm runs in O(N) times. Worst algorithm runs in $O(N^2)$ times.

**Note:** If $A(n) = a_m n^m + ...+a_1n^1 + a_0$ is a polynomial of degree m, then $f(n) = O(n^m)$. Thus, if the frequency of execution of a statement is in the form of A(n), then the statements computing time will be $O(n^m)$.

Formally, O(g(n)) is the set of functions, f, such that for some c > 0, f(n) < cg(n) for all positive integers, n > N, i.e. for all sufficiently large N. It can be represented as $\lim_{n\to\infty} \frac{f(n)}{g(n)} \le c$ .

Informally, we say the O(g) is the set of all functions, which grows no faster than g. The function g is an upper bound to functions in O(g).

We can analyse any algorithm by the O notation irrespective of the programming language and machine.

Consider two other functions: $\Omega(g)$ and $\Theta(g)$.

$\Omega(g)$ is the set of functions f(n) for which f(n) $\geq cg(n)$ for all positive integers, n>N, and $\Theta(g) = \Omega(g) \cap O(g)$

### 1.6.1 Properties of the Big O Notation

The following are the properties of big O notation:

- Constant factors may be ignored: For all k > 0, kf is O(f).

  **e.g.** $cn^2$ and $kn^2$ are both $O(n^2)$.

- Higher powers of n grow faster than lower powers: $n^r$ is $O(n^8)$ if $0 \leq r \leq s$.
- The growth rate of a sum of terms is the growth rate of its fastest growing term: If f is O(g), then f+g is O(g).

  **e.g.** $an^3+bn^2$ is $O(n^3)$.

- The growth rate of polynomial is given by the growth rate of its leading term: If f is a polynomial of degree d, then f is $O(n^d)$.
- If f grows faster than g, which grows faster than h, then f grows faster than h.
- The product of upper bounds of functions gives an upper bound for the product of the functions: If f is O(g) and h is O(r) , then fh is O(gr).

  **e.g.** If f is $O(n^2)$ and g is O(log n), then fg is $O(n^2 \log n)$.

- Exponential functions grow faster than powers: $n^k$ is $O(b^n)$, for all b > 1, k,

  **e.g.** $n^4$ is $O(2^n)$ and $n^4$ is O(exp(n)).

- Logarithms grow more slowly than powers: $\log_b n$ is $O(n^k)$ for all b > 1, k > 0

  **e.g.** $\log_2 n$ is $O(n^{0.5})$.

- All logarithms grow at the same rate: $\log_b n$ is $\Theta(\log_d n)$ for all b, d>1.
- The sum of the n rth powers grows as the (r+1)th power: $\sum_{k=1}^{n} k^r is \Theta(n^{r+1})$

  **e.g.** $\sum_{k=1}^{n} i = \frac{(n+1)n}{2} is \Theta(n^2)$

### 1.6.2 General Rules

- **Simple statement sequence:** It is to be noted first that a sequence of statements executed once only is O(1). It is immaterial as to how many statements are in the sequence; only that the number of statements (or the time that they take to execute) is constant for all problems.
- **Simple loops:** If a problem of size n can be solved with a simple loop. For example,

  ```
  for (i = 0;i < n; ++i)
  {
     Statement(s);
  }
  ```

  Where Statement(s) is an O(1) sequence of statements, then the time complexity is nO(1) or O(n).
- **Nested loops:**

  ```
  for(j = 0;j < n; ++j)
           for(i = 0;i < n; ++i)
           {
             Statement(s);
           }
  ```

  when we have n repetitions of an O(n) sequence, then the complexity is nO(n) or O($n^2$).
- **Loop index does not vary linearly:** Where the index jumps by an increasing amount in each iteration.

  ```
  i = 1;
  while(i ≤ n)
  {
   Statement(s);
   i = 2*i;
  }
  ```

  in which i takes values 1, 2, 4,… until it exceeds n. This sequence has $1 + \lfloor \log_2 n \rfloor$ values, so the complexity is O($\log_2$n).
- **If the inner loop depends on an outer loop index:**

  ```
  for(j = 0;j < n; j++)
     for(i = 0;i < j; i++)
     {
     Statement(s);
     }
  ```

  The inner loop i = 0, 1, 2…n gets executed n times, so the total is:

  $\sum_{1}^{n} i = \frac{n(n+1)}{2}$ and the complexity is O($n^2$).

Notice that the above two nested loops also have the same complexity, so the variable number of iterations of the inner loop does not affect the 'big picture'. However, if the number of iterations of one of the loops decreases by a constant factor with every iteration as shown below:

```
i = n;
while(i > 0)
{  for(i = 0;i < n; ++i)
 {
    Statement(s);
   }
 h = h/2;
}
```

NOTES

Then there are $\log_2 n$ iterations of the outer loop and the inner loop is O(n). So the overall complexity is O(n log n) .

The most common computing times of algorithms in the big O notation are:

$$\boxed{\mathbf{O}(1)<\mathbf{O}(\log n)<\mathbf{O}(n)<\mathbf{O}(n\log n)<\mathbf{O}(n^2) <\mathbf{O}(n^3)<\mathbf{O}(2^n)<=\mathbf{O}(n!)}$$

### 1.6.3 Finding Prime Factor of a Given Number

Finding a prime factor begins with the lowest prime number 2. If 2 divides the number completely and leaves no remainder it is marked as the very first prime factor. It continues dividing until it longer divides evenly. Then the control flow moves to the next lowest prime numbers. The step is repeated until the next prime factor comes. The following algorithm is used to find the prime factor of a given number:

**Algorithm to Find Prime Factor of a Given Number**

```
Step 1: integer input, divisor, count;
Step 2: print 'Enter a value:',
Step 3: read input;
Step 4: count←0;
Step 5: Do
Step 6: divisor←0;
Step 7: if input mod 2==0 OR input==1
//To remove all the factors of 2
break;
count←count+1; //Increase counter value by 1
print count, divisor;
input←input/2; //Remove this factor from input
Step 8:End Do
Step 9: divisor←3;
Step 10: Do
Step 11: if divisor>input
```

```
break;
Step 12: Do //Remove the factors repeatedly
Step 13: if input mod divisor ==0 OR input==1
break;
count←count+1;
print count, divisor;
input←input/divisor;
//Remove factors from input
Step 14: End Do
Step 15: divisor← divisor+2;
//Move to next odd number
Step 16: End Do
```

The above algorithm lists out all prime factors of an `n integer >=2`. First it sides back all factors of `2`. Then, all factors, such as `3, 5, 7` and so on can be removed. This process is run until all the prime factors are sided back and kept in a temporary location. According to the above algorithm, if the input value is 53, the prime factors of 53 are 1 and 53 itself.

**Implementation of Finding Prime Factor of a Given Number**

```
/*———————— START OF PROGRAM ————————*/
#include <stdio.h>
#include <conio.h>
void main()
{
int number, i, j, k;
clrscr();
printf("Enter a number:");
scanf("%d", &d);
while(i<=number)
{
k=0;
if (number%i==0)
{
j=1;
while(j<=i)
{
if(i%j==0)
k++; //Value k is increased by 1
j++; //Value j is increased by 1
}
if(k==2)
printf("\nPrime factors are:");
printf("%d",i);
i++;
}
getch();
}
```

NOTES

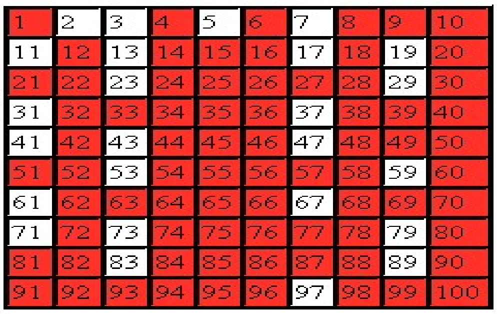
**The result of the above program is as follows:**

```
Enter a number: 123
Prime factors are: 3 41
```

In the above program, finding a prime factor of a given number can be performed in the following step. You first enter a number let say '123' as input value. The prime factors of 123 are 3 and 41 (prime numbers). If you multiple 3 and 41, it returns 123, that is a prime number.

### 1.6.4 List of Prime Numbers

Prime numbers are numbers that can be divided only by 1 or by themselves. Below, in white, are the prime numbers between 1 and 100.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

The following algorithm is used to find the list of prime numbers:

**Algorithm to Find the List of Prime Numbers**

```
Step 1: integer N,D;
// Declare the variables the integer being considered
needed for the integer divison
Step 2: integer N_is_prime;
// N is equal to 1 (default) when N is prime and N = 0
when N is not prime
Step 3: for N←3 to 30 //Running for loop
// This loop considers all prime integers between 3 and
30
N_is_prime←1;
// assume N is prime
Step 4: for D←2 to (N-1)
if N%D == 0
//Returns remainder if N is divided by D
N_is_prime = 0;
// if the remainder is 0 then N is prime
```

```
Step 5: if N_is_prime == 0
break; //Exit from loop
// if N is prime do not do any more integer divisions
Step 6: if N_is_prime == 1
print N;
```

Implementation to find the list of prime numbers

```
/*——————————— START OF PROGRAM ———————————*/
#include<stdio.h>
#include <conio.h>
void main()
{
int N; // the integer being considered
int D; // needed for the integer divison
int N_is_prime; // = 1 (default) when N is prime and = 0
when N is not prime
for (N=3;N<=30;N++)
// This loop considers all prime integers between 3 and
100
{
N_is_prime = 1; // assume N is prime
for (D=2;D<=N-1;D++)
{
if ( N%D == 0 ) N_is_prime = 0;
// if the remainder is 0 then N is prime
if (N_is_prime == 0)
break;
// if N is prime don't do any more integer divisions
}
if (N_is_prime == 1)
printf("%d\n",N);
}
getch();
}
```

The result of the above program is as follows:

```
2
3
5
7
11
13
17
```

```
19
23
29
```

**CHECK YOUR PROGRESS**

10. What do you understand by the linear search technique?
11. When is the binary search method used?
12. What is big O notation?

NOTES

## 1.7 WORST CASE

In the field of computer science, complexity indicates a measure of resources required by an algorithm and worst-case complexity is a measure of resources required by the algorithm to solve a problem in worst case. Here, by 'resources' we mean time of run and memory required. These are time and space complexity. Worst-case indicates the maximum amount of resource required by the algorithm to solve the problem.

Measured in terms of time required, worst-case time-complexity gives maximum time required by an algorithm to perform when any input of size *n* is given with a guarantee that algorithm finishes its work within that time. This also forms a basis for comparing efficiency of two algorithms.

Best case, worst case and average cases are three situations in which analysis is done for a given algorithm. These terms tell about use of resources in terms of *at least*, *at most* and *on average*, respectively. Here, resources usually include running time and amount of memory space the algorithm occupies. It may mean other resources as well, but mostly time and space complexities are used. Worst-case execution time is of particular importance in real time computing as it is critical to know the maximum time required to execute instructions with a guarantee that the algorithm will always finish within that time.

Worst-case is compared with average performance and is mostly used in algorithm analysis. In case of a looping statement there is O(*n*) time complexity. If another loop is put in that loop the complexity increases and it becomes O(*n*^2), since it has to do *n*^2 things for an input size of *n*.

Worst case algorithms are non deterministic.

## 1.8 ADVANTAGE OF LOGARITHMIC ALGORITHMS OVER LINEAR ALGORITHMS

An algorithm that takes search time of the order of O(log *n*) is logarithmic whereas a linear algorithm takes search time of O(*kn*) to find *k*th smallest or largest item in a list containing *n* items. This linear algorithm is not effective when list is large and value of $k$ is large. Such algorithm is suitable only for the cases where value of $k$ is small. Binary search is an example of logarithmic search.

In linear algorithm, program moves in a sequential manner, whereas a binary search adopts the policy of 'divide and conquer'. For example, if we start searching a number, say 30, in a list of 100 numbers, and the list is serially arranged then it will search linearly from 1 to 29 and then come to 30. But in Binary search, it would divide in two halves and will discard one half and will again search in another half. This search strategy reduces the number checking by a factor of two each time to find the target value, if it is in the list and this will be done in logarithmic time.

Suppose we want to design an algorithm to find whether determinant value of a particular $n \times n$ matrix is more than a predetermined value $k$. Using linear search this can be done in $O(n^2)$ time whereas using binary search, ceiling of the determinant value ($d$) can be found in $O(n^2 \log d)$ time. Here, $d$ is the size of output and not of the input.

Algorithm of logarithmic nature always takes less time that of linear search. A linear search has worst case behaviour of $N$ iterations and takes much time as $N$ grows. If there are one million items, it will make one million searches. But in case of logarithmic search such as binary search, time taken in worst case is floor value of $(\log_2 2^{10} - 1)$ which is approximately 9. Search is continued even if iteration fails to find a match at the probed position, with one or other of the two sub-intervals, each with almost halved size. If $N$, denoting the number items, is odd then both sub-intervals will have $(N-1)/2$ elements. But if $N$ is even the two sub-intervals will have $N/2 - 1$ and $N/2$ elements.

In a list of $N$ items, after first iteration, remaining items will $N/2$ items at the maximum and in second iteration, items will get reduced to at most $N/4$ items, then to $N/8$ items and so on. This way algorithm continues, iterating until the span becomes empty. This will, at the most will take $\lfloor \log_2(N) + 1 \rfloor$ iterations. Here, $\lfloor\ \rfloor$ stands for the floor function which neglects the decimal part of the number. In worst case for any $N$ it takes exactly $\lfloor (\log_2 N) + 1 \rfloor$ iterations.

Linear search can be applied to a list sorted as well as unsorted, but binary search is applied to a list that is already sorted.

## 1.9 COMPLEXITY

### 1.9.1 Space Complexity

The space complexity of an algorithm indicates the quantity of temporary storage required for running the algorithm, i.e. the amount of memory needed by the algorithm to run to completion.

In most cases, you do not count the storage required for the inputs or the outputs as part of the space complexity. This is because the space complexity is used to compare different algorithms for the same problem in which case the input/output requirements are fixed.

Also, you cannot do without the input or the output, and you want to count only the storage that may be served. You also do not count the storage required for the program itself since it is independent of the size of the input.

Like time complexity, space complexity refers to the worst case, and it is usually denoted as an asymptotic expression in the size of the input. Thus, *a* o(*n*) – space algorithm requires a constant amount of space independent of the size of the input.

The amount of memory an algorithm needs to run to completion is called its space complexity. The space required by an algorithm consists of the following two components:

**(*i*) Fixed or static part:** Fixed or static part is not dependent on the characteristics (such as number size) of the inputs and outputs. It includes various types spaces, such as instruction space (i.e., space for code), space for simple variables and fixed-size component variables, space for constants, etc.

**(*ii*) Variable or dynamic part:** Variable or dynamic part consists of the space required by component variables whose size is dependent on the particular problem instance at run-time being solved, the space needed by referenced variables and the recursion stack space (depends on instance characteristics).

The space requirements $S(p)$ of an algorithm $p$ is $S(p) = c + Sp$ (instance characteristics), where '$c$' is a constant.

We are supposed to concentrate on estimating SP (instance characteristics) since the first part is static.

The problem instances for algorithm are characterized by $n$, the number of elements to be summed. The space needed by $n$ is one word since it is of type integer. The space needed by $a$ is the space needed by variables of type array of floating-point numbers.

This is at least $n$ words since a must be large enough to hold the $n$ elements to be summed. So, we obtain $S_S(n) = (n + 3)$ ($n$ for $a[\ ]$, one each for $n$, $i$, and $s$).

**Iterative function for sum**

```
Algorithm RSum (a, n)
{
if (n 0) then     return 0.0;
else  return
RSum (a, n − 1) + a[n];
}
```

### 1.9.2 Time Complexity

The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.

The time T(P) consumed by a program P is the sum of the compile-time and the run-time (execution-time). The compile time is independent of the instance characteristics. Also, it may be assumed that a compiled program can be run many times without recompilation. As a result, we are more interested in the run-time of a program. This run-time is denoted by $t_p$ (instance characteristics).

Many factors on which $t_p$ depend are not known at the time a program is written; so it is always better to estimate $t_p$. If we happen to know the type of the compiler used, then we could proceed to find the number of additions, subtractions, multiplications, divisions, compare statements, loads, stores and so on that would be made by a program P.

So we can obtain an expression of the form.

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

Where $n$ denotes the instance characteristics, and $C_a$, $C_s$, $C_m$, $C_d$ and so on denote the time needed for addition, subtraction, multiplication, division, etc.

But here we need to note that the exact amount of time needed for the operations mentioned here cannot be found exactly; so instead we could only count the number of program steps, which means that a program step is counted.

A program step is defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

For example, consider the statement return a + b * c – d % e/f

This can be regarded as a step since its execution time is independent of the instance characteristics.

The number of steps that are assigned to any program statement depend on the type of statement. The comments do not count for the program step. A general assignment statement, which does not call another algorithm, is considered one step whereas in an iterative statement like for, while and repeat_until, you count the step only for the control part of the statement.

The general syntax for 'for' and 'while' statements is as follows:

```
for i = (exprl) to (expr2) do
while(expr) do
```

Each execution of the control part of a while statement is given step count equal to the number of step counts assignable to <expr>. The step count for each execution of the control part of a for statement is one, unless the counts attributable to <expr> and <exprl> are functions of the instance characteristics.

### 1.9.3 Practical Complexities

One of the main reasons to analyse algorithms is to find out the relative performance of two or more algorithms for the same problem. Consider the problem of sorting an array `a[0:n - 1]` for which the best known algorithms are bubble sort, selection sort, insertion sort, quick sort, merge sort, etc. The time complexities of these algorithms fall in two categories. One set of algorithm have O($n^2$) and the other set have O($n$ log $n$). Unless we analyse and find their complexities, it may be difficult to say which is faster and which is not.

While comparing the algorithms we must also keep in mind that n is very large or sufficiently large. The performance of the algorithms strongly depends on the size of $n$ too. For instance, quick sort behaves very badly when n is small, i.e., O($n^2$), which is same as bubble sort or insertion sort. However, when the size is sufficiently large its time complexity is O($n$ log $n$).

The aim of this section is to illustrate the practical time calculation of a program, what are the timing functions to use (given by the operating system or the complier), how to set up the test data, calculate the actual time taken by common asymptotic functions, $f(n)$ using a real computer executing certain number of instructions per second. You will be wondering that certain functions may take years of computer time (even on a Pentium III or mainframe machines) for sufficiently large values of $n$.

**Typical Examples**

(1) Assume that a computer can execute $10^9$ steps/sec and a particular program needs $n^{10}$ steps, then:

for $n = 10$, the time required is = 10 sec

for $n = 100$, the time required is = 3171 years

for $n = 1000$, the time required is = $3.17 \times 10^{13}$ years

(2) Assuming that out task needs $n^2$ steps (bubble sort to selection sort) then:

for $n = 10$, the time required is = 0.1 μsec

for $n = 100$, the time required is = 10 μsec

for $n = 1000$, the time required is = 1msec

for $n = 1000000$, the time required is = 16.6 min

Looking at the first example, we see that even a polynomial function $f(n^{10})$ takes enormous amount of time (unimaginable $\approx 10^{13}$ years). When the function is exponential, (say $2^n$) imagine what could be the time taken.

NOTES

Table 1.1 shows the run-time on a 1,000,000,000 instructions/sec.

***Table 1.1*** *Run-time for Various Values of n*

| $n$ | $n$ | $n \log n$ | $n^2$ | $n^{10}$ | $2^n$ |
|---|---|---|---|---|---|
| 10 | 0.01μs | 0.03μs | 0.1μs | 10s | 1μs |
| 50 | 0.05μs | 0.28μs | 2.5μs | 3.1yrs | 13 days |
| 100 | 0.01μs | 0.66μs | 10μs | 3171yrs | $4 \times 10^{13}$ yrs |
| 1000 | 1μs | 9.96μs | 1ms | $3.17 \times 10^{13}$ yrs | $3.17 \times 10^{43}$ yrs |
| 1000000 | 1ms | 19.92ms | 16.67min | $3.17 \times 10^{43}$ yrs | - |

### 1.9.4 Performance Measurement

This section explains how the run-time of a function is calculated using specific complier options. You will not show the space calculations and the compilation time in this example. There is of course a small difference between the time calculation on DOS/Windows 95 or 98 environment and UNIX environment.

Therefore, the reader must be careful in introducing the appropriate statement in his/her program depending on the platform.

The test data is another important aspect of time complexity calculation. For example, when you want to generate the test data for bubble sort (or for that matter any sorting or searching program) function, it is better to use a random number generator to populate the array. This will enable you to find the run-time on an average case or worst case.

Programs need to be tested and in turn, the time complexity need to be calculated. For example, in the case of bubble sort, the best case is to input a sorted sequence itself. In case of the searching algorithm, just put the key in the 1st position.

Example1.16 shows the famous factorial program with statements required for finding the time complexity on DOS platforms. The function gettime () obtains the time in a structure with hours, minutes, seconds and hundredth of a second. In the program shown, only hundredth of a second is considered assuming that the program does not take of the order of seconds. The time before and after calling the functions is recorded and the difference multiplied by 10 gives the result in milliseconds. If this program gets tested on a Pentium – III @ 500 MHz computer then for most of the values of *n*, you may get the result as 0 as the machine is fast as the program consumes less than a millisecond.

**Example 1.16:** Run-time Calculation – Factorial Program (DOS version)

```
          /* Factorial – Recursive method */
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<dos.h>
```

```
    long Fact(int);
    void main()
    {
       int n;
       long f;
       struct time t;
       long stime, etime;
       printf("Enter n:");
       scanf("%d", &n);
       gettime(&t);
       stime=t.ti_hund;
       f=Fact(n);
       gettime(&t);
       etime=t.ti_hund;
       printf("The Factorial is = %1d\n", f);
       printf("Time taken = %1d\n",(etime - stime) * 10);
/* time in msec */
    }
    long Fact(int n)
    {
       delay(10);
       if (n==0) return 1;
       else return n* Fact(n-1);
    }
```

On UNIX platform, you must follow the following piece of code:

```
    #include<sys/types.h>
    #include<sys/times.h>
    …………….
    …………….
    …………….
    main()
    {
    struct tms t; long stime, etime;
    ………….
    …………
    stime = times (&t);
       f( ); /* function whose run-time to be calculated
*/
       etime = times(&t);
    …………../* calculate the difference *10*/
    …………../* similar to program 1.11   */
}
```

NOTES

## 1.10 ALGORITHM REPRESENTATION THROUGH A PSEUDOCODE

A pseudocode is neither an algorithm nor a program. It is an art of expressing a program in simple English that parallels the forms of a computer language. It is basically useful for working out the logic of a program. Once the logic seems right, you can attend to the details of translating the pseudocode to the actual programming code. The advantage of pseudocode is that it lets you concentrate on the logic and organization of the program while sparing you the efforts of simultaneously worrying how to express the ideas in a computer language.

A simple example of pseudocode:

```
set highest to 100
set lowest to 1
ask user to choose a number
guess ( highest + lowest ) / 2
while guess is wrong, do the following:
   {
   if guess is high, set highest to old guess minus 1
   if guess is low, set lowest to old guess plus 1
   new guess is ( highest + lowest) / 2
   }
```

### 1.10.1 Coding

In the field of computer programming, the term **code** refers to instructions to a computer in a programming language. The terms '**code'** and '**to code'** have different meanings in computer programming. The noun '**code'** stands for source code or machine code. The verb '**to code'**, on the other hand, means writing source code to a program. This usage seems to have originated at the time when the first symbolic languages evolved and were punched onto cards as 'codes'.

It is a common practice among engineers to use the word 'code' to mean a single program. They may say 'I wrote a code' or 'I have two codes'. This inspires wincing among the literate software engineer or computer scientists. They rather prefer to say 'I wrote some code' or 'I have two programs'. As in English it is possible to use virtually any word as a verb, a programmer/coder may also say 'coded a program'; however, since a code is applicable to various concepts, a coder or programmer may say 'hard-coded it right into the program' as opposed to the meta-programming model, which might allow multiple reuses of the same piece of code to achieve multiple goals. As compared to a hard-coded concept, a soft-coded concept has a longer lifespan. This is the reason of soft-coding of concept by the coder.

While writing your code, you need to remember the following key points:

- **Linearity:** If you are using a procedural language, you need to ensure that code is linear at the first executable statement and continues to a final return or end of block statement.
- **If constructs:** You would better use several simpler nested 'if' constructs rather than a complicated and compound 'if' constructs.
- **Layout:** Code layout should be formatted in such a way that it provides clues to the flow of the implementation. Layout is an important part of coding. Thus, before a project starts, there should be agreement on the various layout factors, such as indentation, location of brackets, length of lines, use of tabs or spaces, use of white space, line spacing, etc.
- **External constants:** You should define constant values outside the code. It ensures easy maintenance. Changing hard-coded constants takes too much time and is prone to human error.
- **Error handling:** Writing some form of error handling into your code is equally important.
- **Portability:** Portable code makes it possible for the source file to be compiled with any compiler. It also allows the source file to be executed on any machine and operating system. However, creating a portable code is a fairly complex task. The machine-dependent and machine-independent codes should be kept in separate files.

### 1.10.2 Program Development Steps

The following steps are required to develop a program:

- Statement of the problem
- Analysis
- Designing
- Implementation
- Testing
- Documentation
- Maintenance

**Statement of the problem**: A problem should be explained clearly with required input/output and objectives of the problem. It makes easy to understand the problem to be solved.

**Analysis**: Analysis is the first technical step in the program development process. To find a better solution for a problem, an analyst must understand the problem statement, objectives and required tools for it.

**Designing**: The design phase will begin after the software analysis process. It is a multi-step process. It mainly focuses on data, architecture, user interfaces and program components. The importance of the designing is to get the quality of the product.

**Implementation**: A new system will be implemented based on the designing part. It includes coding and building of new software using a programming language and software tools. Clear and detailed designing greatly helps in generating effective code with less implementing time.

**Testing**: Program testing begins after the implementation. The importance of the software testing is in finding the uncover errors, assuring software quality and reviewing the analysis, design and implementation phases.

### 1.10.3 Software Testing

Software testing will be performed in the following two technical ways:

- **Black box tests** or **Behavioral tests** (testing in the **large**): These types of techniques focus on the information domain of the software.

  **Example:** Graph-based testing, Equivalence partitioning, Boundary value analysis, Comparison testing and Orthogonal array testing.

- **White box tests** or **Glass box tests** (testing in the **small**): These types of techniques focus on the program control structure.

  **Example:** Basis path testing and Condition testing

- **Documentation**: Documentation is descriptive information that explains the usage as well as functionality of the software.

  Documentation can be in several forms:

  - Documentation for programmers
  - Documentation for technical support
  - Documentation for end-users

- **Maintenance**: Software maintenance starts after the software installation. This activity includes amendments, measurements and tests in the existing software. In this activity, problems are fixed and the software updated to make the system faster and better.

Programming is the process of devising programs in order to achieve the desired goals using computers. A good program has the following qualities:

- A program should be **correct** and designed in accordance with the specifications so that anyone can understand the design of the program.
- A program should be **easy to understand.** It should be designed that anyone can understand its logic.
- A program should be **easy to maintain and update.**

- It should be **efficient** in terms of the speed and use of computer resources such as primary storage.
- It should be **reliable.**
- It should be **flexible**; that is to say, it should be able to operate with a wide range of inputs.

**NOTES**

## 1.11 AMORTIZED ANALYSIS

Amortized analysis means finding the average running time per operation over a worst-case sequence of operations. While giving the average case complexity, probability is involved. On the other hand, amortized analysis ensures the time per operation over the worst-case performance.

- Amortized analysis assumes the worst-case input and typically disallows random choices.
- The average case analysis and amortized analysis are two different concepts. In the former, we average all possible inputs, whereas in the latter, we average a sequence of operations.
- The amortized analysis disallows the random selection of input.

Various techniques are used in amortized analysis. They are discussed as follows:

- **Aggregate analysis:** In this type of analysis, the upper bound T(n) on the total cost of a sequence of n operations is decided; then the average cost is calculated as T(n)/n.
- **Accounting method:** In this method, the individual cost of an operation is calculated by combining the immediate execution time and its influence on the run- time of future operations.
- **Potential method:** This method is very much like the accounting method, but overcharges operations early to compensate for undercharges later.

### CHECK YOUR PROGRESS

13. What do you understand by worst-case complexity of algorithms?
14. Define space complexity.
15. What is time complexity?
16. What do you understand by a pseudocode?
17. What is amortized analysis?

## 1.12 SUMMARY

In this unit, you have learned that:

- An algorithm is a step-by-step procedure for performing some task in a finite amount of time. The five important properties (features) of algorithm are: finiteness, definitiveness, input, output and effectiveness.
- Testing of a program comprises two phases: (i) debugging and (ii) profiling. Debugging refers to the process of carrying out programs on sample data sets for finding out faulty results. Profiling refers to the process of executing a correct program on data sets and the measurement of the time and space it takes in computing the results.
- Algorithms can be classified into four categories: approximate algorithms, probabilistic algorithms, infinite algorithm and heuristic algorithms.
- The most commonly used design approaches for designing an algorithm include: incremental approach, divide and conquer approach, dynamic programming approach, greedy strategy, branch and bound algorithm approach, backtracking and randomized algorithms approach.
- The mathematical operation of the form $x^n$ is known as exponentiation. This involves two numbers, base and exponent. Here, in $x^n$, $x$ is the base and $n$ is the exponent.
- The methods which can be used to compute exponentiation fast are: squaring algorithm, Montgomery's ladder technique, sliding window method, Yao's method, and euclidean method.
- Linear search is the easiest and least efficient searching technique. In this technique, the given list of elements are scanned from the first one till either the required element is found or the list is exhausted. This technique is used in direct access media, such as magnetic tapes.
- Binary search is used to search for an element in a sorted list.
- The Fibonacci progression is a numeric progression such that $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1}+F_{n-2}$ for $n_2$. The Fibonacci search splits the given list of elements according to the Fibonacci progression unlike splitting in middle as in the binary search.
- The amount of memory an algorithm needs to run to completion is called its space complexity.
- The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.

## 1.13 KEY TERMS

- **Debugging:** It refers to the process of carrying out programs on sample data sets so as to check for faulty results.
- **Profiling:** It refers to the process of executing a correct program on data sets and the measurement of the time and space it takes in computing the results.
- **Randomized algorithm:** It refers to an algorithm whose input is determined by the values produced by a random number generator.
- **Average case:** It is the function defined by the average number of steps taken on any input of size *n*.
- **Space complexity:** It refers to the amount of memory an algorithm needs to run to completion.
- **Time complexity:** It refers to the amount of time the computer requires to run to completion.

## 1.14 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Efficiency as a function of input size can be measured in terms of the number of bits in an input number as well as the number of data elements (numbers, points).
2. Incremental approach is one of the simplest approaches to design algorithms**.** In this approach, whenever a new element is inserted into its appropriate place, the index is increased. One needs to start moving from the first step, executing each step till he reaches the end. Here, the problem is not split.
3. The following are the two main types of randomized algorithms:
   (i) Las Vegas algorithms
   (ii) Monte Carlo algorithms
4. Instructions include the following:
   - **Arithmetic:** Add, multiply, substract, floor, ceiling, divide
   - **Shift left and shift right**
   - **Data movement:** Assignment, load, copy, store
   - **Logical:** Comparison
   - **Control:** Conditional/unconditional branching, subroutine call, return
5. Primitive operations are low-level operations which are independent of the programming language. They can be identified in the pseudocode.
6. A flowchart refers to a graphical representation of a process which depicts inputs, outputs and units of activity. It represents the whole process at a high or detailed (depending on your use) level of observation. It serves as

an instruction manual or a tool to facilitate a detailed analysis and optimization of workflow as well as service delivery.

7. When a theoretical algorithm design is combined with the real-world data, it is called algorithm engineering.
8. This leads to two basic rules of exponentiation:
   (i) Any number to the power 1 is the same number.
   (ii) Any nonzero number to the power 0 is 1.
9. The following methods can be used to compute exponentiation fast:
   - Squaring algorithm
   - Montgomery's ladder technique
   - 2 *K*-ary method
   - Sliding window method
   - Yao's method
   - Euclidean method
10. In this technique, the given list of elements is scanned till either the required element is found or the list is exhausted. This technique is used in direct access media such as magnetic tapes.
11. Binary search is used to search for an element in a sorted list.
12. The big O notation (also known as big OH notation) provides a convenient way to compare the speed of algorithms. This is a mathematical notation used in the priori analysis. If an algorithm is said to have a computing time of O(g(n)), then it implies that if the algorithm is run on some computer on the same type of data put for increasing the values of n, the resulting times will always be less than same constant times |g(n)|.
13. Worst-case complexity of an algorithm refers to the measure of resources it requires to solve a problem in the worst case. Here, resources include, time of run and memory required. Worst-case indicates the maximum amount of resource required by the algorithm to solve the problem.
14. The space complexity of an algorithm indicates the quantity of temporary storage required for running the algorithm, i.e., the amount of memory needed by the algorithm to run to completion.
15. The time complexity of an algorithm may be defined as the amount of time the computer requires to run to completion.
16. A pseudocode is neither an algorithm nor a program. It is an art of expressing a program in simple English that parallels the forms of a computer language. It is basically useful for working out the logic of a program.
17. Amortized analysis means finding the average running time per operation over a worst-case sequence of operations.

## 1.15 QUESTIONS AND EXERCISES

NOTES

**Short-Answer Questions**

1. Why an algorithm needs to be verified before it gets analysed?
2. Write a short note on Greedy algorithms.
3. Write a short note on squaring algorithm which is used for computing exponentiation fast.
4. Write an algorithm for Fibonacci search.

**Long-Answer Questions**

1. Explain the different methods used to compute exponentiation fast.
2. Write a program to show the implementation of linear search to find a value in a vector or array.
3. Write a program to show the implementation of binary search to find an element in a sorted vector/array using recursion technique.
4. Write a program to show the implementation of Fibonacci search to find a string in a sorted vector/array.

## 1.16 FURTHER READING

Lipschutz, Seymour and Lipson Marc. *Schaum's Outline of Discrete Mathematics,* 3rd edition. New York: McGraw-Hill, 2007.

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms.* Hyderabad: Orient BlackSwan, 2008.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

Brassard, Gilles and Paul Bratley. *Fundamentals of Algorithms*. New Delhi: Prentice Hall of India, 1995.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. New Jersey: Pearson, 2006.

Baase, Sara and Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. New Jersey: Pearson, 2003.

Mott, J.L. *Discrete Mathematics for Computer Scientists*, 2nd edition. New Delhi: Prentice-Hall of India Pvt. Ltd., 2007.

Liu, C.L. *Elements of Discrete Mathematics*. New Delhi: Tata McGraw-Hill Publishing Company, 1977.

Rosen, Kenneth. *Discrete Mathematics and Its Applications*, 6th edition. New York: McGraw-Hill Higher Education, 2007.

# UNIT 2 GRAPH THEORY

NOTES

**Structure**

## 2.0 INTRODUCTION

In this unit, you will learn about the various features of graphs. A graph is a depiction in a diagrammatic format of a set of dots for the vertices, joined by lines or curves for the edges. Every graph has a diagram associated with it. This diagram is helpful in understanding the problems involved in the graph. In this unit you will learn about the various types of graphs and operations involving them. You will also learn the difference between a simple graph and pseudograph, and will also come to know that the degree of a vertex is the number of edges incident with that vertex and that a vertex with degree zero is called an isolated vertex. There are various types of graphs, such as complete graph, bipartite graph and subgraph. The edge connectivity of a graph is the minimum cardinality of a set of edges. This unit also deals with isomorphic and homeographic graphs.

## 2.1 UNIT OBJECTIVES

NOTES

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

- Understand the various types of graphs and their operations
- Describe the characteristics of the degree of a vertex
- Understand the functions of adjacent and incidence matrices
- Explain the various features of a path circuit
- Colour graphs and maps

## 2.2 GRAPHS: TYPES AND OPERATIONS

A graph $G$ is a triplet ($V(G)$, $E(G)$, $\theta_G$) consisting of a non-empty set $V(G)$ of vertices, a set $E(G)$ of edges and a function $\theta_G$ that is assigned to each edge and a subset $\{u, v\}$ of $V(G)$ ($u$, $v$ need not be distinct). If $e$ is an edge and $u$, $v$ are vertices such that $\theta_G(e) = uv$, then $e$ is a line (edge) between $u$ and $v$; the vertices $u$ and $v$ are the end points of the edge $e$.

For example, (*i*) $G = (V(G), E(G), \theta_G)$

Where, $V(G) = \{v_1, v_2, v_3, v_4\}$

$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

$\theta_G(e_1) = \{v_1v_2\}$, $\theta_G(e_2) = \{v_2v_2\}$, $\theta_G(e_3) = \{v_2v_3\}$,

$\theta_G(e_4) = \{v_1v_3\}$, $\theta_G(e_5) = \{v_4v_5\}$ and $\theta_G(e_6) = \{v_1v_4\}$

(*ii*) $G = (V(G), E(G), \theta_G)$

Where, $V(G) = \{v_1, v_2, v_3\}$, $E(G) = \{e_1, e_2, e_3\}$,

$\theta_G(e_1) = \{v_1v_2\}$, $\theta_G(e_2) = \{v_2v_3\}$; $\theta_G(e_3) = \{v_3v_1\}$

Every graph has a diagram associated with it. These diagrams are useful for understanding problems involved in the graph. In the pictorial representation, vertices are represented by small circles and edges by lines whenever the corresponding pair of vertices forms an edge.

The pictorial representation of examples (*i*) and (*ii*) are shown in Figure 2.1.

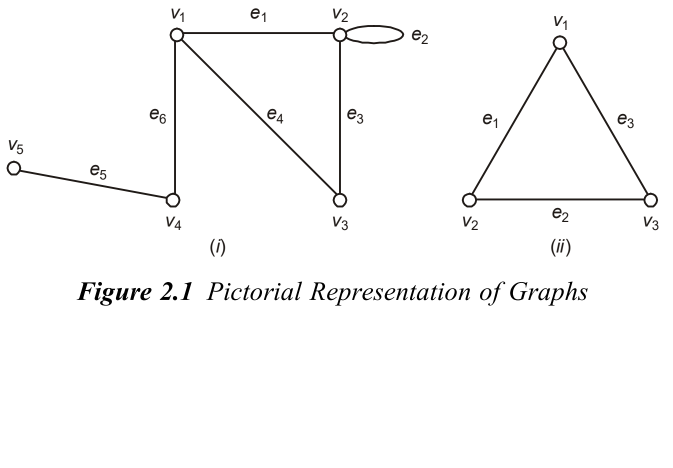


***Figure 2.1*** *Pictorial Representation of Graphs*

***Notes:***

1. In example (*i*), $e_2$ joins the vertex $v_2$ to itself. Such an edge is called self loop.
2. If there is more than one edge between a pair of vertices in a graph, then these edges are called parallel edges.
3. Hereafter the graph $G = (V, E)$ will be denoted for simplicity.
4. A graph which consists of parallel edges is called a multigraph.

**Simple Graph:** A graph with no self loops and parallel edges is called a simple graph.

**Pseudograph:** A graph with self loops and parallel edges is called a pseudograph (see Figure 2.2).

***Note:*** Every simple graph and every multigraph is a pseudograph, but the converse is not true.

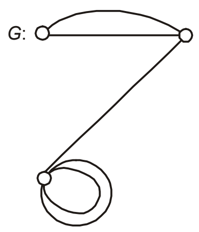


***Figure 2.2*** *A Pseudograph*

The above graph *G* is neither a simple graph nor a multigraph.

Following are some of the types of graphs commonly used:

## 2.2.1 Bipartite Graphs

A simple graph *G* is called bipartite if its vertex set *V* can be partitioned into two disjoint non-empty sets $V_1$ and $V_2$ in such a way that every edge in the graph connects a vertex in $V_1$ and a vertex in $V_2$. Note that no edge in *G* in Figure 2.3 connects either two vertices in $V_1$ or two vertices in $V_2$.

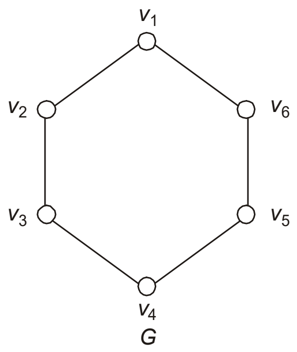


***Figure 2.3*** *A Bipartite Graph*

For example, $G$ is bipartite, because its vertex set $v = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ is partitioned into two non-empty sets $V_1 = \{v_1, v_3, v_5\}$ and $V_2 = \{v_2, v_4, v_6\}$. Also, every edge in $G$ connects a vertex in $V_1$ and a vertex in $V_2$.

**Complete Bipartite Graph**

The complete bipartite graph $k_{m,\,n}$ is the graph that has its vertex set partitioned into two non-empty subsets of $m$ and $n$ vertices, respectively. There is an edge between two vertices, if one vertex is in the first subset then the other vertex is in the second subset.

Figure 2.4 has examples of complete bipartite graphs.

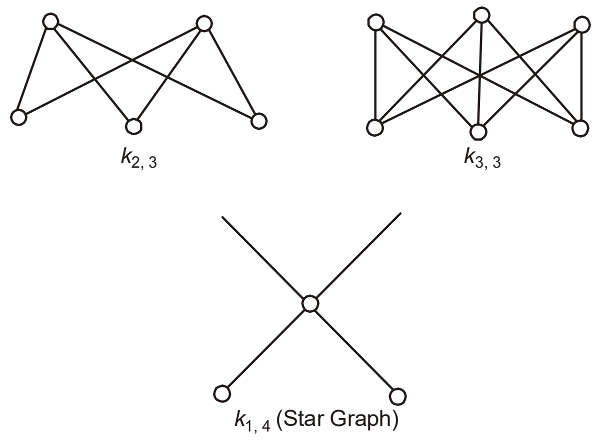


***Figure 2.4*** *Complete Bipartite Graphs*

### 2.2.2 Subgraph

A graph $H = (V(H), E(H))$ is called a subgraph of a graph $G = (V(G), E(G))$ if ($a$) $V(H) \leq V(G)$ and ($b$) $E(H) \leq E(G)$.

A subgraph $H$ of a graph $G$ is called a spanning subgraph if $V(H) = V(G)$.

Figure 2.5 shows examples of subgraphs:

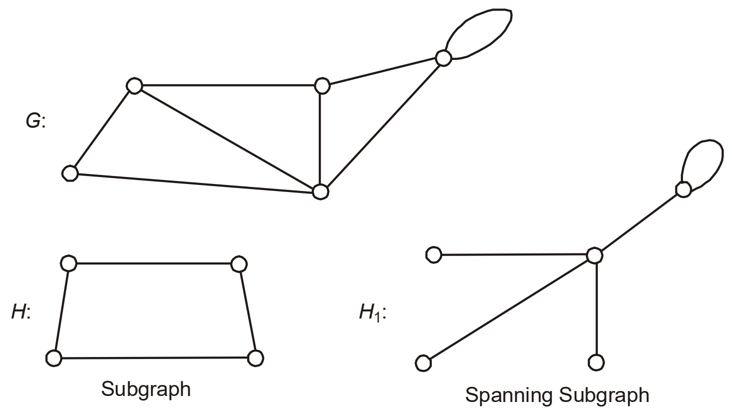


***Figure 2.5*** *Subgraphs*

### 2.2.3 Distance in a Graph

For a non-trial graph $G$ and a pair $u$, $v$ of vertices of $G$, the distance $d_G$ $(u - v)$ is defined as the length of a shortest $(u - v)$ path in $G$ (if such path exists). If $G$ contains no $(u - v)$ path, then one defines $d_G$ $(u - v) = \infty$.

Examples of Figure 2.6 illustrate the distance in a graph.

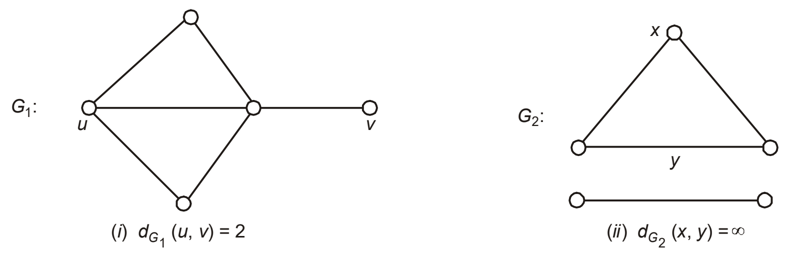


***Figure 2.6*** *Distance in a Graph*

$G$ is a connected graph and $v$ is an arbitrary vertex in $G$. Then, following important terms are defined as follows:

(i) The eccentricity of $v$ is defined as the length of the longest path in $G$ starting from vertex $u$ and is denoted by $e(v)$. $e(v) = \max \{d(u, v): u \in v(G)\}$.

(ii) The diameter of $G$ is defined as the maximum eccentricity among all the vertices of $G$, i.e., diam $(G) = \max \{e(v): v \in V(G)\}$.

(iii) The radius of $G$ is defined as the minimum eccentricity among all the vertices of $G$, i.e., rad $(G) = \min \{e(v): v \in V(G)\}$.

(iv) The centre of $G$ is defined as the set of vertices having minimum eccentricity among all the vertices of $G$, i.e., cent $(G) = \{v \in V(G): e(v) = \text{rad}(G)\}$.

***Notes:***

1. rad $(G) \leq$ diam $(G) \leq 2$ rad $(G)$, $G$ is a graph.
2. The median of a connected graph $G$ is defined as the set of vertices having minimum distance.

### 2.2.4 Cut-Vertices and Cut-Edges

A vertex $v$ in a graph $G$ is said to be a cut-vertex if $\omega(G - v) > \omega(G)$, where $\omega(G)$ is the component of $G$ and a component is a maximal connected subgraph of $G$, i.e., a vertex $v$ of a connected graph is a cut-vertex, iff $(G - v)$ is disconnected (Figure 2.7).

NOTES

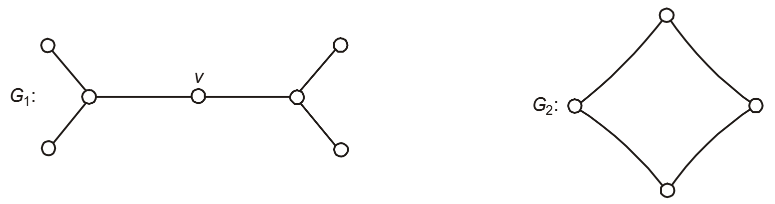


*Figure 2.7 Cut-Vertices and Cut-Edges*

$G_1$ contains one cut-vertex $v$ and $G_2$ contains no cut-vertices (See Figure 2.7).

**Theorem 2.1:** A vertex $v$ in a connected graph $G$ is a cut-vertex iff vertices $u$ and $w$ exist (both are different from $v$) in such a way that every path connecting $u$ and $w$ contains $v$.

**Proof:** Let $G$ be a connected graph and $v$ be a cut-vertex.

**Claim:** There exist vertices $u$ and $w$ in such a way that every path between $u$ and $w$ contains $v$.

Since $v$ is a cut-vertex, $(G - v)$ is disconnected and $(G - v)$ contains two components say $G_1$ and $G_2$. Let $u$ and $w$ be the vertices of $G_1$ and $G_2$, respectively. Clearly there is no $(u - w)$ path in $(G - v)$. Hence, every path connecting $u$ and $w$ must contain $v$.

Conversely, lets assume that there exist vertices $u$ and $w$ in such a way that every $(u - w)$ path contains $v$.

**Claim:** $v$ is a cut-vertex.

Suppose $v$ is not a cut-vertex. Then, $(G - v)$ is connected. Since $u$, $w$ are vertices in $G - v$, there is a path between $u$ and $w$ in $G - v$, which does not contain the vertex $v$. This is a contradiction. Hence, $v$ is a cut-vertex.

**Cut-edge:** An edge $e$ in a graph $G$ is said to be a cut-edge, if $(G - e)$ is disconnected (see Figure 2.8).

For example,

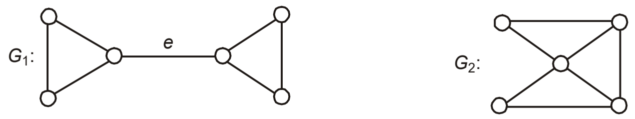


*Figure 2.8 $G_1$ containing One Cut-Edge and $G_2$ with no Cut-edge*

As in cut-vertex, a similar result can be furnished.

**Theorem 2.2:** An edge $e$ in a connected graph $G$ is a cut-edge iff there exists vertices $u$ and $w$ such that every path connecting $u$ and $w$ must contain the edge '$e$'.

**Proof:** Let $G$ be a connected graph and $e$ be a cut-edge.

**Claim:** There exist vertices $u$ and $w$ such that every $(u - w)$ path must contain the edge $e$, since $e$ is a cut-edge in $G$, $(G - e)$ is disconnected and $(G - e)$ contains atleast two components say $G_1$ and $G_2$.

Let $u$ and $w$ be the vertices respectively in $G_1$ and $G_2$. Thus, there is no path between $u$ and $w$ in $(G - 2)$. Hence, every path connecting $u$ and $w$ must contain the edge $e$.

Conversely, suppose that there exist vertices $u$ and $w$ such that every path connecting $u$ and $w$ must contain the edge $e$.

**Claim:** $e$ is a cut-edge.

Suppose $e$ is not a cut-edge. Then $(G - e)$ is connected and hence, $e$ is some circuit in $G$. Therefore, there exists a path connecting $u$ and $w$, which does not contain the edge $e$. This is a contradiction. Hence, $e$ is a cut-edge.

### 2.2.5 Graph Connectivity

In this sub-section, the structure of graphs will be studied. A walk in a graph $G$ is an alternating sequence.

$W: v_0, e_1, v_1, e_2, ..., v_{n-1}, e_n, v_n$ $(n \geq 0)$ of vertices and edges, beginning and ending with vertices, such that $e_i = v_{i-1} v_i$, $i = 1, 2, ..., n$. It is denoted by $(v_0 - v_n)$ walk. The number of edges (not necessarily distinct) is called the length of walk. In graph $G$, $u$, $e_1$, $v$, $e_2$, $w$, $e_6$, $x$, $e_4$, $u$ is a walk of length 4. Figure 2.9 illustrates the path and walk in a graph.

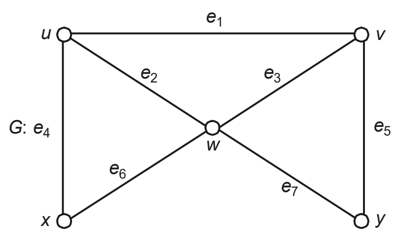


***Figure 2.9*** *Path and Walk in a Graph*

A trail is a *walk* in which no edge is repeated and a *path* is a trail in which no vertex is repeated. Thus, a path is a trail, but not every trail is a *path*. In the above graph $G$, $x$, $e_6$, $w$, $e_3$, $v$, $e_1$, $u$, $e_2$, $w$, $e_7$, $y$ is a *trail* that is not a path, and $u$, $e_4$, $x$, $e_6$, $w$, $e_7$, $y$ $e_5$, $v$ is a path.

**Problem 1:** Every $(u - v)$ walk in a graph contains a $(u - v)$ path.

**Proof:** Let $W$ be a $(u - v)$ walk in a graph $G$. If $u = v$, then $w$ is the trail path, i.e., walk of length zero.

Suppose $u \neq v$ and $W: u = u_0, u_1, u_2, ..., u_n = v$. If no vertex of $G$ appears in $W$ more than once, then $w$ itself is a $(u - v)$ path. Otherwise, there are vertices of $G$ that occur in $w$ twice or more. Let $i$ and $j$ be distinct positive integers such that $i < j$ with $ui = uj$. Then say $u_i, u_{i+1}, ..., u_{j-2}, u_{j-1}$ are removed from $w$ and the resulting sequence is $(u - v)$ walk $w_1$ whose length is less than that of $w$. By induction hypothesis, this $w_1$ contains a $(u - v)$ path and hence $w$ has a $(u - v)$ path. If no vertex of $G$ appears more than once in $w_1$, then $w_1$ is a $(u - v)$ path. If not, apply the procedure, until you get a $(u - v)$ path.

NOTES

**Cycle:** A cycle is a walk. $v_0, v_1, ..., v_n$ is a walk in which $n \geq 3$, $v_0 = v_n$ and the '$n$'-vertices $v_1, v_2, ..., v_n$ are distinct. It is said that a $(u - v)$ walk is closed if $u = v$ and open if $u \neq v$.

**Connection:** Let $u$ and $v$ be vertices in a graph $G$. You say that $u$ is connected to $v$ if $G$ contains a $(u - v)$ path. The graph $G$ is connected, if $u$ is connected to $v$ for every pair and $u$, $v$ are vertices of $G$.

**Disconnection:** A graph $G$ is disconnected, if there exists two vertices $u$ and $v$ for which there is no $(u - v)$ path.

**Component:** A subgraph $H$ of a graph $G$ is called a component of $G$, if $H$ is a maximal connected subgraph of $G$ and component is denoted by $\omega(G)$.

***Note:*** If $\omega(G) > 1$, then $G$ is disconnected (see Figure 2.10)

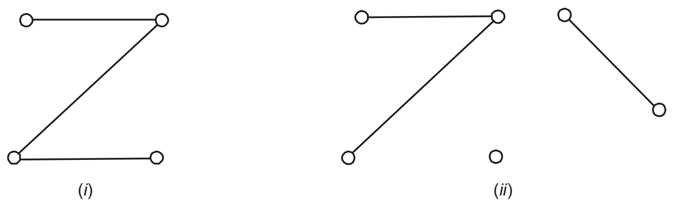


***Figure 2.10*** *Connected and Disconnected Graph*

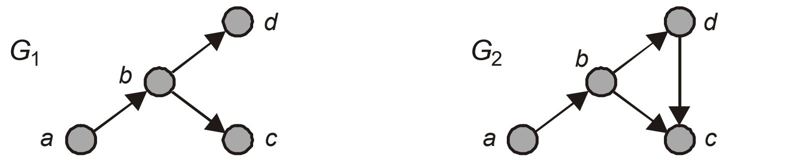
Graph ($i$) is connected and ($ii$) is disconnected.

Note that graph ($ii$) has 3 components.

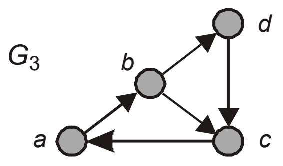
**Connectedness in a Directed Graph**

- **Strongly Connected:** A directed graph is strongly connected if there is a path from $u$ to $v$ and $v$ to $u$, whenever $u$ and $v$ are vertices in the graph.
- **Weakly Connected:** A directed graph is weakly connected, if there is a path between any two vertices in the underlying undirected graph.
- **Unilaterally Connected:** A directed graph is said to be unilaterally connected, if in the two vertices $u$ and $v$, there exists a directed path either from $u$ to $v$ or from $v$ to $u$. (see Figure 2.11)

For example,



*Weakly Connected Directed Graph* *Unilaterally Connected Directed Graph*



*Strongly Connected Directed Graph*

***Figure 2.11*** *Connectedness in Directed Graph*

$G_1$ is weakly connected; $G_2$ unilaterally connected and $G_3$ is strongly connected.

### 2.2.6 Isomorphic Graphs

Two graphs $G$ and $H$ are said to be isomorphic if there exist bijections

$\psi : V(G) \to V(H)$ and $\phi$: $E(G) \to E(H)$ such that iff $\theta_G(e) = uv$ iff $\theta_H(\phi(e)) = \psi(u)\,\psi(v)$.

Such a pair of mappings $(\psi, \phi)$ is called an isomorphism between $G$ and $H$ and is written as $G \cong H$.

In other words, two simple graphs $G$ and $H$ are isomorphic if there is a bijection $\psi : V(G) \to V(H)$ such that $uv \in E(G)$ iff $\psi(u)\,\psi(v) \in E(H)$.

Figure 2.12 shows examples of isomorphic graphs.

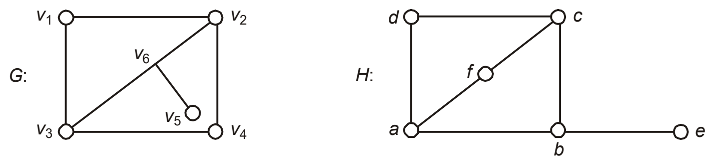


***Figure 2.12*** *Isomorphic Graphs*

Here $G$ and $H$ are isomorphic.

The correspondence which gives isomorphism between $G$ and $H$ is as follows:

$$v_1\,v_2 \in E(G) \Leftrightarrow dc = \psi(v_1)\,\psi(v_2) \in E(H)$$
$$v_1\,v_3 \in E(G) \Leftrightarrow da = \psi(v_1)\,\psi(v_3) \in E(H)$$
$$v_3\,v_6 \in E(G) \Leftrightarrow ab = \psi(v_3)\,\psi(v_6) \in E(H)$$
$$v_6\,v_5 \in E(G) \Leftrightarrow be = \psi(v_6)\,\psi(v_5) \in E(H)$$
$$v_3\,v_4 \in E(G) \Leftrightarrow af = \psi(v_3)\,\psi(v_4) \in E(H)$$

$$v_6 v_2 \in E(G) \Leftrightarrow bc = \psi(v_6)\ \psi(v_2) \in E(H)$$

$$v_4 v_2 \in E(G) \Leftrightarrow fc = \psi(v_4)\ \psi(v_2) \in E(H)$$

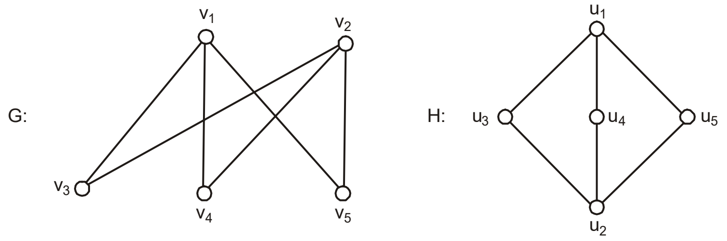
$\therefore$ $$G \cong H$$

NOTES

***Notes:***

1. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if a one-to-one correspondence $\phi$ exists from $V_1$ to $V_2$ such that $u$ and $v$ are adjacent in $G_1$ iff $\phi(u)$ and $\phi(v)$ are adjacent to $G_2$.
2. If $G \cong H$, then degrees of corresponding vertices are equal.

**Example 2.1:** Prove that the following graphs $G$ and $H$ are non-isomorphic.



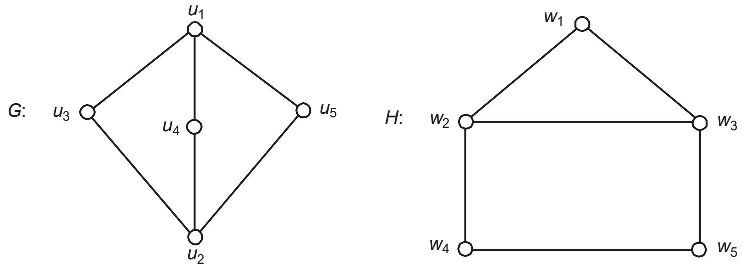
**Solution:** Clearly $G$ and $H$ are isomorphic.

In $G$, $V_1$ is adjacent to the vertices $V_3$, $V_4$, $V_5$; $V_2$ is adjacent to the vertices $V_3$, $V_4$, $V_5$.

In $H$, $u_1$ is adjacent to $u_3$, $u_4$, $u_5$ and $u_2$ is adjacent to $u_3$, $u_4$, $u_5$.

Here, the function defined by $\phi(v_i) = u_i$, $1 \le i \le 5$ gives the isomorphism.

**Example 2.2:** Prove that the following graphs $G$ and $H$ are non-isomorphic.



**Solution:** Clearly $G$ and $H$ are non-isomorphic graphs.

In $G$, these two vertices ($u_1$ and $u_2$) are adjacent with three other vertices ($u_3$, $u_4$, $u_5$) whereas in $H$, the vertex $w_2$ is adjacent to $w_1$, $w_2$, $w_4$ and the vertex $w_3$ is adjacent to $w_1$, $w_2$ and $w_5$ and vertices $w_2$ and $w_3$ are adjacent to each other.

In $G$, $u_1$ and $u_2$ are non-adjacent. Hence, $G$ is not isomorphic to $H$.

***Note:*** From the above example, it is clear that two graphs are isomorphic if they have same number of vertices and same number of edges and the degrees of the corresponding vertices are equal, but the converse is not true.

## 2.2.7 Homeographic Graphs

Two graphs $G_1$ and $G_2$ are homeomorphic if an isomorphism is found from any subdivision of $G_1$ to any subdivision of $G_2$.

Subdivision of a graph is another graph that results from subdivision of edges in that graph. Let there be an edge *e* having $\{u, \text{v}\}$ as endpoints. Subdivision of edge *e* with these endpoints will yield a graph that contains another vertex *w* as a new vertex with an edge set that replaces *e* by two new edges having endpoints $\{u,w\}$ and $\{w,v\}$. Lets take an example of a graph as shown in Figure 2.13.

There is an edge connecting two endpoints $\{u,v\}$.

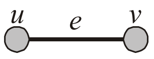


***Figure 2.13*** *An Edge Connecting Two Endpoints*

This graph may have subdivision as two edges, $e_1$ and $e_2$, with a new vertex *w*.

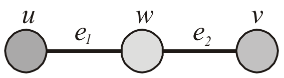


***Figure 2.14*** *The Edges of a Graph*

Reverse of this operation smoothens out a vertex *w* connecting a pair of edges $(e_1, e_2)$ and removes these edges that contain vertex *w* replacing these with a new edge connecting other endpoints of the pair. In Figure 2.14 it is emphasized that only 2-valent vertices can be smoothed.

This can be understood in Figure 2.15. Let there be a simple connected graph having two edges, $e_1$ joining vertices $\{u,w\}$ and $e_2$ joining vertices $\{w,v\}$.

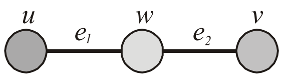


***Figure 2.15*** *A Simple Connected Graph with Two Edges Joining Vertices*

There is common vertex *w* that might be smoothed away. If done so this results in a situation as shown in Figure 2.16.

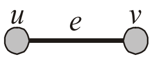


***Figure 2.16*** *Smoothing Away of the Common Vertex*

Determining whether for graphs $G_1$ and $G_2$, $G_2$ is homeomorphic to a subgraph of $G_1$, it is a problem that is NP-complete.

### Barycentric Subdivision

A subdivision of this type is a special subdivision that subdivides every edge of a graph. Such a subdivision results in a bipartite graph and procedure can be repeated in a way that *n*th barycentric subdivision is the barycentric subdivision of *n*-1th barycentric subdivision of the graph. Second subdivision of this type results in a simple graph.

### Embedding on a Surface

Subdivision of a graph preserves planarity. Kuratowski's theorem states that 'a finite graph is planar if and only if it contains no subgraph homeomorphic to the complete graph on five vertices or complete bipartite graph on six vertices, three of which connect to each of the other three'.

A complete graph is denoted as $K_5$ and a complete bipartite graph of six vertices in which three vertices are connected to another three vertices is denoted as $K_{3,3}$. If a graph is homeomorphic to $K_5$ or $K_{3,3}$ it is known as Kuratowski subgraph.

A generalization, that follows from Robertson–Seymour theorem, asserts that for each integer $g$, a finite obstruction set $L(g) = \{G_i^{(g)}\}$ of graphs is there, such that a graph G can be embeded on a surface of genus $g$, iff G does not contain any homeomorphic copy of any of the $G_i^{(g)}$. For example, the finite obstruction set $L(0) = \{K_5, K_{3,3}\}$ contains the Kuratowski subgraphs.

For Example,

In Figure 2.17, graph $G_1$ and graph $G_2$ are homeomorphic.

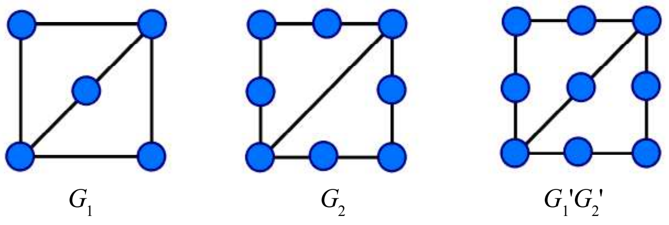


***Figure 2.17*** *Homoeomorphic Graphs*

If $G_1$ is the graph created by subdivision of the outer edges of $G_1$ and $G_2'$ is the graph resulting from subdivision of inner edge of $G_2$, then $G_1'$ and $G_2'$ have similarity in drawing as shown in Figure 2.17 and hence, $G_1'$ and $G_2'$ have isomorphism which leads to the fact that $G_1$ and $G_2$ are homoeomorphic.

### 2.2.8 Cut-Sets and Connectivity of Graphs

Let $G$ be a connected graph. Let us recollect the definition of cut-edge (bridge) and cut-vertex. If $G$ contains an edge $e$ such that $G–e$ is disconnected, then $e$ is a bridge of $G$. Further, if $G$ contains a vertex $v$ such that $G–v$ is disconnected, then $v$ is a cut-vertex of $G$.

**1. Edge cut-set:** A subset $S$ of the edge set of a connected graph $G$ is called an edge cut-set or cut-set of $G$ if,

(*i*) $G - S$ is disconnected.

(*ii*) $G - S_1$ is connected for every proper subset $S_1$ of $S$.

**2. Vertex cut-set:** A subset $u$ of the vertex set of $G$ is called a vertex cut-set if,

(*i*) $G - u$ is disconnected.

(*ii*) $G - u_1$ is connected for every proper subset $u_1$ of $u$.

For example,

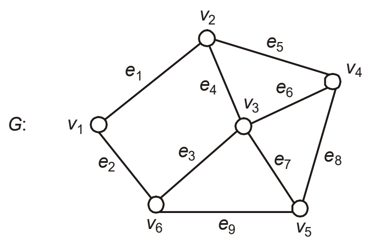


***Figure 2.18*** *Cut-Set*

(*i*) $S = \{e_1, e_4, e_6, e_8\}$ is a cut-set (see Figure 2.18).

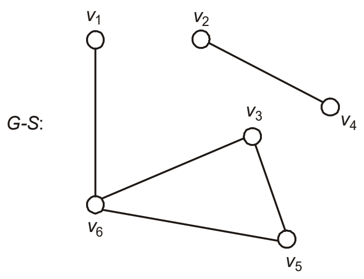


***Figure 2.19*** *Vertex Cut-Set*

(*ii*) $u = \{v_1, v_3, v_5\}$ is a vertex-cut-set (see Figures 2.19 and 2.20).

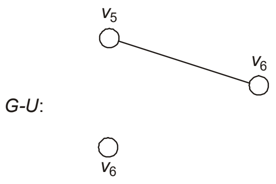


***Figure 2.20***

***Note:*** For a connected graph, there may be more than one cut-set.

For example, consider the graph $G$ in Figure 2.20. Some cut-sets of $G$ are:

$$S_1 = \{e_1, e_4, e_6, e_8\}$$

$$S_2 = \{e_1, e_2\}, S_3 = \{e_1, e_3, e_9\}$$

As s result one is forced to introduce two more parameters for graphs, viz. edge-connectivity $\lambda(G)$ and vertex connectivity $k(G)$.

**1. Edge Connectivity:** The edge connectivity l($G$) of a graph is the minimum cardinality of a set $S$ of edges of $G$ such that $G - S$ is disconnected, i.e., the edge (line) connectivity of a connected graph is the number of edges in a minimum cut-set in the graph (see Figure 2.21).

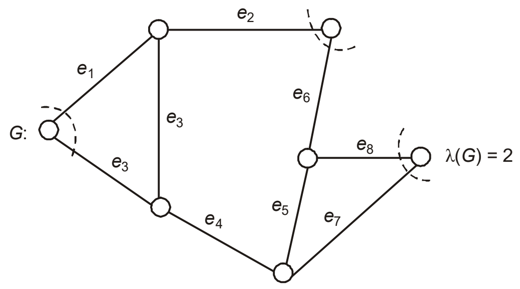


***Figure 2.21*** *Edge Connectivity*

***Notes:***

1. If $G$ is a tree then $\lambda(G) = 1$.
2. $G$ has $\lambda(G) = 0$ iff $G$ is disconnected or trivial.

**2. Vertex Connectivity:** The vertex connectivity K(G) of a graph $G$ is the minimum number of vertices whose deletion makes $G$ a disconnected or trivial graph, i.e., the number of vertices in a minimum vertex cut is called the connectivity of the graph (Figure 2.22).

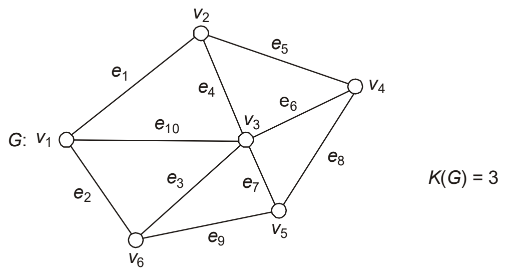


***Figure 2.22*** *Vertex Connectivity*

**Problem 2:** For every graph $G$, $K(G) \leq \lambda(G) \leq \delta(G)$.

**Proof:** Let $v$ be a vertex of $G$ with a minimum degree, i.e., $d(v) = \delta(G)$. Removing $\delta(G)$ edges of $G$ incident with $v$ produces a graph $G_1$, in which $v$ is isolated. Clearly $G$ is disconnected or trivial.

$\therefore \quad \lambda(G) \leq \delta(G)$ ...(2.1)

**Claim:** $K(G) \leq \delta(G)$

If, $\delta(G) = 0$ then $G$ is disconnected.

$\therefore \quad K(G) = 0$.

If $\delta(G) = 1$, then $G$ is a connected graph containing a cut-edge (bridge). Therefore either $G$ is isomorphic to $K_2$ or $G$ is a connected graph having atleast one cut-vertex.

$\therefore$ In both cases, $K(G) = 1$.

Now, let us assume that $\lambda(G) \geq 2$. Let $S$ be a cut-set of $G$ with $\lambda(G)$ edges and $e = xy$ be an edge in $S$. If the edges of $S - \{e\}$ are deleted from $G$, the resulting subgraph $H_1$ is connected and contains $e$ as a cut-edge. Now select an incident vertex different from $x$ and $y$ for each and every edge in $S - \{e\}$. Remove these vertices from $H_1$, the resulting subgraph $H_2$ is disconnected, then

$$K(G) \leq \lambda(G) - 1 < \lambda(G)$$

Suppose the subgraph $H_2$ is connected, then $H_2$ is isomorphic to $K_2$ or the subgraph $H_2$ has a cut-vertex, since $H_2$ is an induced subgraph of $H_1$. In any case, there exists a vertex of $H_2$ whose removal results in a disconnected graph. Therefore,

$$K(G) \leq \lambda(G). \qquad ...(2.2)$$

From equations (2.1) and (2.2), $K(G) \leq \lambda(G) \leq \delta(G)$

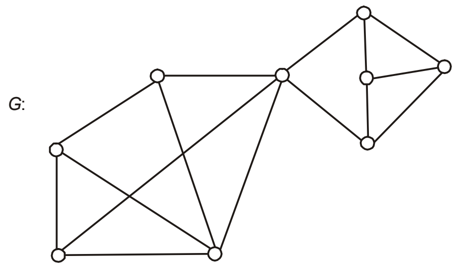


*Figure 2.23*

In Figure 2.23 $K(G) = 1$, $\lambda(G) = 3$ and $\delta(G) = 3$.

***n*-edge Connected:** A graph $G$ is $n$-edge connected ($n \geq 1$) if $\lambda(G) \geq n$ and $G$ is $n$-connected if $K(G) \geq n$.

## 2.2.9 Operations on Graphs

(*i*) The union of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and an edge set $E_1 \cup E_2$ and is denoted by $G_1 \cup G_2$ (see Figure 2.24).

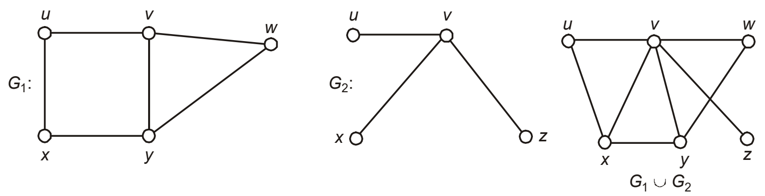


*Figure 2.24*

(*ii*) The intersection of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cap V_2$ and an edge set $E_1 \cap E_2$ and is denoted by $G_1 \cap G_2$ (see Figure 2.25). You need to remember that for $G_1 \cap G_2$, $V_1 \cap V_2$ is always non-empty.

For example,

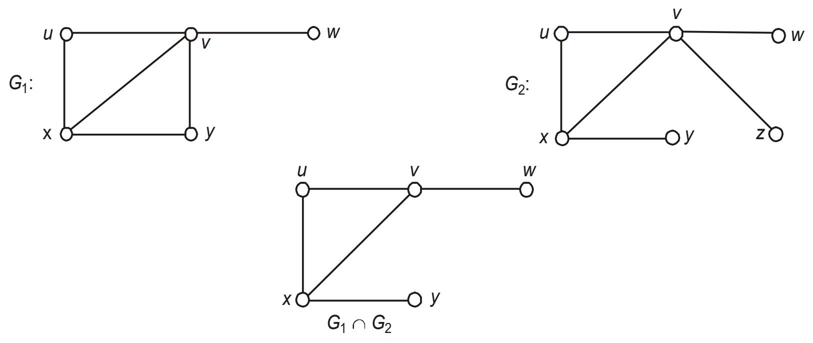


***Figure 2.25***

NOTES

(*iii*) The ring sum of two graphs $G_1$ and $G_2$ is a graph consisting of the vertex set $V_1 \cup V_2$ and of edges that are either in $G_1$ or in $G_2$, but not in both and is denoted by $G_1 \oplus G_2$, i.e.,

$$G_1 = (V_1, E_1);\ G_2 = (V_2, E_2)$$

$$\text{Then, } G_1 \oplus G_2 = (V_1 \cup V_2, E_1 \Delta E_2)$$

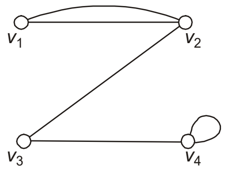
Where Δ is the symmetric difference.

**CHECK YOUR PROGRESS**

1. How can a graph be represented diagrammatically?
2. What is a simple graph?
3. What is a pseudograph?
4. Name the various types of graphs.
5. What does the edge connectivity of a graph mean?
6. What does the vertex connectivity of a graph mean?
7. What are isomorphic graphs?
8. What is a subdivision of graph?

## 2.3 DEGREE OF VERTEX

The degree of a vertex *v* is the number of edges incident with that vertex. In other words, the degree of a vertex is the number of edges having that vertex as an end point, and is denoted by $d(v)$ Figure 2.26.



Here, $d(v_1) = 2$
$d(v_2) = 3$
$d(v_3) = 2$
$d(v_4) = 3$

***Figure 2.26** Degree of a Vertex*

A loop contributes 2 to the degree of vertex.

**Isolated Vertex:** A vertex with degree zero is called an isolated vertex.

**Pendant Vertex:** A vertex with degree one is called a pendant vertex.

**Adjacent Vertices:** A pair of vertices that determine an edge are called adjacent vertices.

***Note:*** A vertex is even or odd based on its degree being even or odd.

**Example 2.3:** Let $G$ be a simple graph with $n$ vertices. Prove that the number of edges $E(G)$ is at most ${}^nC_2$.

**Solution:** Let $G = (V(G), E(G), \theta_G)$ be a simple graph with $|V(G)| = n$.

Since, $\theta_G$ assigns to each edge, a 2-element subset $\{u, v\}$ of $V(G)$, there are at most ${}^nC_2$ 2-element subsets.

Hence, $$E(G) \leq \frac{n(n-1)}{2}$$

**Theorem 2.3:** Let $G$ be a graph with $n$ vertices and $e$ edges. Then,

$$\sum_{i=1}^{n} d(v_i) = 2e$$

**Proof:** Let $G$ be a graph with $n$ vertices and $e$ edges.

Since, every edge contributes degree 2 to this sum, so $\sum_{i=1}^{n} d(v_i) = 2e$.

**Theorem 2.4:** In a graph $G$, the number of odd vertices is an even number.

**Proof:** Let $G$ be a graph with $n$ vertices and $e$ edges.

By Theorem 2.1, you have:

$$\sum_{i=1}^{n} d(v_i) = 2e = \text{Even number} \qquad ...(2.3)$$

Among $n$ vertices, some are even vertices and some are odd vertices. Let $V_e$ and $V_0$ be the even and odd vertices respectively.

Now equation (2.3) can be written as:

$$\sum_{v \in V_e}^{n} d(v) + \sum_{v \in V_0} d(v) = \text{Even number}$$

$$\therefore \qquad \sum_{v \in V_0} d(v) = \text{Even number} - \sum_{v \in V_e} d(v) \qquad ...(2.4)$$

Since every term in the right side of equation (2.4) is even, the sum on the left side must contain an even number of terms, i.e., the number of odd vertices in $G$ is even.

**Minimum and Maximum Degrees:** Let $G$ be a graph. The minimum and maximum degrees of $G$ are $\delta(G)$ and $\Delta(G)$, respectively and are given as:

$$\delta(G) = \min \{d(v);\ v \in V(G)\}$$

and, $$\Delta(G) = \max \{d(v);\ v \in V(G)\}$$

***k*-Regular:** A graph $G$ is $k$-regular or regular of degree $k$, if every vertex of $G$ has degree $k$.

**Complete Graph:** A simple graph in which each pair of distinct vertices is joined by an edge is called a complete graph. A complete graph on $n$ vertices is denoted by $k_n$.

Figure 2.27 are the examples of complete graphs on 2 and 4 vertices, respectively.

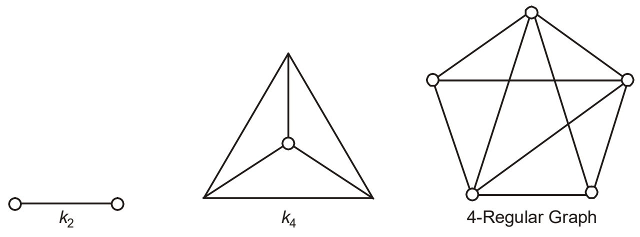


***Figure 2.27*** *Complete Graphs*

***Notes:***

1. Every complete graph $k_n$ is a $(n-1)$ regular graph.
2. There is no 1-regular or 3-regular graphs with 5 vertices, since no graph has an odd number of vertices.

**Complement of a graph:** The complement $\overline{G}$ of a graph $G$ is that graph with $V(G) = V(\overline{G})$ and such that $uv$ is an edge of $\overline{G}$ if and only if $uv$ is not an edge of $G$.

Figure 2.28 shows examples of complement of a graph.

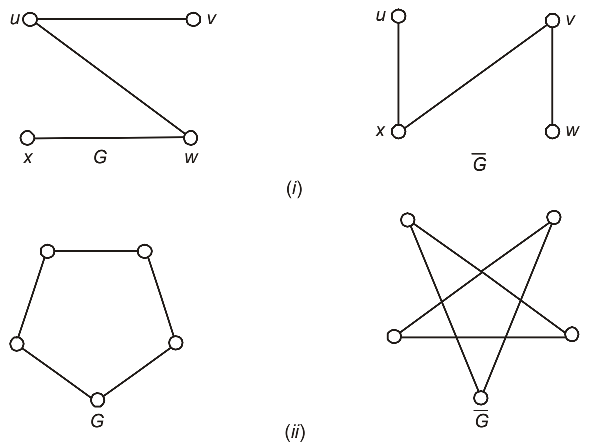


***Figure 2.28*** *Complement of a Graph*

There are also some useful terminologies for graphs with directed edges.

**Graphs with directed edges:** When $(u, v)$ is an edge of the graph $G$ with directed edges, $u$ is said to be adjacent to $v$ and $v$ is said to be adjacent from $u$. The vertex $u$ is called the initial vertex of $(u, v)$ and $v$ is called the terminal or end vertex of the edge $(u, v)$.

Figure 2.29 shows examples of graphs and directed edges.

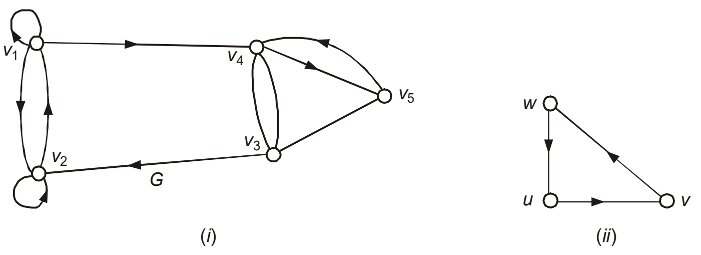
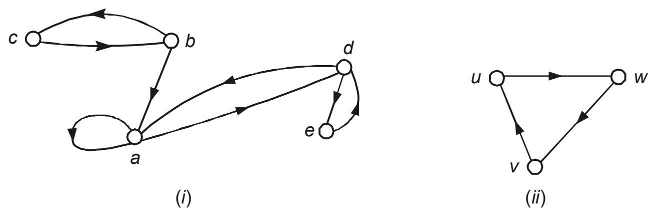


***Figure 2.29*** *Graphs with Directed Edges*

**In-degree and out-degree:** In a graph with directed edges, the in-degree of a vertex $v$ denoted by $d^{-}(v)$ is the number of edges with $v$ as their terminal vertex. The out-degree of $v$ denoted by $d^{+}(v)$ is the number of edges with $v$ as their initial vertex.

***Note:*** Self loop at a vertex contributes 1 to both in-degree and out-degree of this vertex.

**Example 2.4:** Find the in-degree and out-degree of the following graphs.



**Solution:**

(*i*) $d^{-}(a) = 3$; $d^{-}(b) = 1$; $d^{-}(c) = 1$; $d^{-}(d) = 2$; and $d^{-}(e) = 1$

$d^{+}(a) = 2$; $d^{+}(b) = 2$; $d^{+}(c) = 1$; $d^{+}(d) = 2$; and $d^{+}(e) = 1$

(*ii*) $d^{-}(u) = 1$; $d^{-}(v) = 1$; $d^{-}(w) = 1$ and

$d^{+}(u) = 1$; $d^{+}(v) = 1$; $d^{+}(w) = 1$

***Notes:***

1. Let $G = (V, E)$ be a graph with directed edges. Then $\sum_{v\in V} d^{-}(v) = \sum_{v\in V} d^{+}(v) = e$.
2. By ignoring directions of edges in a graph with directed edges, you will get an undirected graph. Such graphs are called underlying undirected graphs.

## 2.4 ADJACENT AND INCIDENCE MATRICES

To any graph $G$, there corresponds a $V \times E$ matrix called the incidence matrix of $G$ and is denoted by $I(G) = [a_{ij}]_{V\times E}$, where

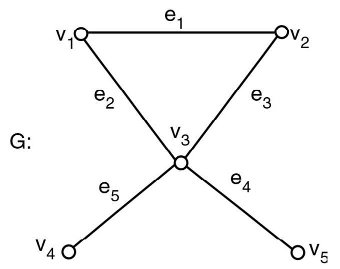
$$a_{ij} = \begin{cases} 1, \text{ if } j\text{th edge is incident with } i\text{th vertex} \\ 0, \text{ otherwise} \end{cases}$$

NOTES

One more matrix associated with graph $G$ is the adjacency matrix, $e$ is denoted by $A(G) = [b_{ij}]_{V \times V}$,

$$a_{ij} = \begin{cases} 1, \text{ if } j\text{th edge is incident with } i\text{th vertex} \\ 0, \text{ otherwise} \end{cases}$$

Some authors used to define $a_{ij}$ as the number of times (0, 1, and 2) $v_i$ and $e_j$ are incident ; $b_{ij}$ is the number of edges $v_i$ and $v_j$.
For example,



| | $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ |
|---|---|---|---|---|---|
| $v_1$ | 1 | 1 | 0 | 0 | 0 |
| $v_2$ | 1 | 0 | 1 | 0 | 0 |
| $v_3$ | 0 | 1 | 1 | 1 | 1 |
| $v_4$ | 0 | 0 | 0 | 0 | 1 |
| $v_5$ | 0 | 0 | 0 | 1 | 0 |

$I(G)$, incidence matrix of $G$

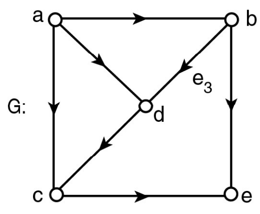
| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| $v_1$ | 0 | 1 | 1 | 0 | 0 |
| $v_2$ | 1 | 0 | 1 | 0 | 0 |
| $v_3$ | 1 | 1 | 0 | 1 | 1 |
| $v_4$ | 0 | 0 | 0 | 1 | 0 |
| $v_5$ | 0 | 0 | 0 | 1 | 0 |

$A(G)$, adjacency matrix of $G$

The adjacency matrix $A(G) = [b_{ij}]$ of a directed graph is also a $V \times V$ matrix,

$$\text{Where} \quad b_{ij} = \begin{cases} 1, \text{ if there is a directed edge from } v_i \text{ to } v_j \\ 0, \text{ otherwise} \end{cases}$$

(Similarly one can define the incidence matrix of a directed graph)
For example,

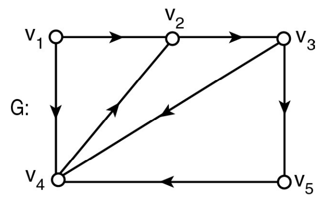


$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

**Example 2.5:** Write the adjacency matrix of graphs (i), (ii) and (iii).

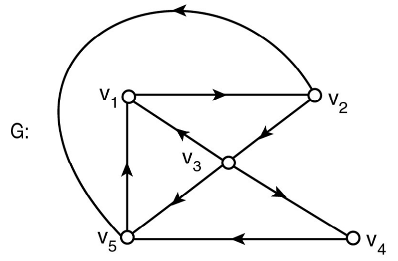
**Solution:** The adjacency matrices of the graphs are:

(*i*)



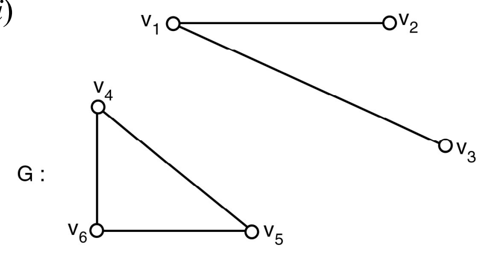
$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(*ii*)



$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(*iii*)



$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

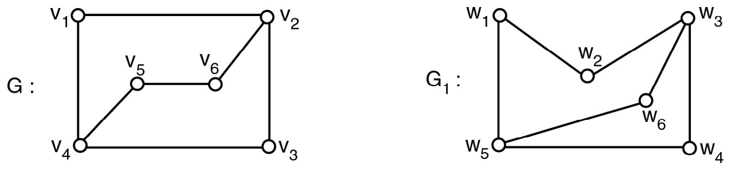
***Notes:*** From Example 2.5 one can conclude that:

1. The diagonal entries of an adjacency matrix are all zero, iff the graph is a graph with no self-loops.
2. If $G$ is disconnected and it has two components, then its adjacency matrix $A(G)$ can be written as,

$$A(G) = \begin{bmatrix} A(G_1) & 0 \\ 0 & A(G_2) \end{bmatrix}, G_1 \text{ and } G_2 \text{ are components.}$$

With the help of these matrices, one can verify whether the given graphs are isomorphic or not.

**Example 2.6:** Verify if $G$ and $G_1$ are isomorphic.



**Solution:** First lets write the adjacency matrices of $G$ and $G_1$.

$$A(G)=\begin{bmatrix}0&1&0&1&0&0\\1&0&1&0&0&1\\0&1&0&1&0&0\\1&0&1&0&1&0\\0&0&0&1&0&1\\0&1&0&0&1&0\end{bmatrix}\qquad A(G_1)=\begin{bmatrix}0&1&0&0&1&0\\1&0&1&0&0&0\\0&1&0&1&0&1\\0&0&1&0&1&0\\1&0&0&1&0&1\\0&0&1&0&1&0\end{bmatrix}$$

By keeping one matrix fixed and by applying permutation of rows and corresponding columns, permutations on the unfixed matrix yields the fixed one. Then, the given graphs are isomorphic.

It keeps $A(G)$ fixed.

Also $G$ and $G_1$ have 4 vertices of degree 2 and two vertices of degree 3. Since $d(v_1) = 2$ and $v_1$ is not adjacent to any other vertex of degree 2, corresponding vertex in $G_1$ is either $w_4$ or $w_6$, the only vertices of degree 2 in $G_1$ not adjacent to a vertex of degree 2.

Without the loss of generality, let us take $v_1 \to w_6$. Suppose this $v_1 \to w_6$ is not ending with isomorphism, one has to take $v_1 \to w_4$.

Similarly, for other vertices of $G$, it can be set as:

$v_2 \to w_3;\ v_3 \to w_4;\ v_4 \to w_5;\ v_5 \to v_1;\ \ v_6 \to v_2$.

Thus, we can modify $A(G_1)$ as

$$A(G_1)=\begin{array}{c|cccccc|} & w_6 & w_3 & w_4 & w_5 & w_1 & w_2\\ w_6 & 0&1&0&1&0&0\\ w_3 & 1&0&1&0&0&1\\ w_4 & 0&1&0&1&0&0\\ w_5 & 1&0&1&0&1&0\\ w_1 & 0&0&0&1&0&1\\ w_2 & 0&1&0&0&1&0\end{array}$$

$\therefore\ A(G) = A(G_1)$ and hence $G \cong\ G_1$.

## 2.5 PATH CIRCUIT

Now you will study about paths in directed graphs and relationship between a relation on a set, before transitive closure.

**Path:** A path from vertex $a$ to vertex $b$ in a directed graph $G$ is a sequence of one or more edges $(v_0, v_1), (v_1, v_2), ...,(v_{n-1}, v_n)$ in $G$, with $v_0 = a$; $v_n = b$, i.e., a sequence of edges whose terminal vertex is same as the intial vertex of the next edge in this path. This path is denoted by $v_0, v_1, ...,v_n$ of length $n$.

Let $R$ be a relation on a set $A = \{1, 2,..,n\}$ and $G_R$ be the corresponding relation graph whose vertices are $a = 1, v_1 = 2, ...,b = n$. There is a path in $G_R$ from $a$ to $b$ if there is a sequence of vertices $a, v_1, v_2,..., v_{n-1}, b$ with $(a, v_1) \in R$, $(v_1, v_2) \in R$, $(v_2, v_3) \in R$,..., $(v_{n-1}, b) \in R$.

**Problem 3:** Let $R$ be a relation on a set $A$. Then,

(*i*) $R^2 = R \circ R$, $R^n = R^{n-1} \circ R$

(*ii*) $R^n \subset R$

(*iii*) In $G_R$, the relational graph of $R$, there is a path of length $n$ from $a$ to $b$ if $(a,b) \in R^n$.

**Connectivity Relation:** Let $R$ be a relation on set $A$. The connectivity relation $R^*$ consists of the pairs $(a,b)$ such that there is a path between $a$ and $b$ in $R$.

$$\text{i.e., } R^* = \bigcup_{n=1}^{\infty} R^n$$

**Problem 4:** The transitive closure of a relation $R$ equals the connectivity relation $R^*$.

Let $R$ be a relation on a set $A$.

***Claim:*** $R^*$ is the transitive closure of $R$. To prove that,

(*i*) $R^*$ is transitive.

(*ii*) $S$ is a transitive relation on $A$ with $R \subseteq S$. Then $R^* \subseteq S$.

By definition, $R^* = \bigcup_{i=1}^{\infty} R^i$

$\therefore$ $R^*$ contains $R$.

(*i*) If $(a,b) \in R^*$ and $(b,c) \in R^*$, then there are paths from $a$ to $b$ and from $b$ to $c$ in $R$. Thus, a path is obtained from $a$ to $c$ by starting with the path $a$ to $b$ and following it with the path $b$ to $c$.

$\therefore$ $(b,c) \in R^*$.

i.e., $R^*$ is transitive.

(*ii*) Let $S$ be a transitive relation containing $R$.

Since, $S$ is transitive, $S^*$ is also transitive.

Further $S^* \subseteq S$. Since, $S^* = \bigcup_{i=1}^{\infty} S^i$ and $S^i \subseteq S$, $S^* \subseteq S$.

Since, any path in $R$ is also a path in $S$, $R^* \subseteq S^*$, if $R \subseteq S$.

Now one gets, $R^* \subseteq S^*$ and $S^* \subseteq S$.

$\Rightarrow R^* \subseteq S^*$

i.e., any transitive relation that contains $R$ must also contain $R^*$.

$\therefore$ $R^*$ is the transitive closure of $R$.

**Transitive Closure:** Let $M_R$ be the relation matrix of a relation $R$ on the set $A$ of $n$ elements. Then the transitive closure matrix $M_{R^*}$ is given by,

$$M_{R^*} = M_R \vee M_{R^2} \vee M_{R^3} \vee ... \vee M_{R^n}$$

**Example 2.7:**

(*i*) Find the transitive closure of a relation $R$ on the set $\{a,b,c\}$, whose relation matrix $M_R$ is given as

$$M_R = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

**Solution:** Let $R^*$ be the transitive closure of $R$. The relation matrix $M_{R^*}$ of $R^*$ is given as,

$$M_{R^*} = M_R \vee M_{R^2} \vee M_{R^3}$$

Now

$$M_{R^2} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}; \quad M_{R^3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$M_{R^*} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \vee \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \vee \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(*ii*) Find the transitive closure matrix of the relation $R$ whose relation matrix is given as

$$M_R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

**Solution:** Let $R^*$ be the transitive closure of $R$ and $M_R{}^*$ be the corresponding relation matrix.

we have $M_{R^*} = M_R \vee M_R{}^2 \vee M_R{}^3$

$$\text{Now} \quad M_{R^2} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}; \quad M_{R^3} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$M_{R^*} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \vee \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \vee \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

***Note:*** The transitive closure can be obtained by the following algorithm.

```
Transitive closure ($M_R$; 0 –1  $n \times n$ matrix)
$A \leftarrow M_R$
$B \leftarrow A$
for $i \leftarrow 2$ to $n$
begin
$A \leftarrow A \times M_R$
```

```
$B \leftarrow B \vee A$
end ($B$ is the required matrix $R^*$)
```

## Paths and Closures

A connected graph might contain more than one spanning tree. Consider the given graphs in Figure 2.30.

In $T_1$ the edges $e_1$, $e_2$, $e_5$, $e_6$ are present, whereas in $T_2$ the edges $e_2$, $e_4$, $e_5$, $e_6$ are present.

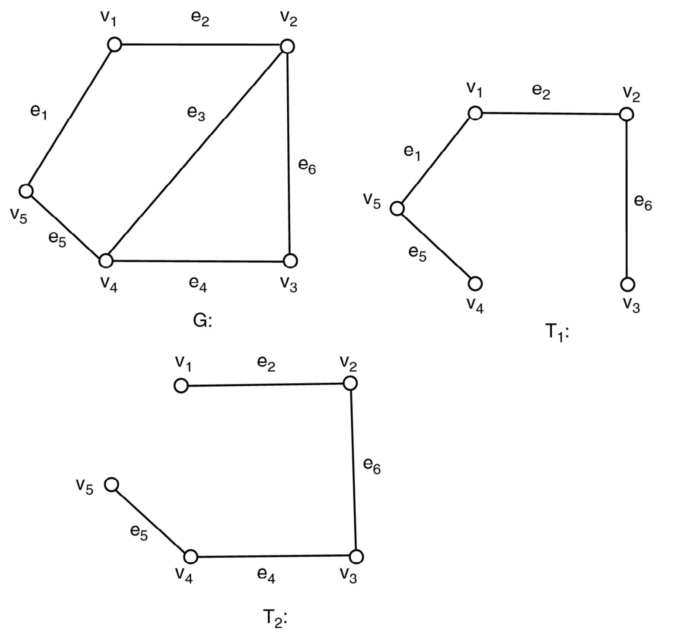


***Figure 2.30***

The edges of $G$ which are present in a spanning tree $T$, are called as the branches of $G$ with respect to $T$. The edges of $G$ which are not present in its spanning tree $T$ are called the *chords* of $G$ with respect to $T$.

In the above example, the branches of $G$ are $e_1$, $e_2$, $e_5$, $e_6$ with respect to $T_1$: the branches of $G$ are $e_2$, $e_4$, $e_5$, $e_6$, with respect to $T_2$.

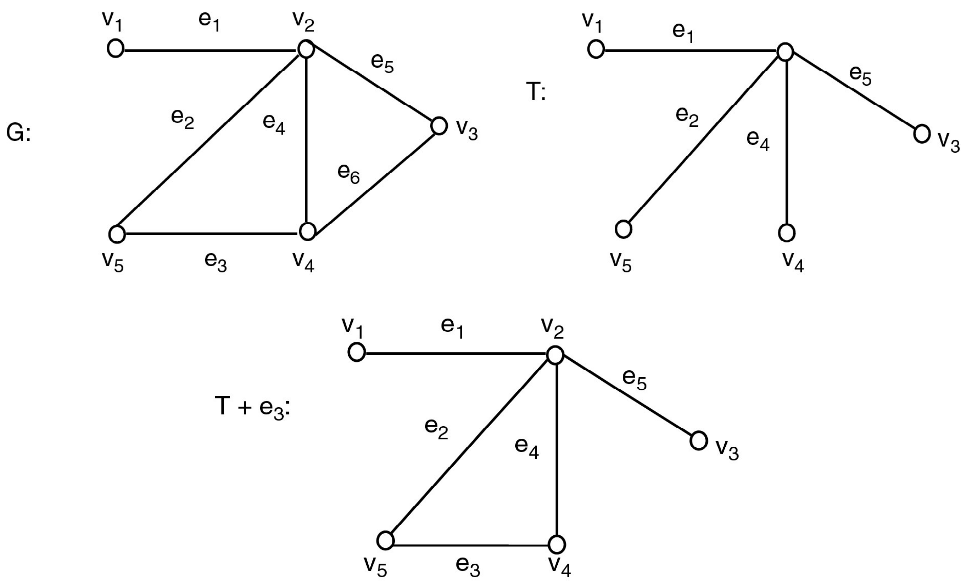
***Note:*** Let $G$ be a connected graph on $n$ vertices; $e$-edges and $T$ be one of its spanning tree. Since $T$ is a tree on $n$ vertices, it has $(n-1)$ edges, i.e., the number of branches of $G$ with respect to $T$ is $(n-1)$. The number of chords of $G$ with respect to $T$ is $e-(n-1)$. Often, the number of branches of $G$ is called as rank of $G$ and is denoted by $r(G)$; the number of chords of $G$ is called as the nullity of $G$, denoted by $\mu(G)$. In general, for a connected graph on $n$-vertices and $e$-edges, $r(G)$, the rank of $G$ is $(n-1)$ and $\mu(G)$, the nullity of $G$ is $e-n+1$.

## Fundamental Circuit

Let $T$ be the spanning tree of a connected graph $G$, and $e$ be a chord of $G$ with respect to $T$. Since the spanning tree $T$ is minimally acyclic, the graph $T$+$e$ contains a unique cycle. This cycle is called a fundamental cycle in $G$ with respect to $T$ as seen in Figure 2.31.

Every chord of $G$ gives rise to a fundamental cycle. Therefore, the number of fundamental cycles possible for a connected graph is atmost $\mu(G)$.

For example,

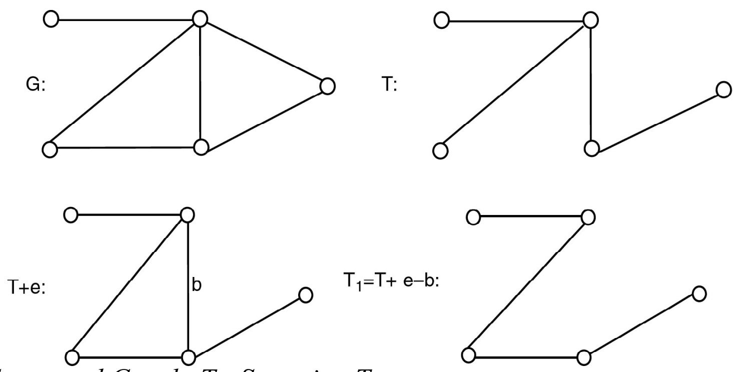


G – Graph, T – Spanning tree of G, T+$e_3$ – Fundamental Cycle

***Figure 2.31***

## Cyclic Interchange

Let $T$ be a spanning tree of $G$ and $e$ be a chord of $G$ with respect to $T$. The graph $T+e$ is a fundamental circuit. In this circuit other than edge $e$, all the other edges are branches of $G$ with respect to $T$. On removal of any of the branches from the fundamental circuit, one gets a spanning tree $T_1$, i.e., $b$ is a branch in the fundamental circuit (with respect to a chord $e$), then spanning tree $T_1$ is obtained by removing $b$ from $T+e$, i.e., $T_1 = T + e - b$. This process is called cyclic interchange (see Figure 2.32).



*G - Connected Graph, T - Spanning Tree*

*T+e - Fundamental Circuit; $T_1$ - Spanning Tree Obtained by Cyclic Interchange.*

***Figure 2.32***

### 2.5.1 Floyd's and Warshall's Algorithms

#### Floyd's Algorithm – Shortest Paths

Floyd's algorithm finds the paths which has least value between all the vertices of a graph. This requires matrix representation of the graph. The matrix represents the distance of edges between vertices which normally corresponds to the cost.

To apply this algorithm a matrix has to be made as a two-dimensional array. For a graph of $n$ vertices matrix will be $n \times n$. Every row in the matrix is a 'starting' vertex, denoted as $i$ and each column is an 'ending' point, denoted as $j$. An edge between $i$ and $j$ in the graph, length of this edge is the position $(i,j)$ of the matrix. In case of undirected graphs, edges are bidirectional, a value is given in position $(j,i)$ of the matrix. If there is no edge as a direct link between two vertices, value is given as infinity. Alternatively, very large value is put, to express the impossibility of movement from $i$ to $j$.

For example, in a graph connecting points 2 and 6, bidirectionally and the edge has a length of 24 units, number 24 will be placed into positions (2, 6) and (6, 2) of the matrix.

After setting such a matrix Floyd's algorithm can be used to find the shortest distance between every pair of vertices in the graph. The algorithm works for every non-direct path between pairs of vertices having least value than the way to move between these vertices. In the event of locating such a path, it is the value between these vertices which are to be tested. Each element of the matrix represents the least value of traversal between the vertices with respect to its row and column. If the graph is directed, $(i,j)$ and $(j,i)$ may not be equal.

#### Warshall's Algorithm

This algorithm is more efficient in determining the access of all pairs of nodes in a graph whether directed or undirected. For a graph $G$ with $n$ nodes, this method constructs a sequence of $n$ adjacency matrices, $P_1,...,P_n$, by using the same set of nodes. One starts by setting $P_0 = G$.

If $P_k$ is already defined then $P_{k+1}$ has all the edges of $P_k$, and additional edges, if any, needed to ensure that every pair of nodes joined by an edge of $P_k$ to node $k + 1$ are joined by an arc of $P_{k+1}$ (in the undirected case) and also for every path $a \to k + 1 \to b$. The pair $(a, b)$ is an edge of $P_{k+1}$ (in the directed case).

The algorithm terminates after $n$ iterations and $P_n$ contains all adjacency relationships which are shown as edges.

Floyd–Warshall's algorithm is an algorithm for graph analysis that finds shortest paths in a graph that is weighted and directed. The algorithm computes the shortest paths between all pairs of vertices. This algorithm is an example of dynamic programming.

A path in the matrix $k$ is defined in such a way that path $k[i][j]$ is true if and only if there is a path from node $i$ to node $j$ and there is no node higher than $k$, except $i$ and $j$ themselves. For any $i$ and $j$ if path $k[i][j]$ = True, it implies that path $(k+1)[i][j]$ is also true. If there is a situation in which path $(k+1)[i][j]$ is true while path $k[i][j]$ it is false, it is possible if there is a path from $i$ to $j$ via node $k + 1$, but no path from $i$ to $j$ via nodes $i$ through $k$. This means that there is a path from $i$ to $k + 1$ through nodes $i$ through $k$ and a similar path from $k + 1$ to $j$. This follows that the path $(k+ 1)[i][j]$ is true if and only if one of the following two conditions holds:

(*i*) Path $k[i][j]$ is true.

(*ii*) (Path $k[i][k+1]$) $\wedge$ (Path $k[k+1][j]$) is true.

Also, path $0[i][j]$ = adjacent. This is since, direct path is there from node $i$ to node $j$ with no intermediate node. It can also be noted that, path (*MAXNODES*-1)$[i][j]$ = Path$[i][j]$, because if a path exists through any node, then any path from node $i$ to node $j$ may be selected.

This is Warshall's algorithm which is named, after its discoverer.

This algorithm compares every possible path between every pair of vertices in the graph. It makes only $V_3$ comparisons. Maximum number of edges may be given as $V_2$ in the graph with every combination of edges tested. It estimates the shortest path between two vertices, by improving it incrementally to find an optimal solution.

Let there be a graph with a set $V$, of vertices and each vertex numbered 1 through $N$. Let there be a function defined as shortest path $(i, j, k)$ which returns the shortest possible path from $i$ to $j$ using only vertices 1, 2, 3, ….., $k$ as intermediate nodes. The objective is to find the shortest possible path from each $i$ to each $j$ by using only nodes 1, 2, 3, …., $k + 1$.

There are two alternative paths:

(*i*) Shortest path that uses nodes only in the set (1...$k$).

or

(*ii*) Another path that goes from $i$ to $j$, via $k + 1$.

Best path from $i$ to $j$ is the one that uses only nodes 1…. $k$ which is defined by the function. If there was a better path from $i$ to $j$ via $k + 1$, then the length of this path would be the sum total of the shortest path from $i$ to $k + 1$, traversing vertices 1.....$k$ and the shortest path from $k + 1$ to $j$ by using same set of vertices, i.e., 1…....$k$.

One may define shortest path $(i, j, k)$, that is recursive in nature.

This formula is the heart of Floyd Warshall algorithm which works by first computing shortest path $(i, j, 1)$ for all $(i, j)$ pairs, then using that to find shortest path $(i, j, 2)$ for all $(i, j)$ pairs, and so on and terminates when $k = n$. This finds the shortest path for all $(i, j)$ pairs by using any intermediate vertices.

## 2.5.2 Eulerian Path and Circuit

### Euler Path

In this section, special graphs and also the origin of the graph theory will be studied.

The Koingsberg town in The Russian Republic was separated into four lands by the river Pregel. These islands were connected by seven bridges. The problem was, could people walk from one island, travel across all the seven bridges and return to the island where they had started without using any bridge more than once? For almost two centuries, nobody was in a position to state whether such a walk was possible or not.

In 1736, the great mathematician Leonhard Euler concluded that such a walk was impossible. He used multigraph to study and solve this problem. Euler, today, is considered as the father of the graph theory.

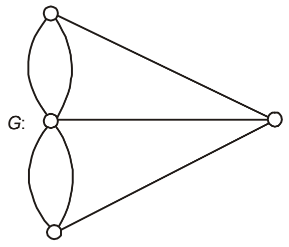


***Figure 2.33*** *Euler Path*

The four lands and the seven bridges are represented by vertices and edges respectively in G. (see Figure 2.33) This problem is called as Koingsberg bridge problem.

### Euler Circuits

A trail that traverses every edge of *G* is called an Euler trail of *G*. A circuit (tour) of *G* is a closed walk that traverses each edge of *G* exactly once. An Euler tour is a tour which traverses each edge exactly once. A graph is Eulerian if it contains an Euler tour.

**Theorem 2.5:** A connected graph is Eulerian iff it has no vertices of odd degree.

**Proof:** Let *G* be Eulerian and let *C* be an Euler tour of *G*, which begins and ends at some vertex *u*.

**Claim:** *G* has no vertices of odd degree, i.e., to prove that every vertex of *G* is even. Consider a vertex $w \neq u$. Since *w* is neither the first nor the last vertex of *C*, each time *w* is encountered, it is reached by some edge and left by another edge. Hence, each occurrence of *w* in *C* contributes 2 to its degree. Thus *w* is of even degree. This is true for all internal vertices of *C*. The initial occurrence and final occurrence of the vertex *u* in *C* contributes 1 to the degree of *u*. Therefore, every vertex of *G* is of even degree.

Conversely, let us assume that every vertex of a connected graph $G$ is even.

**Claim:** $G$ is Eulerian.

Suppose $G$ be a connected non-Eulerian graph with no vertices of odd degree.

Among such graphs, choose one, say $G$ having the least number of edges.

Since each vertex of $G$ has atleast two edges, $G$ contains a trail. Let $C$ be a closed trail of maximum possible length in $G$. By assumption, $C$ is not a Euler circuit of $G$ and hence $G - E(C)$ has edges.

Therefore, $G - E(C)$ has some component $G'$ with edges. Since $C$ itself is Eulerian, degree of every vertex in $C$ is even. Hence, degree of every vertex in $G - E(C)$ is also even. Therefore degree of every vertex in $G'$ is even. Since $E(G') < E(G)$, [by the choice of $G$ in (1)].

$G'$ is Eulerian and hence $G'$ has an Euler circuit (tour) say $C'$. Since $G$ is connected, there is a vertex $v$ in $V(C) \cap V(C')$ and one may assume without the loss of generality that $v$ is the initial and the terminal vertex of both circuits $C$ and $C'$. Now $(C \cup C')$ is a closed trail of $G$ with $E(C \cup C') > E(C)$. This contradicts the choice of $C$. Hence, every non-empty connected graph with no vertices of odd degree is Eulerian. Figure 2.34 shows examples of Eulerian graphs.

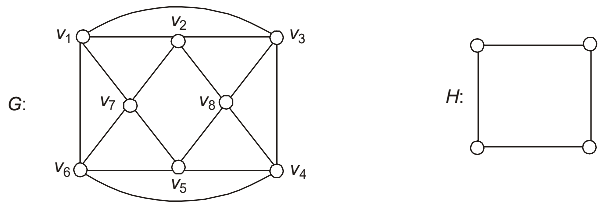


***Figure 2.34***

$G$ and $H$ are Eulerian graphs.

**Theorem 2.6:** A connected graph $G$ has an Eulerian trail iff $G$ has exactly two odd vertices.

**Proof:** Let $G$ be a connected graph with an Eulerian $(u - v)$ trail. By the similar argument in the previous theorem, it is concluded that all the vertices on the trail except $u$ and $v$, have even degree. Conversely, let $G$ be connected graph with two odd vertices $u$ and $v$. Let $G'$ be the graph obtain from $G$ by adding a new edge $e = uv$ between $u$ and $v$. By applying the previous theorem to $G'$, one can obtain an Eulerian tour in which the edge $e$ is the first edge. Hence, this Eulerian trail of $G$ can be obtained that starts at $v$ and ends at $u$. Therefore, $G$ has an Eulerian trail.

## Eulerian Digraphs

An Eulerian trail of a connected directed graph $D$ is a trail that contains all the edges of $D$; while an Eulerian circuit of $D$ is a circuit which contains every edge of $D$. A directed graph that contains an Eulerian circuit is called Eulerian digraph(See Figure 2.35).

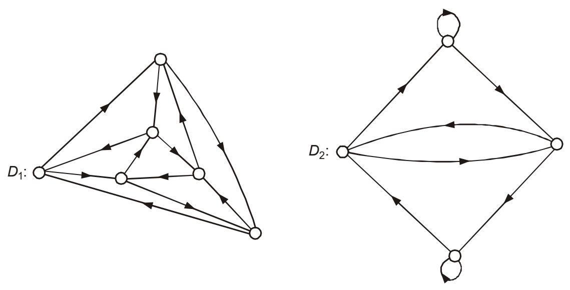


***Figure 2.35*** *Eulerian Digraphs*

**Theorem 2.7:** Let $D$ be a connected directed graph. $D$ is Eulerian iff $d^+(v) = d^-(v)$, $\forall\, v \in G$, then $G$ is called a balanced digraph.

**Proof:** Let $D$ be an Euler directed graph. Then $D$ contains an Euler circuit $C$ with common initial and terminal vertex $v$. Let, $b_u$ be the number of occurrence of an internal vertex $u$ in $C$.

Whenever $C$ enters $u$ through some edge incident into $u$, there is another edge incident out of $u$ through which $C$ leaves $u$. Thus, each occurrence of $u$ contributes one in-degree and one out-degree. Moreover, $C$ contains all the edges of $D$. Thus,

$$d^+(u) = d^-(u) = bu$$

Similarly, $d^+(v) = d^-(v)$

$\therefore$ Hence, $d^+(v) = d^-(v)$, $\forall\, v \in V(D)$

Conversely, suppose the connected digraph $D$ is balanced. Then, for each vertex $u$, $d^+(u) = d^-(u) \neq 0$. Start with an arbitrary vertex $u_1$, $d^+(u_1) \neq 0$. There exists an edge incident out of $u_1$. Let $u_2$ be the terminal vertex of this edge, $d^+(u_2) \neq 0$. Hence, there exists an edge incident out of $u_2$. Continuing like this one reaches a vertex which has been traversed directly. Thus a directed circuit $C_1$ in $D$ is obtained. If $E(C_1) = E(D)$, then $C_1$ is the required Euler circuit. If not, i.e., $E(C_1) \neq E(D)$, then remove all the edges of $C_1$ from $D$ to obtain a spanning subgraph $D_1$. Since $D$ is balanced, $D_1$ is also balanced. By applying the above process to $D_1$, one will obtain a circuit $C_2$ in $D_1$. If $E(D) = E(C_1) \cup E(C_2)$ then $C_1$ and $C_2$ can be combined to obtain an Euler circuit in $D_1$. Otherwise, edges of $C_2$ is removed from $D_1$ to obtain a spanning subgraph $D_2$ of $D$. The above process is repeated in $D_2$ and after a finite number of steps, one obtains an edge of disjoint circuits $C_1, C_2, ..., C_k$ such that $E(D) = E(C_1) \cup E(C_2) \cup ... \cup E(C_k)$. Since $D$ is connected, any two of these cycles have a common vertex and the circuits $C_1$, $C_2$, ... ,$C_k$ can be combined to obtain an Euler circuit in $D$. Hence, $D$ is an Euler graph.

### 2.5.3 Hamiltonian Graphs

In 1857, Sir William Rowan Hamilton invented a game called 'Around the World'. In this game, a solid regular dodecahedron (20 vertices, 30 edges and 12 faces) and a supply of string is given. Every vertex is given the name of an important city. The objective of the game is to find a route along the edges of dodecahedron that visits every city exactly once and terminates where it started

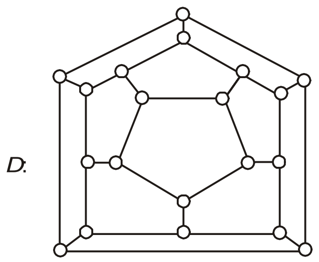


***Figure 2.36*** *Hamiltonian Graphs*

The graph $D$ is a dodecahedron.

Another famous problem is 'The Knight's Puzzle'. Is it possible for a knight to tour the chess board, i.e., visit each square exactly once and return to its initial square?

It can be represented by a graph $G$, where the vertices $u_i$ correspond to squares $S_i$ of the chess board and $u_j$ is adjacent to $u_i$ iff it is possible for a knight to proceed from $S_i$ to $S_j$ in a single move.

To solve 'Around the World' and 'Knight's Puzzle', one must determine if the given graph is Hamiltonian.

A path that contains every vertex of $G$ is called a Hamilton path of $G$. Similarly, a Hamilton cycle of $G$ is a cycle that contains every vertex of $G$ in other words spanning cycle. A graph is Hamiltonian if it contains a Hamilton cycle or a spanning cycle.

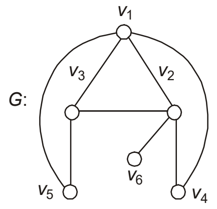
**Example 2.8:** Prove that $k_n$ has a Hamiltonian circuit, $\forall\, n \geq 3$.

**Solution:** Let us construct the Hamiltonian circuit in $k_n$ ($n \geq 3$) as follows:

Choose a vertex arbitrarily in $k_n$ and begin the Hamiltonian circuit at this vertex. Such a circuit can be built by traversing vertices in any order, as long as the path begins and terminates at the same vertex and visits other vertices exactly once. This is possible in $k_n$, since every vertex is adjacent to all other vertices. Also $k_1$ and $k_2$ has only Hamiltonian path, not circuit.

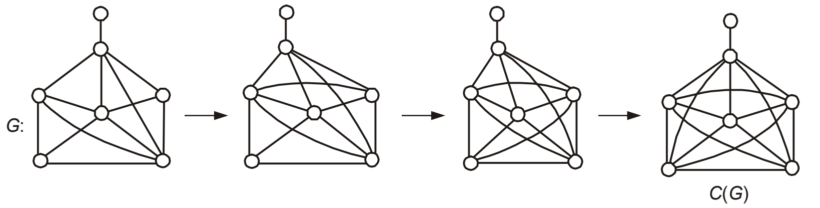
**Graphical:** A sequence $d = (d_1, d_2, ...., d_n)$ is graphical if there exists a simple undirected graph on $n$ vertices with the degrees of the vertices $d_1, d_2, ..., d_n$ respectively.

For example, a graph with degree sequence of vertices $v_1$ to $v_6$ (4, 4, 3, 2, 2, 1) is shown in the following figure.



**Closure:** A closure (CG) of a $n$-vertex graph $G$ is a graph from $G$ by recursively joining pairs of non-adjacent vertices whose degree sum is atleast $n$ until no such pair remains.

For example,



The above figures explain one way of constructing a closure of a graph.

**Important Theorems**

**Theorem 2.8:** Let $G$ be a $n$-vertex graph. Suppose $G_1$ and $G_2$ are two graphs obtained from $G$ by recursively joining pairs of non-adjacent vertices whose degree sum is atleast $n$. Then $G_1 = G_2$. In other words, $C(G)$, the closure of a graph $G$ is unique.

**Proof:** Let $e_1, e_2, ..., e_k$ and $f_1, f_2, f_3, ..., f_e$ be the edges added to $G$ to obtain $G_1$ and $G_2$, respectively. It has to be proved that every $e_i$ $(1 \leq i \leq k)$ is an edge of $G_2$ and $f_j$ $(1 \leq j \leq l)$ is an edge of $G_1$. Suppose that some edge in the sequence $e_1, e_2, ..., e_k$ does not belong to $G_2$. Let $p$ be the smallest positive integer such that $e_{p+1}$ is not an edge of $G_2$. Let $e_{p+1} = uv$ and $H = G + \{e_1, e_2, ..., e_p\}$. Then, $H$ is a subgraph of $G_1$ and $G_2$. By the construction of $G_1$, one gets

$$d_H(u) + d_H(v) \geq n$$

Therefore, $d_{G_2}(u) + d_{G_2}(v) \geq d_H(u) + d_H(v) \geq n.$

This is a contradiction since $u$ and $v$ are non-adjacent in $G_2$. Therefore, each $e_i$ is an edge of $G_2$. Similarly, each $f_j$ belongs to $G_1$. Hence, $G_1 = G_2$.

**Theorem 2.9:** A graph $G$ is Hamiltonian iff its closure $C(G)$ is Hamiltonian.

**Proof:** Let $e_1, e_2,...e_n$ be edges added to $G$ to obtain its closure $C(G)$. Let $G_i$ be the graph obtained from $G$ by adding the edge $e_i$.

By repeated application of Example (2.8).

$G$ is Hamiltonian $\Leftrightarrow$ $C(G)$ is also Hamiltonian.

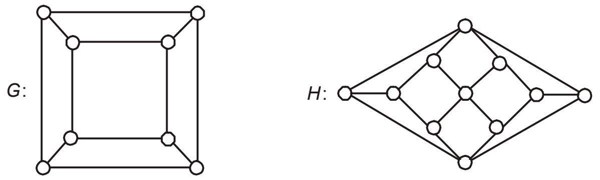
**Corollary 1:** Let $G$ be a graph with atleast 3 vertices. If $C(G) \cong k_n$, $(n \geq 3)$ then $G$ is Hamiltonian.

**Proof:** By corollary 1, $k_n$ is Hamiltonian. Since $C(G) \cong k_n$, $C(G)$ is Hamiltonian and hence $G$ is also Hamiltonian.

**Corollary 2:** Let $G$ be a graph with atleast 3 vertices iff $d(u) + d(v) \geq n(n \geq 3)$, for all pairs $u$ and $v$ of non-adjacent vertices of $G$, then $G$ is Hamiltonian.

**Proof:** Let $G$ be a graph with atleast 3 vertices. Given that $d(u) + d(v) \geq n$ $(n \geq 3)$ for all pairs of non-adjacent vertices of $G$. Hence, one can add edges between such pair of vertices to obtain $C(G)$. Since, $C(G)$ is complete by corollary, $G$ is also Hamiltonian.

For example,



$G$, Hamiltonian graph and $H$, non-Hamiltonian graph.

**Theorem 2.10:** If $G$ is Hamiltonian then, for every non-empty proper subset $S$ of $V$, $w(G - s) \leq |s|$.

**Proof:** Let $G$ be a Hamiltonian graph and $S$ be a proper subset of $V$. Since, $G$ is Hamiltonian, $G$ has a Hamiltonian cycle $C$. Suppose $w(G - S) = n$, where $G_1$, $G_2$,... $G_n$ are the components of $G - S$. Let $u_i$ $(1 \leq i \leq v)$ be the last vertex of $C$ that belongs to $G_i$ and let $v_i$ be the vertex that immediately follows $u_i$ on $C$. Clearly $v_i \in S$ for each $i$ and $v_j \neq v_k$ for $j \neq k$. Hence, there are atleast as many vertices in $S$ as components in $G - S$.

i.e., $w(G - S) \leq |S|$.

**Weight graph:** A graph $G$ is called a weight graph if every edge of $G$ is assigned with a real number.

## Travelling Salesmen Problem (TSP)

Suppose that a salesman is expected to take a trip through a given collection of $n$ cities $(n \geq 3)$. What route should he take to minimize the total distance travelled? This can be represented as a weight graph. Let $G$ be a connected weight graph whose vertices represent the cities to be visited and let the weight of an edge $v_i\, v_j$ be the distance between the cities $v_i$ and $v_j$. Now TSP is equivalent to finding a minimum Hamiltonian cycle in a connected weight graph.

### CHECK YOUR PROGRESS

9. What does a path mean in a directed graph?
10. What is the function of Floyd's algorithm?
11. What is the function of Warshall's algorithm?
12. What is Floyd–Warshall algorithm?

## 2.6 GRAPH COLOURING

**Colouring**

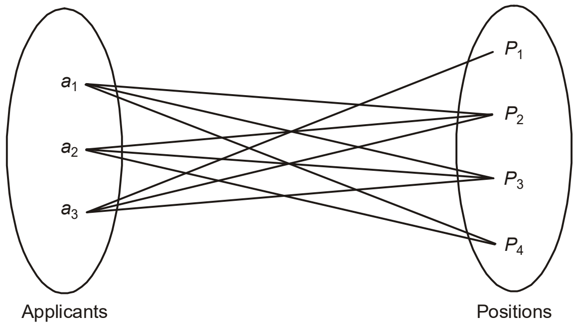
No two adjacent vertices having the same colour is called the proper colouring or colouring of a graph. A graph $G$ that requires $K$ different colours (minimum number) for its proper colouring can be referred to as $k$-chromatic graph, and the number $k$ is called the chromatic number of $G$.

A graph consisting of only isolated vertices is 1-chromatic.

A graph with one or more edges (not a self-top) is at least 2-chromatic also called bichromatic.

**Bipartite Graph:** A graph $G$ is called bipartite if its vertex set $V$ can be decomposed into two disjoint subsets $v_1$ and $v_2$ such that every edge in $G$ joins a vertex in $v_1$ with a vertex in $v_2$.

***Note:*** Every tree is a bipartite graph.



**Theorem 2.11:** Every tree with two or more vertices is bichromatic.

**Proof:** Choose any vertex $r$ in the given tree $T$. Let $T$ be a rooted tree at vertex $v$. Paint $v$ with colour 1. Paint all vertices adjacent to $v$ with colour 2. Next, paint the vertices adjacent to these using colour 1. Repeat this process till every vertex in $T$ has been painted. Hence all vertices at odd distances from $v$ have colour 2 while $v$ and vertices at even distances from $v$ have colour 1.

Along any path in $T$ the vertices are of alternating colours. Since there is one and only one path between any two vertices in a tree, no two adjacent vertices have the same colour.

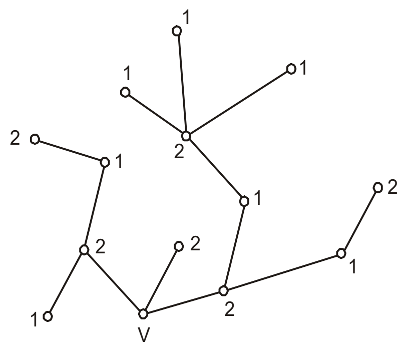


***Figure 2.37*** *Proper Colouring of a Tree*

**Theorem 2.12:** A tree with atleast one edge is bichromatic.

**Proof:** Let $T$ be a tree and $v$ be any vertex of $T$. For this vertex $v$ assigns colour 1. Assign colour 2 to be adjacent vertex of $v$. Continuing this process, we can see that all vertices at even distance from $v$ are assigned colour 1 and the vertices at odd distance from $v$ are assigned colour 2.

This gives a two colouring of $T$. Since $T$ is a connected graph with atleast one edge, $T$ is not 1-coloured.

$\therefore$ $T$ is bichromatic.

**Theorem 2.13:** A graph is bipartite iff it is bichromatic.

**Proof:** Let $G$ a bipartite graph with at least one edge. Let $(v_1, v_2)$ be the partition of the vertex set of $G$. In $v_1$ as well as in $v_2$, no two vertices are adjacent. Now assign colour 1 to the vertices in $v_1$ and assign colour 2 to the vertices in $v_2$. Hence, $G$ is bichromatic.

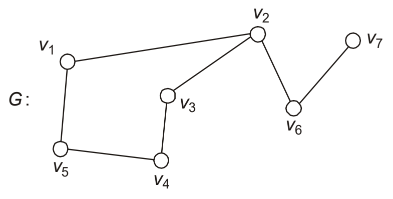
Conversely, let us assume that $G$ is a bichromatic graph. Therefore $x(G) = 2$. Let $v_1$ be the set of all vertices for which colour 1 is assigned and $v_2$ be the set of all vertices for which colour 2 is assigned. Clearly $(v_1, v_2)$ is the partition of the vertex set of $G$. Otherwise at least two vertices in $v_1$ or $v_2$ have the same colour. Therefore, $G$ is a bipartite graph.

***Note:*** A graph $G$ is bipartite if it contains no odd cycles.

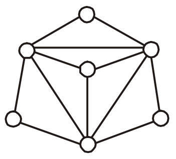
**Independent Set:** A set $U$ of vertices in a graph $G$ is called an independent set if no two vertices in $v$ are adjacent in $G$. An independent set $U$ of vertices in a graph $G$ is called a maximal independent set if $U$ is not a proper subset of any other independent set of vertices of $G$. The cardinality of a maximal independent set is called as an independence number and is denoted by $\beta(G)$.

For example, here $\{v_1, v_4, v_2, v_7\}$, $\{v_1, v_6\}$, $\{v_1, v_7\}$, $\{v_2, v_7\}$ are all independent set and the sets $\{v_1, v_4, v_2, v_7\}$ is a maximal independent set and $\beta(G) = 4$.

**Dominating Set:** A set $S$ of vertices in a graph $G$ is a dominating *set* if every vertex not in $S$ is adjacent to a vertex in $S$. A dominating set $S$ is called as a minimal dominating set if no proper subset of $S$ is a dominating set. The cardinality of a minimal dominating set is called as domination number and is donoted by $\sigma(G)$.



For example,



*G*, Graph with $\sigma(G) = 2$

***Note:*** For every graph $G$, $\sigma(G) \leq \beta(G)$.

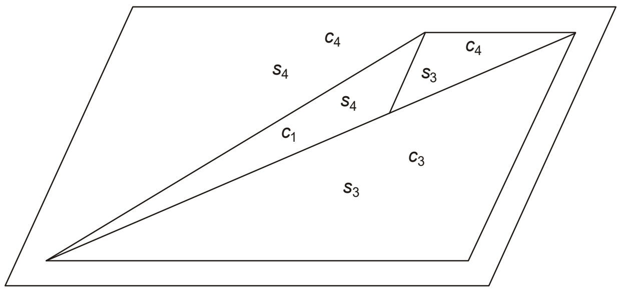
NOTES

**Map Colourings**

When we colour the map of a country we see to it that its neighbouring states are coloured differently. In such circumstances, one may ask what is the minimum number of colours needed to colour so that the adjacent states receive different colours?

From the following example, one can say that 3 colours are not sufficient.

For example,



In the above map, $S_i$ denotes the states and $C_i$ denotes the colours received by $S_i$, clearly 3 colours are not sufficient for map colouring so that no two adjacent states receive the same colour.

The above map can be represented as a graph. The states are denoted by vertices and the adjacency between states are denoted by edges. Now the graph obtained in the above fashion is not 3-colourable.

***Note:*** Every map can be represented as a planar graph.

**The four colour problem:** Can the regions of a planar graph be coloured with four colours so that the adjacent states are coloured differently?

Yes, every planar graph is 4-colourable.

**Theorem 2.14:** Every simple planar graph is 4-colourable. It is discussed as follows:

### 2.6.1 Four Colour Theorem

If we are given a graph which has many regions and we have to show these regions separate in the graph, we should colour this graph such that no two adjacent regions have same colour. How many colours we need so that every region is shown separate? It has been discovered that four colours are sufficient for this purpose in case of a planar graph. This is 'four colour theorem' which is true for any planar graph. This is also known as 'four colour map theorem'.

Although it is possible to colour a map by using only three colours, but this is inadequate when a region is surrounded by three regions. In that case a fourth colour becomes necessary. Few map-makers make use of fifth colour as an easier approach, but it is optional as four colours are sufficient for such applications.

In 1976, this theorem was proved. For this computer was used. All maps were categorized into more than 1900 cases. One special-purpose computer program was run and the concept of four colour theorem was tested. But since this could not be verified by hand, it was not accepted by all mathematicians.

As per the four colour theorem, *every planar map is four colourable*. This conclusion was based on the work of Appel and Haken. Although, not accepted by all, the original work of Kempe has put forward some basic tools, which was used at a later stage, to prove it.

Adding edges to a graph does not decrease its chromatic number. Thus, a maximal planar graph, a kind of triangular graphs, where every face is bounded by three edges, is to be considered here. Let $v$, $e$, and $f$ be the number of vertices, edges, and faces, respectively. We know that each edge is shared by two faces, hence, $2e = 3f$. Using this fact with Euler's formula $v - e + f = 2 \Rightarrow 6v - 2e = 12$. Let $v_n$ be the number of vertices of degree $n$ and $D$ be the maximum degree. Then,

$$6v - 2e = 6\sum_{i=1}^{D} v_i - \sum_{i=1}^{D} iv_i = \sum_{i=1}^{D}(6-i)v_i = 12$$

Here, $12 > 0$ and $6 - i \le 0$ and so, $i \ge 6$. Thus, there is at least one vertex of degree 5 or less. There may be more than one vertex which may have degree 5.

If a maximal planar graph exists that requires 5 colours, then there is a *minimal* such graph and removing any vertex reduces it to four-colourable. Let us take a graph $G$. This graph can not have a vertex having degree 3 or less than 3. If $d(v) \le 3$, $v$ can be removed from $G$, we colour the smaller graph using four-colour and then re-add $v$. After re-adding $v$, we extend the four-colouring to this graph by selecting a colour different from its neighbours.

Kempe argued that $G$ can have no vertex of degree 4. Once again, we remove a vertex $v$ and apply four colours for remaining vertices.

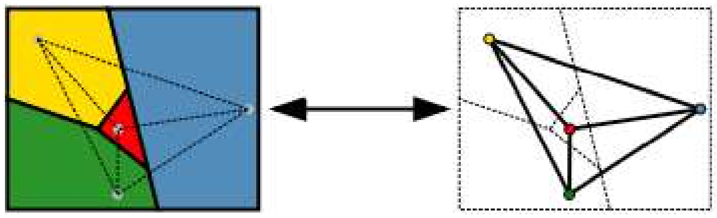


***Figure 2.38*** *Colouring a Graph*

If we select the four colours say red, green, blue, and yellow in clockwise order in the neighbourhood of *v,* these are different. We look for an alternating path of vertices coloured red and blue joining the red and blue neighbours. This path, on the name of Kempe, is known as Kempe chain. Thus, there may be a Kempe chain that joins red and blue neighbours, similarly, there may be a Kempe chain that joins green and yellow neighbours, but not both. This is so since these two paths will intersect, and the intersecting vertex can not be coloured. Let us consider that the red and blue neighbours are not chained together. If we explore all vertices attached to the red neighbour using red-blue alternating paths, and then reverse the colours on all these vertices. The result so obtained is again a four-colouring which is valid and we add *v* back.

The four colour theorem is not applicable in geopolitical mapping where regions of a country are non-contiguous. For example, Alaska is a part of the United State and is noncontiguous whereas Leningrad is a part of Russia.

**CHECK YOUR PROGRESS**

13. What does the four colour theorem say?
14. In which case is the four color theorem not applicable?

## 2.7 SUMMARY

In this unit, you have learned that:

- A graph is a triplet consisting of a non-empty set of vertices, a set of edges, a function that is assigned to each edge and a subset that need not be distinct.
- Every graph has a diagram associated with it and this diagram is useful in understanding the problems involved in the graph.
- A simple graph is called bipartite if its vertex set can be partitioned into two disjoint non-empty sets in such a way that every edge in the graph connects a vertex.
- In a graph's structure, a trail is a walk in which no edge is repeated and a path is a trail in which no vertex is repeated.
- Two graphs *G* and *H* are said to be isomorphic if there exists bijections in them.
- Bijection is a function *f* from a set *X* to a set *Y* with the property that, for every *y* in *Y*, there is exactly one *x* in *X* such that $f(x) = y$ and no unmapped element exists in either *X* or *Y*.
- Two graphs are homeomorphic if an isomorphism is found from any subdivision of one graph to the subdivision of another graph.

- Subdivision of a graph refers to another graph that results from subdivision of edges in that graph.
- Barycentric subdivision is a special subdivision that subdivides every edge of a graph.
- The degree of a vertex is the number of edges having that vertex as an end point, and is denoted by $d(v)$.
- A vertex with degree zero is called an isolated vertex.
- A vertex with degree one is called a pendant vertex.
- A path from various vertices in a directed graph is a sequence of one or more edges.
- Floyd's algorithm finds those paths which have least value between all the vertices of a graph and it requires the matrix representation of the graph.
- Floyd's algorithm works for every non-direct path between pairs of vertices having least value than the way to move between these vertices.
- Warshall's algorithm is more efficient in determining the access of all pairs of node in a graph, whether directed or undirected.

## 2.8 KEY TERMS

- **Simple graph:** A graph with no self loops and parallel edges is called a simple graph.
- **Pseudograph:** A graph with self loops and parallel edges is called a pseudograph.
- **Isolated vertex:** A vertex with degree zero is called an isolated vertex.
- **Pendant vertex:** A vertex with degree one is called a pendant vertex.
- **Adjacent vertices:** A pair of vertices that determine an edge are called adjacent vertices.
- **Complete graph:** A simple graph in which each pair of distinct vertices is joined by an edge is called a complete graph.
- **Bipartite graph:** A simple graph is called bipartite if its vertex set can be partitioned into two disjoint non-empty sets.
- **Floyd's algorithm:** It finds the paths which has least value between all the vertices of a graph.
- **Floyd–Warshall algorithm:** Floyd–Warshall algorithm is an algorithm for graph analysis that finds shortest paths in a graph that is weighted and directed.

## 2.9 ANSWERS TO 'CHECK YOUR PROGRESS'

NOTES

1. In the diagrammatic representation of a graph, vertices are represented by small circles and edges by lines whenever the corresponding pair of vertices forms an edge.
2. A graph with no self loops and parallel edges is called a simple graph.
3. A graph with self loops and parallel edges is called a pseudograph.
4. The various types of graphs are as follows:
   (i) Bipartite graphs
   (ii) Subgraph
   (iii) Complete graph
5. The edge connectivity l($G$) of a graph is the minimum cardinality of a set $S$ of edges of $G$ such that $G - S$ is disconnected.
6. The vertex connectivity $K(G)$ of a graph $G$ is the minimum number of vertices whose deletion makes $G$ a disconnected or a trivial graph.
7. Two graphs are said to be isomorphic if there exist bijections (it is a function $f$ from a set $X$ to a set $Y$ with the property that, for every $y$ in $Y$, there is exactly one $x$ in $X$ such that $f(x) = y$ and no unmapped element exists in either $X$ or $Y$) amongst them.
8. Subdivision of a graph is another graph that results from subdivision of edges in that graph.
9. A path from various vertices in a directed graph is a sequence of one or more edges.
10. Floyd's algorithm finds the paths which have least value between all the vertices of a graph and it requires a matrix representation of the graph.
11. Warshall's algorithm determines the access of all pairs of nodes in a graph, whether directed or undirected.
12. Floyd–Warshall algorithm is an algorithm for graph analysis that finds the shortest paths in a graph that is weighed and directed.
13. As per the four colour theorem, *every planar map is four colourable*.
14. The four colour theorem is not applicable in geopolitical mapping where regions of a country are non-contiguous.

## 2.10 QUESTIONS AND EXERCISES

NOTES

**Short-Answer Questions**

1. What do you understand by the degree of vertex?
2. Differentiate between edge-connectivity and vertex-connectivity.
3. What do you understand by connectivity relation?
4. Write a short note on cyclic interchange.
5. Write a short note on Euler path.

**Long-Answer Questions**

1. Explain the various types of graphs and their operations.
2. Explain cut-vertices and cut-edges.
3. Explain the structures of various type of graphs.
4. Explain with the help of a diagram the concepts of adjacent and incidence matrices.
5. Discuss the characteristics of Floyd–Warshall algorithm.
6. Describe Euler circuits.

## 2.11 FURTHER READING

Lipschutz, Seymour and Lipson Marc. *Schaum's Outline of Discrete Mathematics,* 3rd edition. New York: McGraw-Hill, 2007.

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms.* Hyderabad: Orient BlackSwan, 2008.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

Brassard, Gilles and Paul Bratley. *Fundamentals of Algorithms*. New Delhi: Prentice Hall of India, 1995.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. New Jersey: Pearson, 2006.

Baase, Sara and Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. New Jersey: Pearson, 2003.

Mott, J.L. *Discrete Mathematics for Computer Scientists*, 2nd edition. New Delhi: Prentice-Hall of India Pvt. Ltd., 2007.

Liu, C.L. *Elements of Discrete Mathematics*. New Delhi: Tata McGraw-Hill Publishing Company, 1977.

Rosen, Kenneth. *Discrete Mathematics and Its Applications*, 6th edition. New York: McGraw-Hill Higher Education, 2007.

# UNIT 3 TREES

NOTES

**Structure**

## 3.0 INTRODUCTION

In this unit, you will learn about the various types of tree structures and their applications. In a graph theory, a tree is defined as a graph in which two vertices are connected by one and only one path. Generally, trees are known as open graphs. An organizational hierarchy is considered to be a very good example of a tree structure. In a tree, every edge is a cut-edge. Traversal algorithm is a methodical way for visiting each and every vertex of an ordered rooted tree. An ordered rooted tree is primarily used for representing any arithmetic expressions and compound proposition expressions.

You will learn how trees are helpful in merge sort. In addition, you will learn to compute mathematical expressions by using algorithms and formulae and their graphical representation by using tree structures.

## 3.1 UNIT OBJECTIVES

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

- Identify the various types of trees
- Understand the basics of a tree
- Explain the features of minimum height and minimum distance spanning trees
- Comprehend the meaning and functions of planar graphs

## 3.2 TREES: BASICS

NOTES

In this section, you will study the characteristics of a tree. In graph theory, a tree is a graph in which any two vertices are connected by exactly one path.

**Acyclic Graph:** A graph $G$, which has no cycles is called an acyclic graph.

**Tree:** A connected acyclic graph $G$ is called a tree.

Figure 3.1 shows some tree graphs:

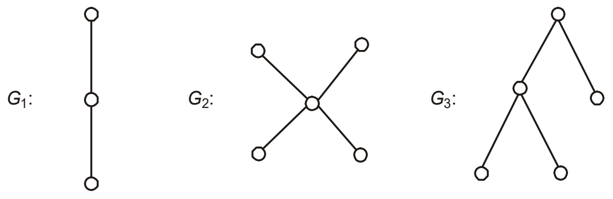


***Figure 3.1*** *Tree Graphs*

***Notes:***

1. Trees are often also known as open graphs.
2. Any organizational hierarchy is an example of tree.

Before proceeding further to learn the types of trees, let us understand some theorems.

**Theorem 3.1:** Every two vertices in a tree, are joined by a unique path.

**Proof:** (By contradiction) let $G$ be a tree and assume that there are two distinct, $(v, w)$ paths $P_1$ and $P_2$ in $G$. Since $P_1 \neq P_2$, there is an edge $e = V_1V_2$ of $P_1$ that is not in $P_2$. Clearly $(P_1 \cup P_2) - e$ is connected. Therefore, it contains a $(V_1 - V_2)$ path $P$. Now $P + e$ is a cycle in the acyclic graph $G$, which is a contradiction to the fact that $G$ is a tree.

**Theorem 3.2:** If $G$ is a tree on $n$ vertices, then $G$ has $(n - 1)$ edges.

**Proof:** By induction on the number of vertices.

When $n = 1$, $E(G) = 0 = n - 1$ $(G \cong K_1)$

When $n = 2$, $E(G) = 1 = n - 1$ $(G \cong K_2)$

Let us assume that this theorem is true for all trees of $G$ with fewer than $n$ vertices.

Now, let $G$ be a tree on $n$ vertices. Let $e = uv$ be an edge in $G$. Then $G - e$ is disconnected and $G$ has two components say $G_1$ and $G_2$ of $G - e$. Since $G$ is acyclic, $G_1$ and $G_2$ are also acyclic and hence $G_1$ and $G_2$ are also trees. Moreover $G_1$ and $G_2$ has fewer than $n$ vertices say $n_1$ and $n_2$, respectively. Therefore, by induction hypothesis,

$G_1$ has $(n_1 - 1)$ edges and $G_2$ has $(n_2 - 1)$ edges.

$\therefore \quad E(G) = E(G_1) + (G_2) + 1$

(Here, 1 in the sum corresponds to the edge $e$)

$$= (n_1 - 1) + (n_2 - 1) + 1 = n_1 + n_2 - 1 = n - 1$$

Therefore, any vertex tree, has $(n - 1)$ edges.

**Theorem 3.3:** Every tree has at least two vertices of degree one or in a tree, there are atleast two pendant vertices.

**Proof:** Let $G$ be a tree on $n$ vertices. Then,

$$d(v) \geq 1, \ \forall v \in v(G) \qquad ...(3.1)$$

Already we have, $\sum_{v \in v(G)} d(v) = 2 \times E(G) = 2 \times e \qquad ...(3.2)$

Since $G$ is an $n$-vertex tree, it has $(n - 1)$ edges.

$\therefore \quad \sum_{v \in v(G)} d(v) = (2n - 2) \qquad ...(3.3)$

From equations (3.1) and (3.3), it follows that $d(v) = 1$ for at least two vertices.

***Note:*** In a tree, every edge is a cut-edge.

Now, let us learn about the various types of trees.

## 1. Rooted Tree

In a directed tree (every edge assigned with a direction), a particular vertex is called a root if that vertex is of degree zero. A tree together with its root produces a graph called a rooted tree as shown in Figure 3.2. Note that in the rooted tree, every edge is directed away from the root.

For example, suppose $T$ is a rooted tree. If a vertex $u$ is a vertex in $T$ other than the root then the parent of $u$ is the unique vertex $u_1$ such that there is a directed edge from $u_1$ to $u$. Here, $u$ is called as a child of $u_1$. Vertices of the same parent are called as siblings. A vertex of a rooted tree is called as a leaf if it has no children and those vertices which have children, are called as internal vertices.

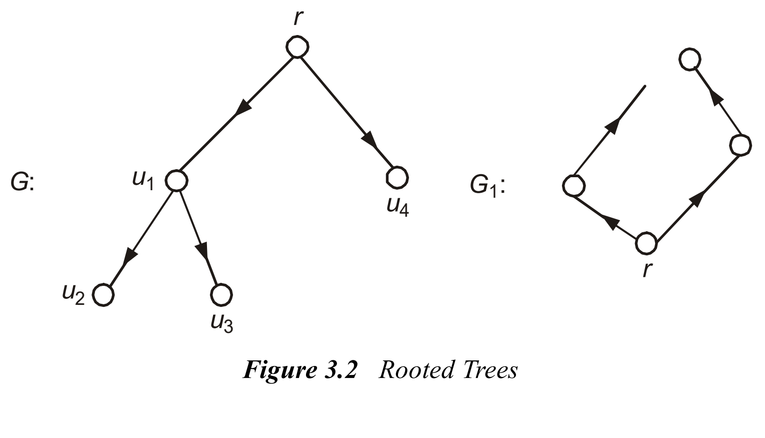


***Figure 3.2*** *Rooted Trees*

If $v$ is a vertex in a tree, then a subtree with $v$ as its root is the subgraph of the tree consisting of $v$ and its children and all edges incident to these children as shown in Figure 3.3:

NOTES

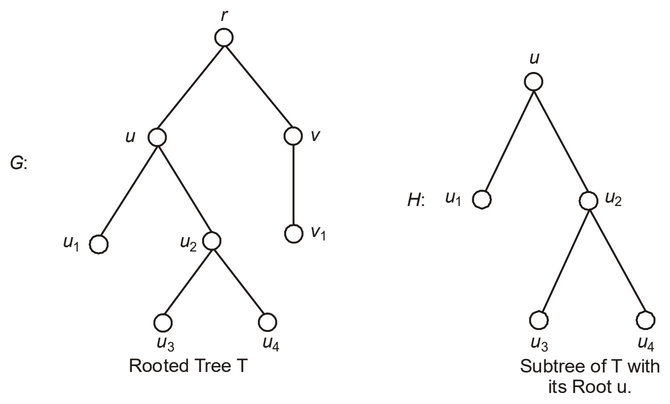


***Figure 3.3*** *Rooted Tree and Subtree*

### Level and Height in a Rooted Tree

The level of a vertex $v$ in a rooted tree is the length of the path from the root to this vertex. The height of a rooted tree is the length of the longest path from the root to any vertex as illustrated in Figure 3.4:

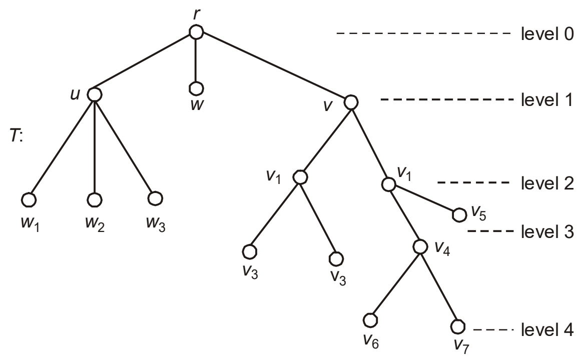


***Figure 3.4*** *Height of a Tree*

Rooted Tree $T$ with its different Levels.

Height of $T$ is 4.

### 2. *k*-ary Tree

A rooted tree is called a $k$-ary tree if every internal vertex does not have more than $k$-children. The tree is called a full $k$-ary tree if every internal vertex has exactly $k$-children. A $k$-ary tree with $k = 2$ is called a binary tree.

Figure 3.5 shows some $k$-aray trees.

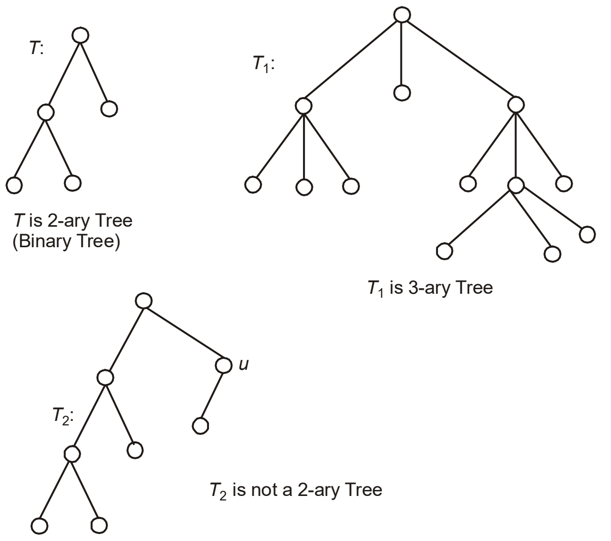


***Figure 3.5** Types of k-ary Trees*

NOTES

$T_2$ is not a 2-ary tree because vertex $u$ has only one child, whereas all the other vertices have two children.

A tree $T$ is called as a binary tree if there is at least one vertex with degree 2 and the remaining vertices are of degree 1 or 2.

**Example 3.1:** Prove that a full $k$-ary tree with $i$-internal vertices contains $k_{i+1}$ vertices.

**Solution:** In a full $k$-ary tree, every internal vertex has $k$ children and hence a full $k$-ary tree with $i$-internal vertices can have $k_i$ vertices. If we include the root, the tree has $k_{i+1}$ vertices. By looking at the fall of $k$-ary tree, we can observe the following:

(*i*) $n$ vertices has $i = (n - 1)/k$ internal vertices and $p = [(k - 1)\ n + 1]/k$ leaves.

(*ii*) $i$ internal vertices has $n = k_{i+1}$ vertices and $p = (m - 1)\ i + 1$ leaves.

(*iii*) $p$ leaves has $n = (kp - 1)/(k - 1)$ vertices and $i = (p - 1)/(k - 1)$ internal vertices.

### 3. Balanced Tree

A rooted $k$-ary tree of height $h$ is balanced if all the leaves are at level $h$ or $(h - 1)$.

### 4. Binary Search Trees

Binary search tree is a binary tree in which each child is either a left or right child; no vertex has more than one left child and one right child, and the data are associated with vertices.

**Example 3.2:** Build a binary search tree for the words banana, peach, apple, pear, coconut, mango and papaya using the alphabetical order.

**Solution:**

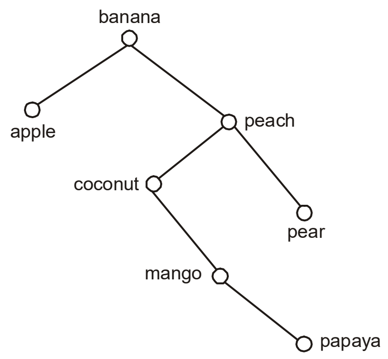


***Figure 3.6*** *Binary Search Tree*

For, if apple < peach, coconut < pear.

Further, mango is the right child of coconut and papaya is the right child of mango.

**5. Decision Trees**

If in a rooted tree, each internal vertex is assigned to a decision with a sub-tree at the vertices, then each possible outcome of the decision is called a decision tree.

**Traversal of a tree**

A systematic method of visiting every vertex of an ordered rooted tree is called a 'Traversal Algorithm'.

**Pre-order:** Let $T$ be an ordered rooted tree with root $r$. Suppose $T$ has only one vertex say $r$, then $r$ is the pre-order traversal of $T$. Suppose that $T_1, T_2, ..., T_k$ are the subtrees at $r$ from left to right in $T$, then the pre-order traversal begins by visiting $r$. It continues by traversing $T_1$ in pre-order, then $T_2$ in pre-order and so on, until $T_k$ is reached. This is illustrated in Figure 3.7:

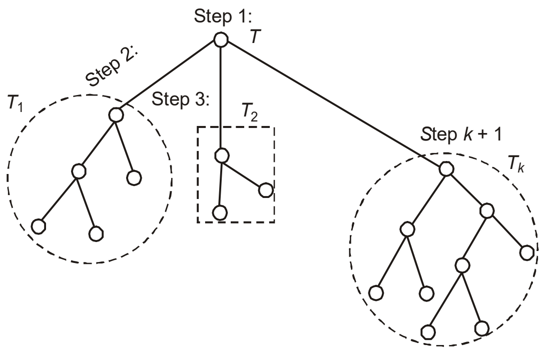


***Figure 3.7*** *Pre-Order Traversal*

**Step 1.** Visit the root *r*.

**Step 2.** Visit $T_1$ in pre-order.

**Step 3.** Visit $T_2$ in pre-order.

**Step *k* + 1.** Visit $T_k$ in pre-order.

NOTES

An example of pre-order traversal is presented in Figure 3.8:

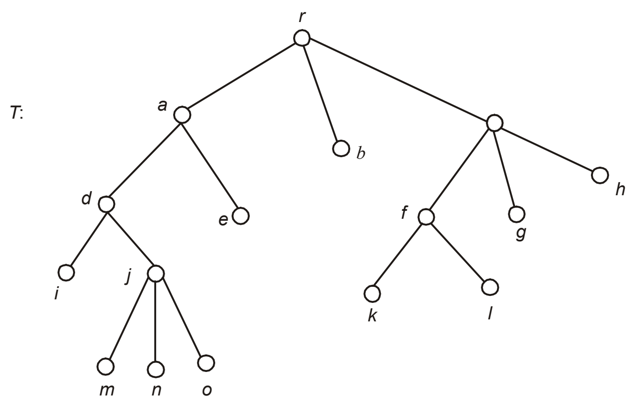
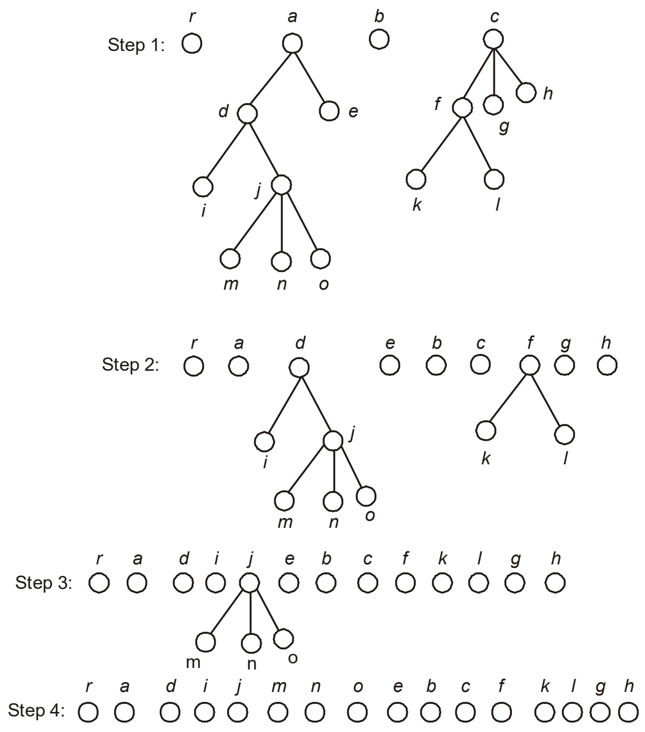


***Figure 3.8*** *Example of Pre-order Traversal*

Let *T* be an ordered root tree. The steps of the pre-order traversal of *T* are as follows:

In Figure 3.8, first you traverse *T* in pre-order by listing the root *r*, followed by the pre-order list of subtree with root *a*, the pre-order list of subtree with root *b*, and the pre-order list of subtree with root *c*. These steps are shown in Figure 3.9



Step 1:
r a b c d e f g h i j k l m n o
Step 2:
r a d e b c f g h i j k l m n o
Step 3:
r a d i j e b c f k l g h m n o
Step 4:
r a d i j m n o e b c f k l g h

***Figure 3.9*** *Steps of Pre-order Travesal*

NOTES

**Algorithm: Pre-order traversal**

**Step 1.** Visit root $r$ and then list $r$.

**Step 2.** For each child of $r$ from left to right, list the root of the first subtree then, the next sub-tree and so on until you complete listing the roots of subtrees at level 1.

**Step 3.** Repeat step 2, until you arrive at the leaves of the given tree.

**Step 4.** Stop.

**In-order Traversal**

Let $T$ be an ordered, rooted tree with its root at vertex $r$. Suppose $T$ consists of only root $r$, then $r$ is the in-order traversal of $T$. If not, i.e., suppose $T$ has subtrees $T_1$, $T_2$, ..., $T_k$ at $r$ from left to right. The in-order traversal begins by traversing $T_1$ in-order, then visiting $r$. It continues by traversing $T_2$ in-order, then $T_3$ in-order and so on and finally $T_k$ in-order. This is shown in Figure 3.10:

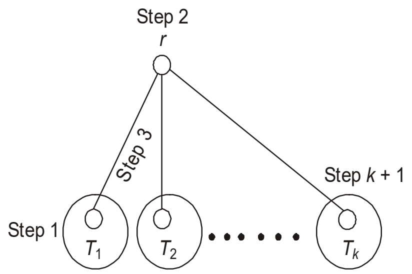


***Figure 3.10*** *In-Order Traversal*

**Step 1.** Visit $T_1$ in-order.

**Step 2.** Visit root.

**Step 3.** Visit $T_2$ in-order.

**Step *k* + 1.** Visit $T_k$ in-order.

**Example 3.3:** Determine the order in which the vertices of the rooted tree shown in Figure 3.11 is visited using an in-order traversal.

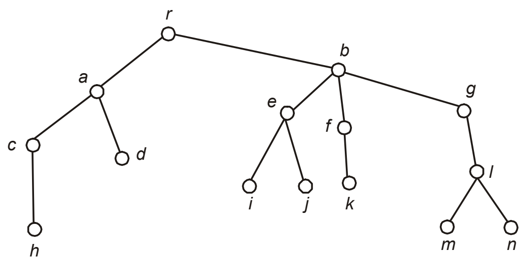


***Figure 3.11*** *Rooted Tree*

**Solution:** As shown in Figure 3.12, the in-order traversal begins with an in-order traversal of the subtree with root at $a$, followed by the root $r$, and the in-order listing of the subtree with root $b$.

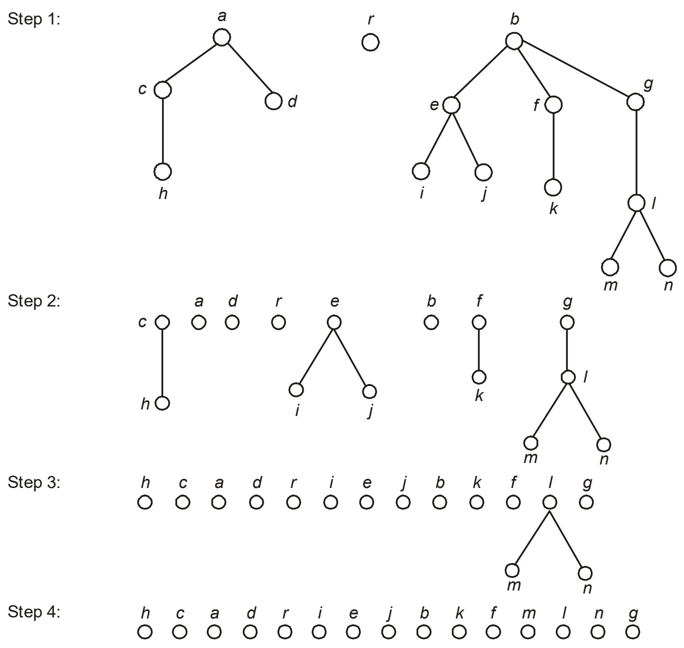
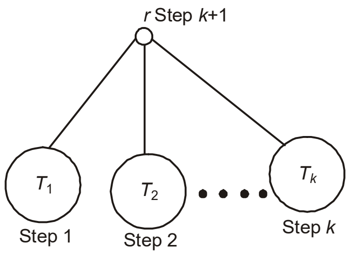


*Figure 3.12* Steps of In-order Traversal

NOTES

## Post-order traversal

Let $T$ be an ordered rooted tree with root $r$. If $T$ has only one vertex $r$, then $r$ is the post-order traversal of $T$. But if $T$ has subtrees $T_1$, $T_2$, ..., $T_k$ at $r$ from left to right, the post-order traversal begins by traversing $T_1$ in post-order, then $T_2$ in post-order and so on until $T_k$ is reached and ends by visiting $r$. This is shown in Figure 3.13:



(*ii*)

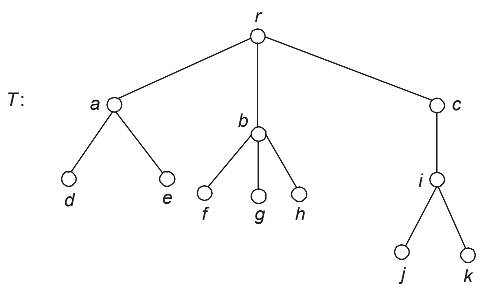


*Figure 3.13* Post-order Traversal

NOTES

The post-order traversal begins with the post-order traversal of the subtree with root *a*, the post-order traversal of the subtree with root *b*, and the post-order traversal of the subtree with root *c*, followed by the root *r*. These steps are shown in Figure 3.14:

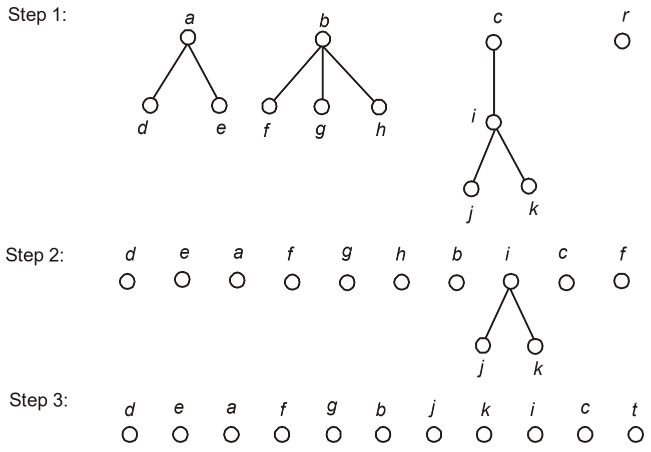


***Figure 3.14*** *Steps of post-order traversal*

**Infix, Prefix and Postfix Notation**

One can represent any expression (like arithmetic, compound proposition) using ordered rooted trees. An ordered rooted tree can be used to represent expressions, where the internal vertices represent operations, the leaves represent the variables or numerals.

**Example 3.4:** What is the ordered rooted tree that represents the expression $((a + b)\uparrow 3) + ((a - 6)/3)$?

**Solution:** First construct a subtree for $a + b$. Then this tree is included as a part of the next subtree of $((a + b)\uparrow 3)$. Similarly a subtree is constructed for $(a - 6)$ then this tree is included as a part of the next subtree of $(a - 6)/3$. Finally the subtrees $((a + b)\uparrow 3)$ and $(a - 6)/3$ are combined to form the required tree corresponding to the given expression. This is shown in Figure 3.15:

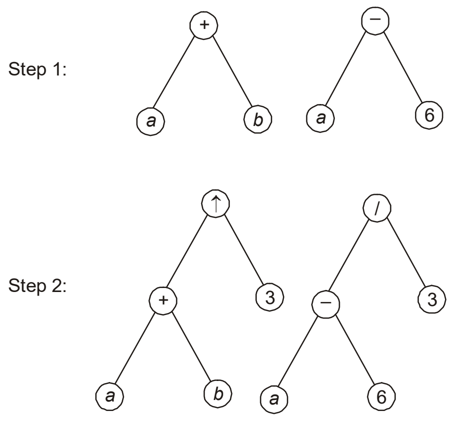
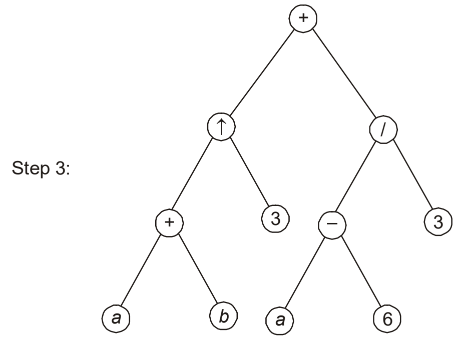


***Figure 3.15*** *Ordered Rooted Tree Corresponding to the Expression* $((a + b)\uparrow 3) + ((a - b)/3)$

(*Figure 3.15 contd...*)

**NOTES**



An in-order traversal of the binary tree representing an expression, produces the original expression with the elements and operations in the same order as they originally appeared (except unary operator).

If you use parenthesis, whenever you encounter an operation where there will be no ambiguity. Such fully parenthesized expression is said to be *infix form*.

To get the *prefix form* of an expression, we traverse its rooted tree in pre-order.

Expressions written in prefix form are called polish notations.

**Example 3.5:** What is the prefix form of $((a + b)\uparrow 3) + ((a - 6)/3)$?

**Solution:** The ordered rooted tree corresponding to the expression $((a + b)\uparrow 3) + ((a - 6)/3)$ is shown in Figure 3.16:

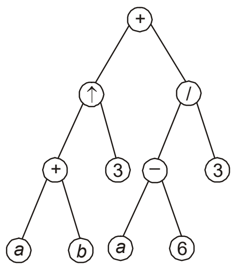


***Figure 3.16*** *Ordered rooted tree of (( a +b)↑3) + ((a–6)/3)*

One obtains the prefix form of the given expression, one has to traverse the binary tree in pre-order. Prefix form of the expression $((a + b)\uparrow 3) + ((a - 6)/3$ is $+ \uparrow ab\ 3/ - a\ 63$.

One obtains the postfix form of an expression by traversing its binary tree in pre-order.

**Example 3.6:** What is the postfix form of $((a + b)\uparrow 3) + ((a - 6)/3)$?

**Solution:** The binary tree corresponding to the expression is given in the Figure 3.17:

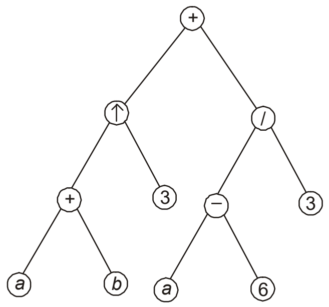


***Figure 3.17*** *Binary Tree for* $((a+b)\uparrow 3) + ((a-6)/3)$

To obtain the postfix form of the given expression, one has to traverse its binary tree in post-order. The required *postfix form* is $ab + 3 \uparrow a6 -3/+$.

**Example 3.7:** Draw the decision tree that orders the elements of the list $a$, $b$, $c$.

**Solution:**

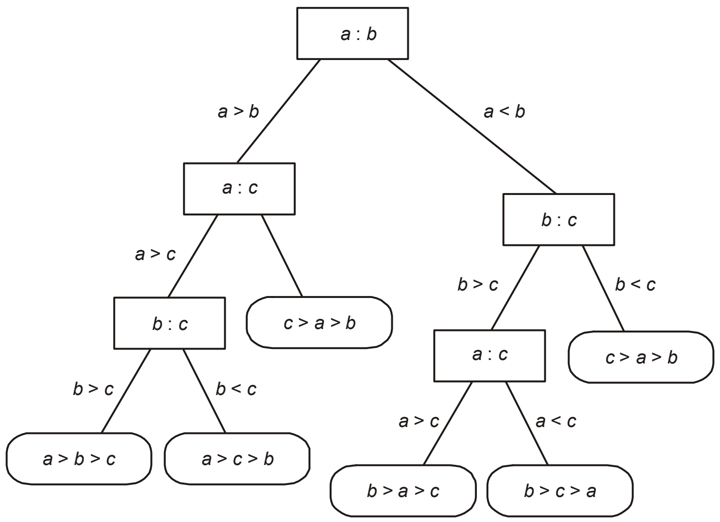


***Figure 3.18*** *Decision Tree*

### 3.2.1 Trees and Sorting

To sort a list of elements there are several methods. Here, it will be seen how trees are helpful in merge sort.

In general, a merge sort proceeds by iteratively splitting lists into two sublists of equal size (nearly) until each sublist consists of only one element.

This succession of sublists can be represented by a balanced binary tree. The procedure continues by successively merging pairs of lists (where both lists are in increasing order) into a larger list with elements in increasing order until the original list is put into increasing order. The succession in a merged list can be represented by a balanced binary tree.

NOTES

**Example 3.8:** Draw the recursive tree for merge sort of the list 9, 7, 11, 4, 5, 3, 6, 8, 12, 10.

**Solution:** The list of elements can be represented as:

$$a[0] = 9;\ a[1] = 7;\ a[2] = 11;\ a[3] = 4;\ a[4] = 5;$$
$$a[5] = 3;\ a[6] = 6;\ a[7] = 8;\ a[8] = 12;\ a[9] = 10.$$

Let us denote 0–9 as the position of the elements. Given list is [9, 7, 11, 4, 5, 3, 6, 8, 12, 10].

As a first step, this list is splitted into two sublists of size 0-4 and 5-9, respectively. Then these two sublists are further splitted into two sublists until each sublist consist of only one element. The required tree is given in Figure 3.19:

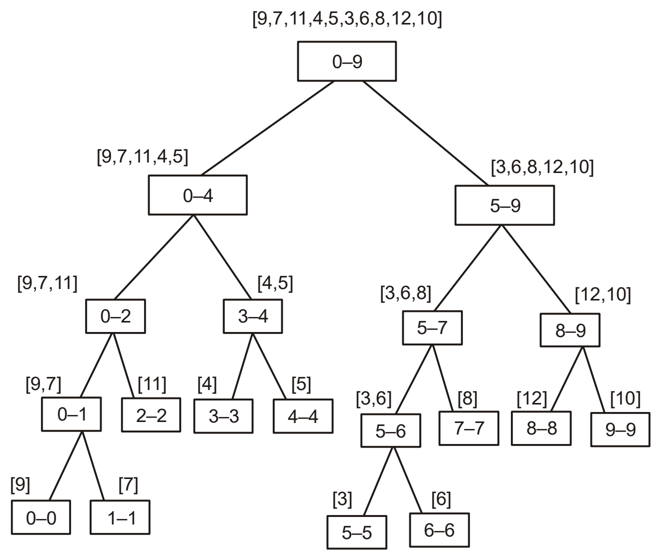


***Figure 3.19*** *Recursive Tree*

**Check Your Progress**

1. What is an acyclic graph?
2. Define a rooted tree.
3. What is the vertex of a rooted tree known as?
4. Define a binary tree.
5. What is a decision tree?

## 3.3 MINIMUM HEIGHT AND MINIMUM DISTANCE SPANNING TREES

NOTES

In this section, study will be done for the spanning acyclic subgraph of a connected subgraph, and its optimality.

Let *G* be a simple connected graph. A spanning tree of *G* is a subgraph of *G*, i.e., a tree containing every vertex of *G*. This is shown in Figure 3.20:

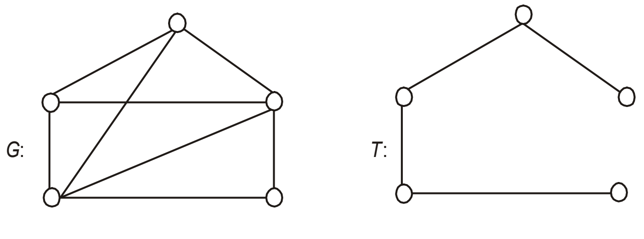


***Figure 3.20*** *Simple Graph G and its Spanning Tree T*

**Theorem 3.4:** A simple graph is connected if there exists atleast one spanning tree.

**Proof:** Let *G* be a simple connected graph. Suppose *G* has no circuits then *G* itself is a spanning tree. Suppose *G* has a simple circuit. By deleting an edge from one of these simple circuits, the resulting subgraph is still connected if it is a spanning subgraph. If this subgraph has simple circuits, then delete an edge from one of these simple circuits. Repeat this process until no simple circuits are there. Thus in this manner a tree *T* is obtained in which $V(T) = V(G)$. Therefore, *T* is a spanning tree of *G*.

***Note:*** The converse of this theorem also holds true.

### 3.3.1 Depth-First Search and Breadth-First Search

One can build the spanning tree of a connected graph by using Depth-First Search (DFS) and Breadth-First Search (BFS). First, it will be seen how DFS are useful in construction of a spanning tree from a given connected graph.

#### Depth-First Search

Let *G* be the given connected graph. Arbitrarily, select a vertex as the root. Find a path starting from this choosen vertex by successively adding edges, where each edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding edges to this path as long as possible. If this path consists of all the vertices of *G*, then this path is the required spanning tree. If not, then more edges should be added. Navigate back to the vertex next to last that is in this path, and if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this is not possible, move to another vertex in this path (i.e., 2 vertices back from the last) and try again. Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new long paths until no more edges can be added. This process ends with a spanning tree.

When this procedure returns to the vertices previously visited, it is also called as backtracking.

**Example 3.9:** Construct a spanning tree for the graph *G* which is shown in Figure 3.21.

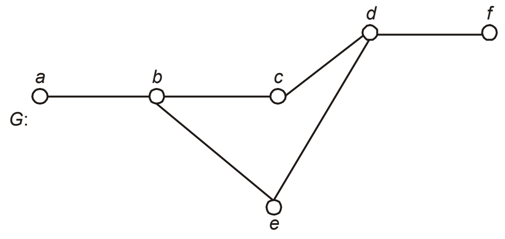


***Figure 3.21*** *Graph G*

**Solution:** First, arbitrarily choose a vertex, say *e* as the root. Form a path at *e*, i.e., *cdf* is the path. Backtrack to *d*. Form a path beginning at *d* in such a way that it has to visit the vertices which where not visited in the previous path, *d e b a*. Since all the vertices of *G* are visited, this procedure gives the spanning tree *T*, which is shown in Figure 3.22.

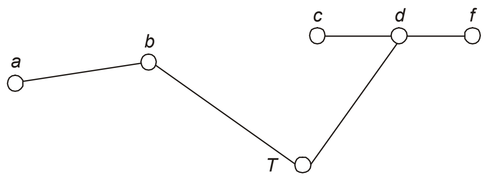


***Figure 3.22*** *Spanning Tree T of Graph G*

### Breadth-First Search

First, choose a vertex arbitrarily as the root. Add the edges of *G* which are incident with this vertex. The new vertices added at this stage becomes level 1 in the spanning tree. Order these vertices arbitrarily. Next, for each vertex at level 1 visited in order, add each edge incident to this vertex to the tree as long as it does not- create a simple circuit. Order the children of each vertex at level 1 arbitrarily. This produces the vertices at level 2 in the tree. Continue in this manner until all the vertices of *G* have been added. Ultimately a spanning tree, is created.

**Example 3.10:** Construct a spanning tree of the graph *G*. Which is shown in Figure 3.23:

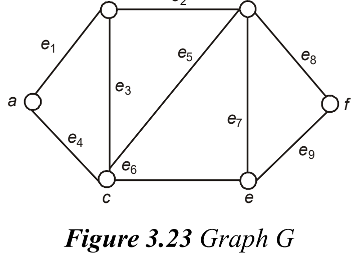


***Figure 3.23*** *Graph G*

**Solution:** First choose a vertex say $d$ (arbitrarily) as the root. Add the edges incident to this vertex $d$. Hence, the edges $e_2, e_5, e_7, e_8$ are incident with the vertex $d$. These vertices create level 1, as shown in Figure 3.24:

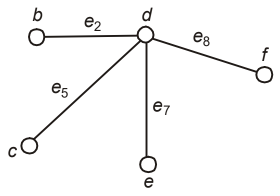


***Figure 3.24*** *Level 1*

Now, add the edges which are incident to the vertices $b, c, e, f$ in such a way, that the resulting graph does not contain any circuit. Thus, at this level itself you have got the spanning tree $T$. Which is shown in Figure 3.25:

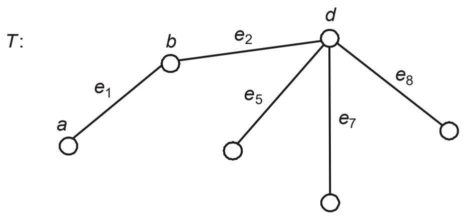


***Figure 3.25*** *Spanning tree*

***Note:*** If the given graph is a directed graph, then you can construct the underlying undirected graph and apply DFS or BFS to obtain a spanning graph.

### 3.3.2 Optimal Spanning Graph

Let $G$ be a weighted graph. Every edge of the graph is associated with a real number. We have to find the minimum weight spanning tree of the graph $G$. The minimum weight spanning tree is called an optimal spanning tree. Weight of a tree is the sum of weights of the edges in a tree and is denoted by $wt(T)$.

There are three algorithms to find the optimal spanning tree.

(*i*) Kruskal's algorithm

(*ii*) Prim's algorithm

(*iii*) Boruvka's algorithm

#### Kruskal's Algorithm

Let $G$ be a connected graph on $n$ vertices.

**Step 1:** Arrange the edges in ascending order according to their weights. Choose the minimum weight edge say $e_1$.

**Step 2:** Having selected $e_1, e_2, ..., e_k$ in such a way that the subgraph formed by these edges $<e_1, e_2, ..., e_k>$ is acyclic, choose $e_{k+1}$ such that of the remaining edges, weight of $e_{k+1}$ is minimum.

**Step 3:** Repeat steps 1 and 2 until $(n-1)$ edges are selected.

This is shown in Figure 3.26:

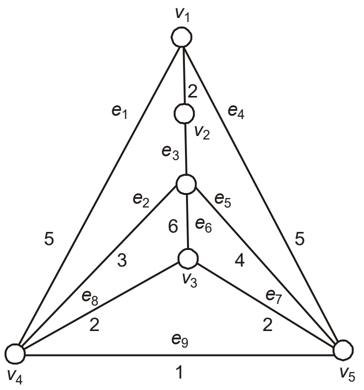
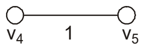


***Figure 3.26*** *Connected Graph G according to Weights*

**Equations:** $e_9, e_7, e_8, e_3, e_2, e_5, e_4, e_1, e_6$

Among these equations $e_9$ has the minimum weight 1.



After applying step 2 and step 3, the spanning tree created is shown in Figure 3.27:

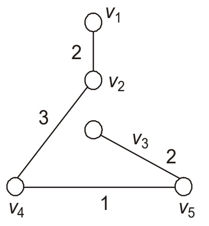


***Figure 3.27*** *Spanning Tree according to Weights*

Weight of the optimal spanning tree is $2 + 3 + 1 + 2 = 8$

**Prim's Algorithm**

Let $G$ be a connected graph.

**Step 1:** Arbitrarily choose a vertex say $v_1$ and an edge $e_1$ with minimum weight among the edges incident with $v_1$.

**Step 2:** Having selected the vertices $v_1, v_2,...,v_k$ and the edges $e_1, e_2,...,e_k$ choose the edge $e_{k+1}$ as follows. $e_{k+1}$ in incident with any one of the vertices $\{v_1, v_2, ..., v_k\}$ and incident with $v(G) - \{v_1, v_2, ..., v_k\}$. Moreover the subgraph formed with $v_1, v_2,...,v_k, v_{k+1}$ and the edges $e_1, e_2,.., e_k, e_{k+1}$ is acyclic and of the remaining edges $e_{k+1}$ has minimum weight.

**Step 3:** Repeat steps 1 and 2 till $(n - 1)$ edges are arrived.

NOTES

This is shown in Figure 3.28:

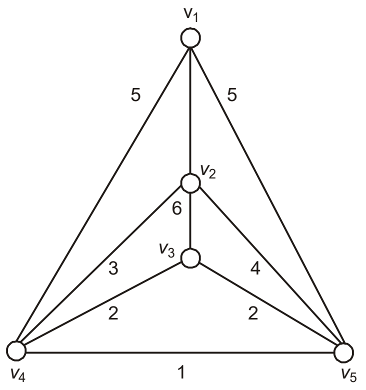


***Figure 3.28*** *Connected Graph n according to Prim's Algorithm*

**Step 1:** Choose arbitrarily vertex $v_3$ and apply step 2 and step 3. Now, you will get the spanning trees. Which are shown in Figure 3.29:

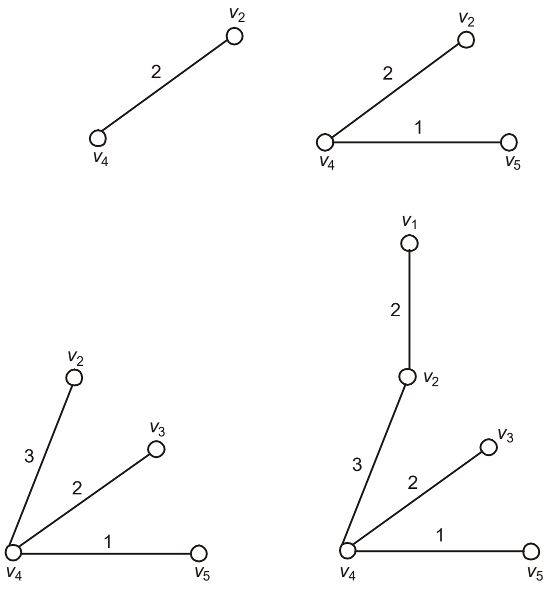


***Figure 3.29*** *Spanning Trees according to Prim's Algorithm*

So the final weight of the spanning tree is 8.

**Boruvka's Algorithm**

Boruvka's algorithm finds a minimum spanning tree in a weighted graph. Boruvka developed this for constructing an efficient electrical network.

Every vertex in the graph finds its lightest edge, and then the vertices at the ends of each lightest edge are marked. This process goes and the entire graph collapses into a single point. The tree consists of all the lightest edges are so found.

The algorithm starts by examining every vertex one-by-one and selecting the cheapest edge from that vertex to another in the graph, without regard about the already added edges. It continues joining these groupings in a similar manner and a tree spanning all vertices is formed.

Every vertex or set of connected vertices is termed as a 'component'. The pseudocode for this algorithm is given as follows:

NOTES

```
1. Start with a connected graph G containing edges of distinct weights, and an
   empty set of edges T.
2. While vertices of G connected by T are disjoint,
     • Start with an empty set of edges E.
     • For each edge in the component,
         – Start with an empty set of edges S.
     • For each vertex in the component,
         – Add the cheapest edge from the vertex in the component to
           another vertex in a disjoint component to S.
         – Add the cheapest edge in S to E.
         – Add the resulting set of edges E to T.
3. The resulting set of edges T is the minimum spanning tree of G.
```

Boruvka's algorithm takes O(log *V)* iterations of the outer loop before termination, and runs in time O(*E*log *V),* where *E* is the number of edges, and *V* is the number of vertices in *G*.

Faster algorithms can be obtained by combining Prim's algorithm with Boruvka's.

**CHECK YOUR PROGRESS**

6. Define an optimal spanning tree.
7. What is the weight of a tree?
8. What is the working of the Boruvka's algorithm?
9. Define a 'component'.

## 3.4 PLANAR GRAPHS

A graph *G* is said to be *planar* if a geometric representation of *G* exists, which can be drawn on a plane such that no two of its edges intersect ('meeting' of edges at a vertex is not considered an intersection). A graph that cannot be drawn on a plane without a cross over between its edges is called a non-planar graph. A drawing of a geometric representation of a graph on any surface such that no edges intersect is called embedding.

Some planar graphs are presputed in Figure 3.30.

***Note:*** To show a that graph *G* is nonplanar you have to prove that all the possible geometric representations of *G*, cannot be embedded in a plane.

**Theorem 3.5:** The complete graph of five vertices is non-planar.

**Proof:** Let the five vertices in the complete graph be $v_1$, $v_2$, $v_3$, $v_4$ and $v_5$. By using the definition of the complete graph, you must have a circuit going from $v_1$ - $v_2$ - $v_3$ - $v_4$ - $v_5$ to $v_1$, i.e, a pentagon. This pentagon must divide the plane of the paper into two regions, one inside and the other, outside.

Since $v_1$ is to be connected to $v_3$ by means of an edge, this edge may be drawn inside or outside the pentagon (without intersecting the five edges drawn previously). Suppose you choose to draw a line from $v_1$ to $v_3$ inside the pentagon. In this case have to draw an edge from $v_2$ to $v_3$ and another one from $v_2$ to $v_5$. Since neither of these edges can be drawn inside the pentagon without crossing over the edge already drawn, you need to draw both these edges outside the pentagon. The edge connecting $v_3$ and $v_5$ cannot be drawn outside the pentagon without crossing the edge between $v_2$ and $v_4$. Therefore, $v_3$ and $v_5$ have to be connected with an edge inside the pentagon.

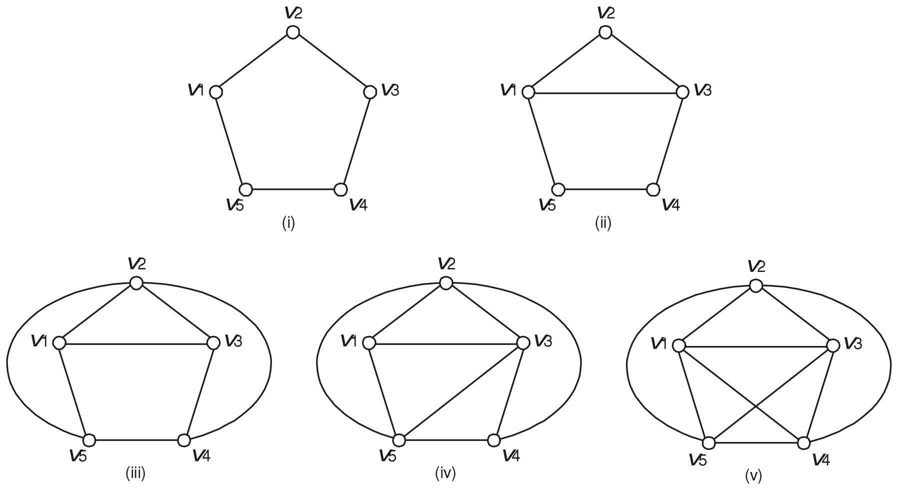


***Figure 3.30*** *Planar Graphs*

***Note:*** A complete graph is nothing but a simple graph in which every vertex is joined to every other vertex by means of an edge.

**Theorem 3.6:** Kurtowski's (Polish mathematician) second graph is also nonplanar. ($k_{3,3}$ is nonplanar).

***Note:*** In the plane, a continuous non-self intersecting curve whose origin and terminus coincide is said to be a Jordan curve. If *j* is a Jordan curve in the plane $\pi$, then $\pi - j$ is a union of two disjoint connected open sets called the interior and the exterior of *j*.

**Example 3.11:** Prove that $K_5$ is nonplanar.

**Solution:**

**Step 1.** Draw a circuit *c* on 5 vertices. This circuit *c* divides the plane into two regions called interior and exterior of *c* as shown in Figure 3.31:

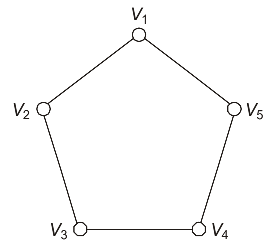


***Figure 3.31*** *Circuit C*

**Step 2.** Draw the edges $v_1v_3$, $v_1v_4$ in the interior as shown in Figure 3.32. You cannot draw any more edge in the interior of $c$, without intersecting any edge.

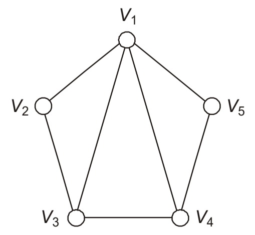


***Figure 3.32*** *Circuit c with edges in the interior*

Now, draw the edges $v_2v_5$, $v_2v_4$ in the exterior of $c$ as shown in Figure 3.33. But the edge $v_3v_5$ cannot be drawn in the interior or exterior of $c$, without intersecting the edge of $c$.

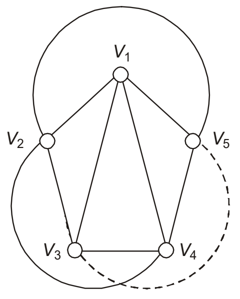


***Figure 3.33*** *Circuit c with interior and exterior edges*

Thus, $k_5$ is nonplanar.

In addition, you can prove that $k_{3,\,3}$ is nonplanar in the following manner:

Assume that $k_{3,3}$ is planar. Let the vertex of $k_{3,3}$ is $\{v_1,...,v_6\}$.

Let

$P = \{v_1, v_3, v_5\}$ and $Q = \{v_2, v_4, v_6\}$.

Let $C$ be the cycle $v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_1$. It is a Jordan curve. The other three edges $v_1v_4$, $v_2v_5$, $v_3v_6$ are chords of the cycle $C$. So, either the interior of $C$ or exterior of $C$ contains two of these three chords. Say there are two chords in Int c. These two chords must cross each other, which is a contradiction, hence $k_{3,3}$ is nonplanar.

***Contour:*** Let $G$ be a connected planar graph. A region of $G$ is the domain of the plane surrounded by edges of the graph such that any two points in it can be joined by a line not crossing any edge. The edges 'touching' a region contain a simple cycle called the contour of the region. Two regions are said to be adjacent if the contours of the two regions have atleast one edge in common. This is illustrated in Figure 3.34.

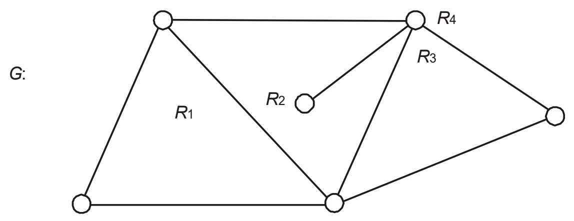


***Figure 3.34*** *Connected Planar Graph*

In a planar graph $G$: $R_i$, $i = 1, 2, 3, 4$, are the regions of $G$. Here, $R_4$ is the infinite region.

**Euler's Formula**

If $G$, a connected planar graph has $n$ vertices, $e$ edges and $r$ regions, then, $n - e + r = 2$

**Proof:** By induction on $e$, the number of edges:

If $e = 0$, then $G = K_1$( $\because$ $G$ is connected)

$\therefore$ $n = 1$ ; $r = 1$ (Infinite face) $\qquad$ $\therefore$ $n - e + r = 1 - 0 + 2 = 3$

If $e = 1$ then $n = 2$ ( $\because$ $G$ is connected) and $r = 1$ (Infinite face)

$\therefore n - e + r = 2 - 1 + 1 = 2$

$\therefore$ This result is true for $e = 0$ and $e = 1$.

Let us assume that this result is true for all the connected planar graphs on $(e - 1)$ edges.

Let $G$ be a connected planar graphs with $e$ edges.

**Case (*i*)** If $G$ is a tree with $e$ edges then $n = e + 1$

$\because$ Tree on $n$ vertices has $(n - 1)$ edges.

$r = 1$

$\therefore$ $n - e + r = e + 1 - e + 1 = 2.$

**Case (*ii*)** If $G$ is not a tree.

Since $G$ is connected, it contains cycles.

Let $e_1$ be an edge in some simple circuit of $G$.

Let $G_1$ be the graph obtained from $G$ by deleting the $e_1$, i.e., $G_1 = G - e_1$

Now, number of vertices in $G_1 = n$

Number of edges in $G_1 = e–1$

Number of regions in $G_1 = r–1$

Since $G_1$ has less then $e$ edges, the result is true for $G_1$ also.

$\therefore$ By induction hypothesis, $n_1 – e_1 + r_1 = 2$, where $n_1$ is the number of vertices, $e_1$ is the number of edges and $r_1$ is the number of regions of $G_1$ respectively.

$\therefore$ $n–(e–1) + r–1 = 2 \Rightarrow n – e + r = 2$.

$\therefore$ In all these cases, the result in true.

**Corollary:** If $G$ is a connected simple planar graph without loops and has $n$ vertices, $e \geq 2$ edges and $r$ regions, then $3/2\ r \leq e \leq 3n–6$.

**Proof:** If $r = 1$ then $3/2 \leq e \leq 3n – 6$ is true, since $e \geq 2$.

If $r > 1$. Let $k$ be the number of edges in the contours of the finite regions.

Since $G$ is simple, each region (finite) is bounded by atleast 3 edges.

Therefore $k \geq 3\ (r – 1)$ ...(3.4)

But, in a planar graph, an edge belongs to the contours of atmost two regions and atleast 3 edges touch the infinite region.

$\therefore$ $k \leq 2e – 3$ ...(3.5)

From equations (3.4) and (3.5), $3r–3 \leq k \leq 2e–3$

$\Rightarrow$ $3r – 3 \leq 2e – 3$

$\Rightarrow$ $3r \leq 2e \Rightarrow 3/2\ r \leq e$ ...(3.6)

Since $G$ is planar, $n – e + r = 2$, by Euler's Formula.

$\therefore$ $n – e + 2/3\ e \geq 2$ [$\because$ From equation (3.10) $r \leq 2/3\ e$]

$\Rightarrow$ $3n – 3e + 2e \geq 6$

$\Rightarrow$ $– e \geq – 3n + 6$

$\Rightarrow$ $e \leq 3n – 6$ ...(3.7)

From equations (3.6) and (3.7), $3/2r \leq e \leq 3n – 6$

**Example 3.12:** Prove that $K_5$ is nonplanar.

**Solution:** Suppose $K_5$ is planar, then by the above corollory, $e \leq 3n – 6$. In $K_5$ $n = 5$, $e = 10$;

$\therefore$ $10 \leq 3 \times 5 – 6 = 9$, which is absurd. $\therefore$ $K_5$ is nonplanar.

**Remark:** $K_5, K_{3,3}$ are called Kuratowski's first graph, second graph respectively.

**Corollary:** If $G$ is a simple connected planar graph on $n$ vertices, $e$ edges and $r$ regions and does not contain any triangle, then $2r \leq e \leq (2n – 4)$.

**NOTES**

***Subdivision:*** A subdivision of a graph *G* is obtained by inserting vertices of degree 2 into the edges of *G*, as shown in Figure 3.35:

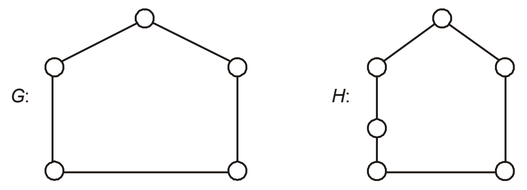


***Figure 3.35*** *Subdivision of Graph G*

*H* is the subdivsision of *G*.

***Kuratowski theorem:*** A graph is planar if it contains no subgraph that is isomorphic or is a subdivision of $K_5$ or $K_{3,3}$.

**CHECK YOUR PROGRESS**

10. What is a non-planar graph?
11. Define a complete graph?

## 3.5 SUMMARY

In this unit, you have learned that:

- In graph theory, a tree refers to a graph in which two vertices are attached by only one path.
- Trees are generally open graphs.
- In a tree, every edge is a cut-edge.
- Traversal algorithm refers to the systematic method for visiting every vertex of an ordered rooted tree.
- Arithmetic expression and compound proposition can be represented by using ordered rooted trees.
- A merge sort proceeds by repeatedly splitting lists into two sub-lists of equal size (nearly) to a point such that each sub-list consists of only one element.
- A spanning tree of a connected graph using either of the two methods:
  - o Depth-First Search (DFS)
  - o Breadth-First Search (BFS)
- A graph G is considered to be planar if there exists some geometric representation of G which can be drawn on a plane in such a manner that no two of its edges intersect ('meeting' of edges at a vertex is not considered an intersection).
- Euler's Formula states that in case G, a connected planar graph has n vertices, e edges and r regions, then, n – e + r = 2

## 3.6 KEY TERMS

NOTES

- **Tree:** A connected acyclic graph $G$ is called a tree.
- **Vertex:** It refers to a point of intersection in any diagram containing two or more edges.
- **Rooted tree:** In a directed tree a particular vertex is called a root if that vertex is of degree zero.
- **Balanced tree:** A rooted $k$-ary tree of height $h$ is balanced if all the leaves are at level $h$ or $(h - 1)$.

## 3.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A graph $G$, which has no cycles is called an acyclic graph.
2. A tree together with its root produces a graph called a rooted tree.
3 A vertex of a rooted tree is known as a leaf.
4. A tree is called as a binary tree if there is at least one vertex with degree 2 and the remaining vertices are of degree 1 or 2.
5. If in a rooted tree, each internal vertex is assigned to a decision with a sub-tree at the vertices, then each possible outcome of the decision is called a decision tree.
6. The minimum weight spanning tree is called an optimal spanning tree.
7. The weight of a tree is the sum of weights of the edges in the tree and is denoted by $wt(T)$.
8. Boruvka's algorithm finds a minimum spanning tree in a weighted graph.
9. Every vertex or set of connected vertices is termed as a 'component'.
10 A graph that cannot be drawn on a plane without a cross-over between its edges is called a non-planar graph.
11 A complete graph is nothing but a simple graph in which every vertex is joined to every other vertex by means of an edge.

## 3.8 QUESTIONS AND EXERCISES

**Short-Answer Questions**

1. Briefly explain how trees are helpful in the process of merge sort.
2. State the three types of algorithms used to find the optimal spanning tree.
3. What is a fundamental circuit?
4. Diagrammatically state what a contour.

NOTES

**Long-Answer Questions**

1. In a graph theory, prove that $G$ has ($n$-1) edges, if $G$ is a tree with $n$ number of vertices.
2. Explain the concept of traversal of a tree.
3. Discuss the algorithms used to find optimal spanning trees.
4. 'The complete graph of five vertices is non-planar.' Prove it.

## 3.9 FURTHER READING

Lipschutz, Seymour and Lipson Marc. *Schaum's Outline of Discrete Mathematics,* 3rd edition. New York: McGraw-Hill, 2007.

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms.* Hyderabad: Orient BlackSwan, 2008.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

Brassard, Gilles and Paul Bratley. *Fundamentals of Algorithms*. New Delhi: Prentice Hall of India, 1995.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. New Jersey: Pearson, 2006.

Baase, Sara and Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. New Jersey: Pearson, 2003.

Mott, J.L. *Discrete Mathematics for Computer Scientists*, 2nd edition. New Delhi: Prentice-Hall of India Pvt. Ltd., 2007.

Liu, C.L. *Elements of Discrete Mathematics*. New Delhi: Tata McGraw-Hill Publishing Company, 1977.

Rosen, Kenneth. *Discrete Mathematics and Its Applications*, 6th edition. New York: McGraw-Hill Higher Education, 2007.

# UNIT 4 RECURSION

NOTES

**Structure**

## 4.0 INTRODUCTION

Recursion is a concept prevalent in mathematics and computer science. It is a method of defining functions in which the function being defined is applied within its own definition. This specifically means defining an infinite statement using finite components. The term is also used more generally to describe a process of repeating objects in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other, the nested images that occur are a form of infinite recursion.

This in plain English means that recursion is the process a procedure goes through when one of the steps of that procedure involves re-running the procedure. A procedure that goes through recursion is recursive, that is, if one of the steps that makes up the procedure calls for a new running of the procedure. A recursive procedure must complete each of all its steps. Even if a new running is called for in one of its steps, each running must run through the remaining steps.

In this unit, you will learn about merge sort, insertion sort, bubble sort and selection sort, binary and decimal numbers, recursion and recurrent relations and recursive procedures.

## 4.1 UNIT OBJECTIVES

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

- Understand the concepts of merge sort, bubble sort, insertion sort and selection sort
- Describe binary and decimal numbers
- Convert binary numbers to decimal numbers and vice versa
- Explain the concepts of recursion and recurrence relations
- Describe the various features of recursive procedures

## 4.2 MERGESORT

The `Mergesort` algorithm basically works according to a divide and conquer strategy in which the sequence is divided into two halves. Each half is independently sorted and then both halves are merged to make a combine sequence. In this process, the validity of input data required in `Mergesort` is as follows:

- Check the input sequences. If there is only one element then the `Mergesort` operation is not performed.
- The input sequences are separated into two halves.
- Sort the input sequences.
- Merge both sorted input sequences to generate the result.

In the merging process, the elements of two arrays are combined, creating a new array. The algorithm is based on the merging process where all the elements are copied in one array and kept in the separate new array. Then it adds the second array to the new array. After combining the sorted array a `Mergesort` array is created. For example, the two arrays `A[5]` and `B[3]` are manipulated and then merged to create a new array. The newly created array, namely C, will have 5+3=8 elements. The required steps are as follows:

- Compare the very first elements of both `A[0]` and `B[0]`. If `A[0] < B[0]` then the value of `A[0]` is shifted to `C[0]`. Then the size of both arrays `[Arrays A and C]` current pointers are increased by one.
- The elements of array A and array B are compared where the pointers are pointing, that is, the first element of array A and the null element of B, i.e., `A[1]` and `B [0]`.
- If `B[0]<A[1]` then `B[0]` is moved to `C[1]`. The current pointer of B is incremented to point the next element in array B.

The following algorithm checks the sequences of validation of arrays:

```
Function Mergesort(M1, M2)
{
list A ← Empty
while (neither M1 nor M2)
{
compare first items of M1 and M2
remove smaller of the M1 and M2 from the list
add to end of A
}
catenate remaining list to end of A
return A
}
```

**`Mergesort` problem:** Sort a sequence of given *n* elements in a non-decreasing way. It follows the **DCC** mechanism that represents **D**ivide, **C**onquer and **C**ombine:

**Divide**: Divides the *n* element sequence that is sorted into two subsequences of *n*/2 elements.

**Conquer**: Sorts by using `Mergesort` the two recursive subsequences.

**Combine**: Merges both subsequences to produce the sorted result.

The required steps in the `Mergesort` algorithm are as follows:

**Input**: Sort a sequence of *n* numbers that is stored in an array.

**Output:** Produce an ordered sequence of n numbers.

The following algorithm is applied in `mergesort` mechanism:

```
Mergesort(A,m,n) //It sorts A[m…n] by divide and conquer
method
Step 1: if m<n
Step 2: then r←[(m+n)/2]
Step 3: Mergesort (A,m,r)
Step 4: Mergesort (A, r+1, n)
Step 5: Merge(A,m,n,r) //This step merges A[m…n] with
A[r+1…n]
Merge (A,m,n,p)
```

NOTES

```
Step 1: n1←n - m+1
Step 2: n2←p - n
Step 3: for i ← 1 to n1
Step 4: do L[i] ←A[m+i-1]
Step 5: for j ←1 to n2
Step 6: do R[j] ←A[n+j]
Step 7: L[n1+1] ←∞
Step 8: R[n2+1] ←∞
Step 9: i←1
Step 10: j←1
Step 11: for k←m to p
Step 12: do if L[i]<=R[j]
Step 13: then A[k]←L[i]
Step 14: i← i+1
Step 15: else A[k] ←R[j]
Step 16: j← j+1
```

In the above algorithm, `L[i]` and `R[j]` are the smallest elements of `L` and `R` that are not copied back into `A`. Figure 4.1 shows the `Mergesort` process that is based on this algorithm:

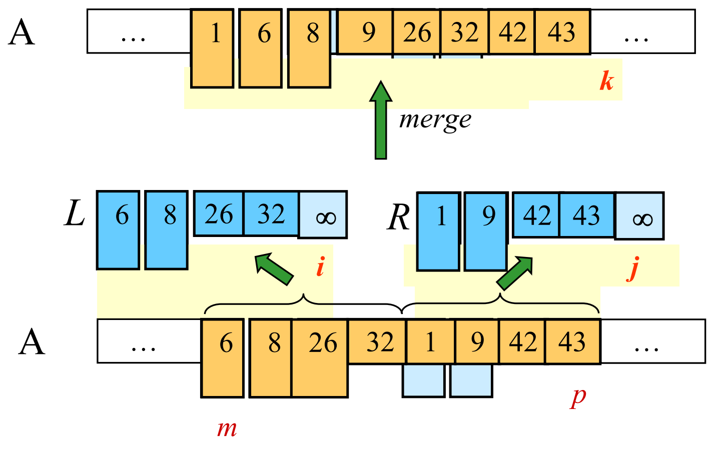


***Figure 4.1*** *Elements sorted [1, 6, 8, 9, 26, 32, 42, 43] using* `Mergesort`

**Analysis of Mergesort Algorithm**

In Figure 4.2, an array A is taken in which there are eight elements. The operation of `Mergesort` on the array A is [5, 2, 4, 7, 1, 3, 2, 6]. The length of the sorted sequences is merged as the steps required in algorithm from bottom to top.

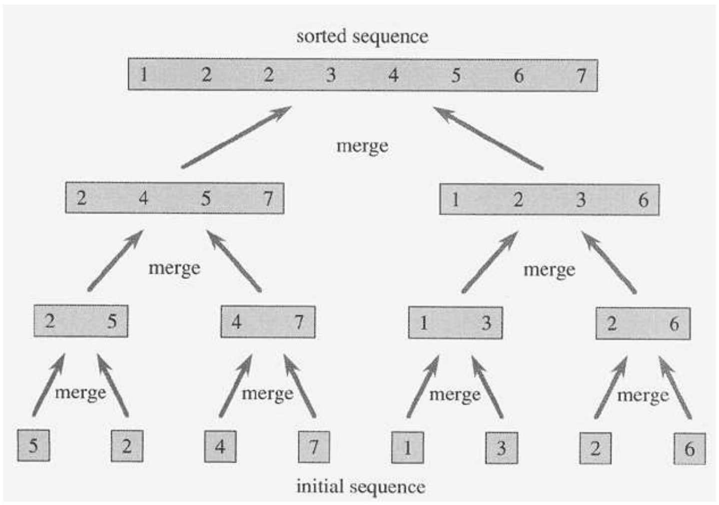


***Figure 4.2*** *A Mergesort Algorithm*

## Implementation of `Mergesort` for Two Vectors of Seven Elements

```
/*——————————— START OF PROGRAM ———————————*/
#include <stdio.h>
#include <conio.h>
void Mergesort(int [], int [], int [], int, int);
void main()
{
    int A_Array[50], B_Array [50], C_Array [100], m, n,
i;
    printf("\n Enter the array elements for first array
[max 50]: ");
    scanf("%d", &m);
     printf("\mEnter the array elements in ascending
order:");
for (i=0; i<m; i++)
        scanf("%d", &A_Array[i]);
    printf("\nEnter the array elements for second array
    [max 50]: ");
    scanf("%d", &n);
    printf("Enter the array elements in ascending order:");
    for (i=0; i<n; i++)
        scanf("%d", &B_Array[i]);
    Mergesort(A_Array, B_Array, C_Array, m, n);
    printf("\n The sorted array is : ");
    for (i=0; i<m+n; i++)
        printf("%d\n", C_Array[i]);
   }
```

NOTES

```
void Mergesort(int A_Array[], int B_Array[], int C_Array[],
int m, int n)
{
    int a_ele=0, b_ele=0, c_ele=0;
    for (a_ele =0, b_ele=0, c_ele =0; a_ele<m && b_ele<n;)
    {
        if (A_Array[a_ele]< B_Array[b_ele])
//Check the elements of A_Array are less than elements of
B_Array
           C_Array[c_ele++] = A_Array[a_ele++];
//Assign the values of C_Array in A_Array otherwise B_Array
        else
           C_Array[c_ele++] = B_Array [b_ele++];
    }
    if (a_ele<m)
        while (a_ele<m)
           C_Array[c_ele++] = A[a_ele++];
    else
        while (b_ele<n)
           C_Array[c_ele++] = B_Array[b_ele++];
}
```

The arrays `A_Array` and `B_Array` are the input arrays that contain elements in ascending order. Their sizes are `m` and `n` respectively. The `C_Array` is the output array containing the elements from the two combined arrays in sorted order.

The result comes in the following way:

```
Enter the array elements for first array [max 50]: 3
Enter the array elements in ascending order:
4
8
10
Enter the array elements for second array [max 50]:4
Enter the array elements in ascending order:
3
5
7
9
```

**Output** : The sorted array is:

```
3
4
5
7
8
9
10
```

## 4.3 INSERTION SORT

Insertion sort refers to a simple sorting algorithm. In it, the sorted array (or list) is built one entry at a time. As compared to more advanced algorithms, such as quick sort, heap sort or merge sort, it is less efficient on large lists. However, insertion sort has many advantages, such as:

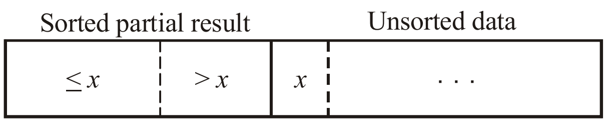
- Its implementation is simple.
- It is efficient for (quite) small data sets.
- It is efficient for data sets that are already substantially sorted. The time complexity is O($n + d$), where $d$ is the number of inversions.
- It is more efficient in practice as compared to most other simple quadratic i.e., O($n^2$) algorithms, such as selection sort or bubble sort. The average running time of insertion sort is $n^2/4$. Further, in the best case scenario, the running time is linear.
- It is stable. In other words, it does not change the relative order of elements with equal keys.
- It is in place, i.e., it only requires a constant amount O(1) of additional memory space.
- It is online, i.e., it can sort a list as it receives it.

Most people while sorting—ordering a deck of cards, for example—use the insertion sort like method.

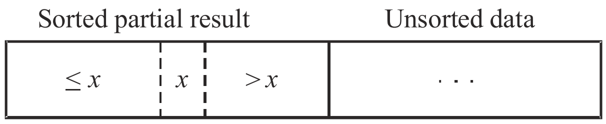
In abstract terms, each iteration of insertion sort removes an element from the input data and then inserts it into the correct position in the list that is already sorted. The process continues till all input elements are inserted. The element to be removed from the input is chosen arbitrarily. Almost any choosen algorithm can be used for this.

Sorting is typically done in-place. The resulting array after $k$ iterations has the property where the first $k$ entries are sorted. In each iteration, the first remaining entry of the input is removed, inserted into the result at the correct position, thus

NOTES

extending the result: 

becomes



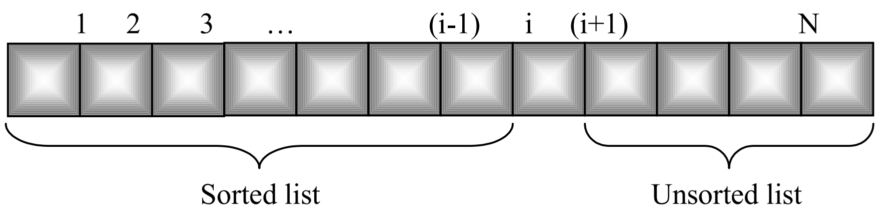
with each element greater than *x* copied to the right as it is compared against *x*.

Consider a function called *Insert,* which is designed for inserting a value into a sorted sequence at the beginning of an array. It starts operating at the end of the sequence and shifts each element one place to the right unless an appropriate position becomes available for the new element. This function has a problem. It can overwrite the value that is stored just after the sorted sequence in the array.

For performing an insertion sort, you need to begin at the leftmost element of the array and invoke *Insert* in order to insert each element which is encountered into its correct position. The ordered sequence of inserted elements is stored at the beginning of the array. These elements are stored in the set of indices already examined. Each insertion overwrites a single value, i.e., the value which is being inserted.

**Algorithm for Insertion Sort**

```
Procedure InsSort(A,N).
[Where A is a vector and N denotes number of elements in
the vector.
I,J acts as indices of vector A and Max].
1. [Initialize I]
    I = 0
2. [Perform sort]
    REPEAT THRU Step 6 until I < N
3. [Initialize Max,J]
    Max = A[I]
      J = I
4. [Backtrack and change]
    REPEAT WHILE J > 0 AND Max < A[J – 1]) /*Backtrack */
       A[J] = A[J – 1]
       J = J – 1
5. [Assign Max]
     A[J] = Max
6. [Increment I]
    I = I + 1
7. [Finished]
    RETURN.
```



NOTES

**Example 4.1:** Sort the elements 16, 19, 4,1, 20, 2 using Insertion sort.

**Solution:**

| Set of elements | 2nd Iteration | 3rd Iteration | 4th Iteration | 5th Iteration | 6th Iteration |
|---|---|---|---|---|---|
| 16 | **16** | **4** | **1** | **1** | **1** |
| 19 | 19 | **16** | **4** | **4** | **2** |
| 4 | **4** | 19 | **16** | **16** | **4** |
| 1 | **1** | **1** | 19 | **19** | **16** |
| 20 | **20** | **20** | **20** | 20 | **19** |
| 2 | **2** | **2** | **2** | **2** | 20 |

From the insertion sort algorithm, sorting is achieved by each iteration as shown in the diagram. In each row, the elements are in sorted order relative to each other above the element within a block; below this element, the elements are not affected.

***Analysis of Insertion sort:*** The time complexity of the insertion sort is O($N^2$), where '$N$' is the number of elements in the array. On an average, the number of interchanges required is ($N^2$/4) and in worst cases about ($N^2$/2). The insertion sort is highly efficient if the array is already in almost sorted order.

### Implementation of Insertion Sort for a Vector having Numbers as its Elements

```
#include<stdio.h>
#define MAX 100
typedef VECTOR[MAX];
void InsSort(VECTOR a, int n)
{int i, j, Max;
 for(i = 0; i < n; ++i)
 {
     Max = a[i];
     j = i;
     while(j > 0 && Max < a[j – 1]) /*backtrack */
     {
     a[j] = a[j – 1];
     j = j – 1;
     }
     a[j] = max;
 }
}
void main()
{VECTOR a = {5, 4, 3, 2, 1};
```

```
 int i;
 InsSort(a, 5);
 for(i = 0; i < 5; ++i)
     printf("%d ", a[i]);
}
```

**Output:** 1 2 3 4 5

**Implementation of Insertion Sort for a Vector having Strings as its Elements**

```
#include<stdio.h>
#include<string.h>
#define MAXROWS 10
#define MAXCOLS 20
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void InsSort(STRINGS A,int N)
{
 int I, J;
 STRING MaxStr;
 for(I = 0; I < N; ++I)
 {
     strcpy(MaxStr, A[I]);
     J = I;
     while(J > 0 && strcmp(MaxStr, A[J - 1])<0) /*backtrack
*/
     {
       strcpy(A[J], A[J - 1]);
       J = J - 1;
     }
     strcpy(A[J], MaxStr);
 }
}
void main()
{
 STRINGS A = {"EE", "AA", "BB", "DD", "CC"};
 int i;
 InsSort(A, 5);
 for(i = 0; i < 5; ++i)
    printf("%s", A[i]);
}
```

**OUTPUT:** AA BB CC DD EE

The array which is already sorted is considered the best case input. In the given case, insertion sort has a linear running time, i.e., O($n$). During each iteration, the first remaining element of the input would only be compared with the rightmost element of the sorted subsection of the array.

An array sorted in the reverse order is the worst case input. In the given case, insertion sort has a quadratic running time, i.e., O($n^2$). Every iteration of the inner loop scans and shifts the entire sorted subsection of the array before the next element is inserted. The average case is also quadratic. That is why the insertion sort is not practical for sorting large arrays. However, for sorting arrays having less than ten elements, insertion sort is one of the fastest algorithms.

**CHECK YOUR PROGRESS**

1. What is the Mergesort algorithm based on?
2. List out any two advantages of Insertion sort.
3. What is the role of the insert function?

## 4.4 BUBBLE SORT AND SELECTION SORT

In the fields of computer science and mathematics, a sorting algorithm refers to an algorithm whose function is to put elements of a list in a certain order. The numerical and lexicographical orders are the most used orders. In order to optimize the use of other algorithms, such as search and merge algorithms, efficient sorting is essential, as these algorithms require sorted lists to work correctly. Sorting is often used to canonicalize data and to produce human-readable output. The output must meet the following two conditions:

- The output should be in non-decreasing order (each element should not be smaller than the previous element according to the desired total order).
- The output should be a permutation or reordering of the input.

Since the beginning of computing, the sorting problem has greatly attracted the attention of researchers, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, the analysis of bubble sort was done as early as 1956. Many consider it a solved problem. However, the invention of new sorting algorithms has not stopped. Library sort, for example, was first published in 2004. Sorting algorithms are taught in introductory computer science classes. Students are introduced to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs and lower bounds.

Sorting is a method of arranging keys in a file in the ascending or descending order. Sorting makes handling of records in a file easier.

NOTES

Sorting can be classified into the following two types:

**Internal sorting**: Sorting of records in a file, which is stored in the main memory.

**External sorting**: Sorting of records in a file, which is stored in the secondary memory. Some sorting techniques are as follows:

- Bubble sort
- Insertion sort
- Selection sort
- Quick sort
- Tree sort
- Arrangement of elements in a list according to the increasing (or decreasing) values of some key field of each element.
- Sorting will be useful to search, insert or delete a data item in a list.

There are various methods for sorting explained in the following sections.

### 4.4.1 Bubble Sort

Bubble sort comes under the category of exchange sort technique.

- Consider an array A has *n* elements `A[0]` to `A[n - 1]`. The array is to be sorted in the ascending order.
- Compare `A[0]` and `A[1]` and arrange such that `A[0] < A[1]`. Then compare `A[1]` and `A[2]` and arrange such that `A[1] < A[2]`. Repeat this process till the largest element is bubbled to the *n*th position.
- Since the largest value is now in the last position as required for the ascending order, consider the first $(n – 1)$ elements. Repeat the above process as to bubble the next largest value to $(n – 1)$th position. Then consider the first $(n – 2)$ elements and in this way proceed to bubble till all the elements are bubbled to their respective positions. Then sorting will be completed.

**Algorithm for Bubble Sort or Exchange Sort**

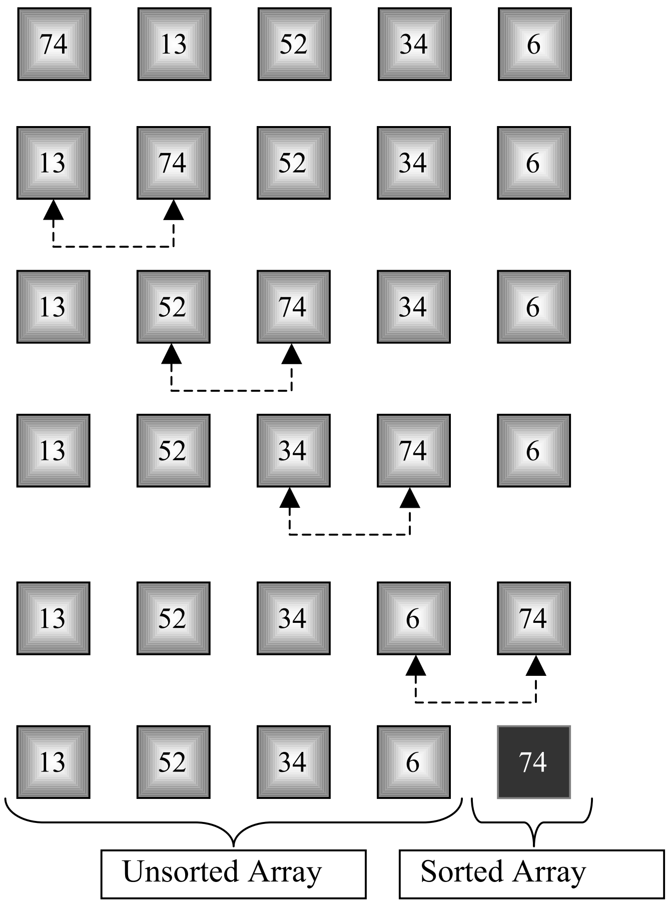
```
BUBBLE_SORT(B,N). Where B is a vector having N elements
1. [Initialization]
  Last = N (entire list assumed unsorted at this point)
2. [Loop on I index]
  REPEAT THRU STEP 5 FOR I = 1 TO N - 1 DO
3. [Initialize exchanges counter for this pass]
  EXS = 0
4. [Compare the unsorted pairs]
  REPEAT FOR J = 1 TO Last - 1 DO
   IF B[J] < B[J+1] THEN
     B[J] = B[J+1]
  EXS = EXS + 1
```

```
5. [Check whether any exchanges occur or?]
  IF EXS = 0 THEN
RETURN (Sorting finished)
  ELSE
Last = Last - 1(reduce the size of unsorted list)
6. [maximum number of passes finished]
  RETURN
```

NOTES

**Example 4.2:** Sort the elements 74, 13, 52, 34, 6 using bubble sort.

**Solution:**



Apply the same procedure for the unsorted array and repeat the same process until the elements are not exchanged in any of the pass, then result will be the sorted list: 6, 13, 34, 52, 74.

**Implementation of Bubble Sort to Sort Strings of Vector/Array**

**Program for Bubble Sort of Numbers**

```
/*——————————START OF PROGRAM——————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define MAXROWS 10
```

```
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void bub_sort(STRINGS a,int n)
{
   int i,j;
for(i = 0;i <n - 1; ++i)
{
int pass = 0;
for(j = 0; j < n - 1 - i; ++j)
{
if(strcmp(a[j], a[j + 1]) > 0)
      {
   STRING temp;
   strcpy(temp,a[j]);
        strcpy(a[j], a[j + 1]);
        strcpy(a[j + 1],temp);
        pass = 1;
      }
 }
      if(pass == 0)
   break;
   }
}
void main()
{
   STRINGS a = {"EE","BA","AB","CD","AA"};
int i;
clrscr();
bub_sort(a,5);
for(i = 0; i < 5; ++i)
printf("%s ",a[i]);
}
/*————————END OF PROGRAM————————*/
OUTPUT: AA AB BA CD EE
```

### Implementation of Bubble Sort to Sort Integers of a Vector/Array

```
/*——————START OF PROGRAM————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
```

```
typedef int VECTOR[MAXCOLS];
void bub_sort(VECTOR a,int n)
{
    int i, j;
    for(i = 0; i < n - 1; ++i)
{
       int pass = 0;
       for(j = 0; j < n - 1 - i; ++j){

if(a[j] > a[j + 1])
{
int temp;
  temp = a[j];
  a[j] = a[j + 1];
  a[j + 1] = temp;
  pass = 1;
}
    }
if(pass == 0)
break;
}
}
void main()
{
VECTOR a = {5, 4, 3, 2, 1};
int i;
bub_sort(a, 5);
for(i = 0; i < 5; ++i)
printf("%d ", a[i]);
}
/*——————————END OF THE PROGRAM——————————*/
OUTPUT: 1 2 3 4 5
```

NOTES

### 4.4.2 Selection Sort

Selection sort is a simple sorting technique to sort a list of elements. This method helps to find the smallest value in the array. This is exchanged with the first element. The next smallest is found and exchanged with the second element. This is continued till all elements are completed. A disadvantage of selection sort is that its running time depends only slightly on the amount of order already in the given list of elements.

NOTES

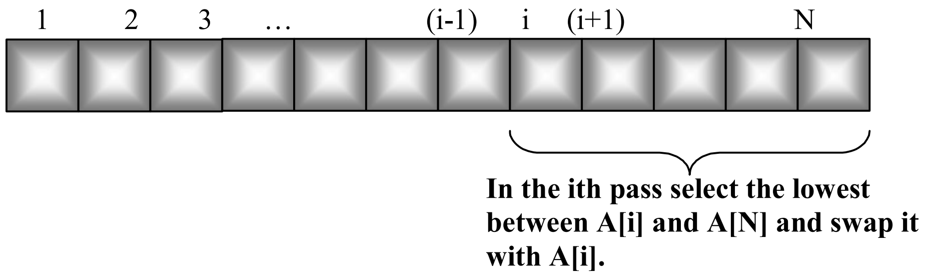
**SELECTION_SORT**(A,N)

```
[Where A is a vector having N elements]
1.[Loop on I index]
   REPEAT THRU Step 4 FOR I = 1, 2,..., N ‟ 1
2.[Initially assume minimum index is in I]
   Mindex = I
3. [For each pass, get small value]
  REPEAT FOR J = I + 1 to N
      IF A[MIndex] >A[J] THEN
         Mindex = J
4.[Interchange Elements]
   IF Mindex <> I THEN
      A[I]A[Mindex]
5.[Sorted values will be returned]
   RETURN
```

**Explanation:** In this algorithm, for each `I` to `N – 1`, exchange `A[I]` with the minimum element in the array `A[I],...,A[N]`. As the index `I` travels from left to right, the elements to its left are in their final position in the array and will not be touched again, so the array is fully sorted when I reaches the right end.

**Example 4.3:** Sort the elements 16, 19, 4, 1, 20, 2 using selection sort.

**Solution:**


1
2
3
...
(i-1)
i
(i+1)
N
In the ith pass select the lowest between A[i] and A[N] and swap it with A[i].

| Set of elements | **1st Iteration** | 2nd Iteration | 3rd Iteration | 4th Iteration | 5th Iteration |
|---|---|---|---|---|---|
| 16 | 1 | 1 | 1 | 1 | 1 |
| 19 | **19** | 2 | 2 | 2 | 2 |
| 4 | **4** | **4** | 4 | 4 | 4 |
| 1 | **16** | **16** | **16** | 16 | 16 |
| 20 | **20** | **20** | **20** | **20** | 19 |
| 2 | **2** | **19** | **19** | **19** | 20 |

**Implementation of Selection Sort to Sort Values of a Vector/Array**

**Program for Selection Sort of Numbers**

```
/*————————————START OF PROGRAM——————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 10
```

```
typedef int VECTOR[MAXCOLS];
void sel_sort(VECTOR a, int n)
{
    int i, j, flag, index;
for(i = 0; i < n – 1; ++i)
{
  index = i;
  Flag = 0;
for(j = i + 1; j < n; ++j)
    {
if(a[index] > a[j])
       {
   index = j;
   flag = 1;
}
 }
   if(flag)
   {
 int temp;
 temp = a[i];
     a[i] =a[index];
     a[index] = temp;
   }
}
void main()
{
    VECTOR a = {5, 4, 3, 2, 1};
    int i;
    sel_sort(a, 5);
for(i = 0; i < 5; ++i)
       printf("%d ", a[i]);
}
/*——————————END OF PROGRAM——————————*/
```

**OUTPUT**: 1 2 3 4 5

NOTES

### Implementation of Selection Sort to Sort Strings of Vector/Array

```
/*——————————START OF PROGRAM————————————*/
#include<stdio.h>
#include<conio.h>
#define MAXCOLS 20
#define MAXROWS 10
```

```
typedef char STRINGS[MAXROWS][MAXCOLS];
typedef char STRING[MAXCOLS];
void sel_sort(STRINGS a, int n)
{
int i, j, flag, index;
for(i = 0; i < n - 1; ++i)
{
  flag = 0, index = i;
       for(j = i + 1; j < n; ++j){
        if(strcmp(a[index], a[j]) > 0)
       {
          index = j;
          flag = 1;
       }
    }
   if(flag)
   {
    STRING temp;
 strcpy(temp, a[i]);
     strcpy(a[i], a[j]);
     strcpy(a[j], temp);
    }

}
}
void main()
{
STRINGS a = {"EE", "BB", "EA", "DD", "AA"};
    int i;
    sel_sort(a, 5);
    for(i = 0; i < 5; ++i)
       printf("%s ", a[i]);
}
/*———————————END OF THE PROGRAM——————————*/
OUTPUT: AA BB DD EA EE
```

## Check Your Progress

4. What do you understand by sorting?
5. What is internal sorting?
6. Name any two sorting techniques.

## 4.5 BINARY AND DECIMAL NUMBERS

### 4.5.1 Binary Number System

**NOTES**

A number system that uses only two digits, 0 and 1, is called the binary number system. The binary number system is also called a base two system. The two symbols 0 and 1 are known as bits or binary digits.

The binary system groups numbers by two and by powers of two, shown in Figure 4.3. The word binary comes from a Latin word meaning two at a time.

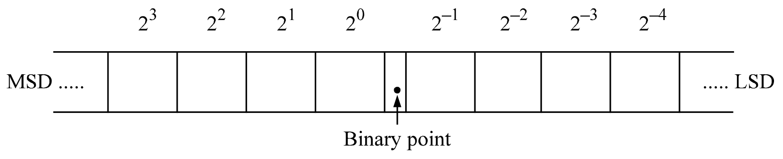


***Figure 4.3*** *Binary Position Values*

The weight or place value of each position can be expressed in terms of 2, and is represented as $2^0$, $2^1$, $2^2$, etc. The least significant digit has a weight of $2^0$ (= 1). The second position to the left of the least significant digit is multiplied by $2^1$ (= 2). The third position has a weight equal to $2^2$ (= 4). Thus, the weights are in the ascending powers of 2 or 1, 2, 4, 8, 16, 32, 64, 128, etc.

The numeral $10_{two}$ or $10_2$ (one, zero, base two) stands for two, the base of the system. In binary counting, single digits are used for none and one. Two-digit numbers are used for $10_{two}$ and $11_{two}$ (2 and 3 in decimal numerals). For the next counting number, $100_{two}$ (4 in decimal numerals) three digits are necessary. After $111_{two}$ (7 in decimal numerals), four-digit numerals are used until $1111_{two}$ (15 in decimal numerals) is reached, and so on. In a binary numeral, every position has a value 2 times the value of the position to its right.

A binary number with 4 bits is called a *nibble* and a binary number with 8 bits is known as a *byte*.

For example, the number $1011_2$ actually stands for the following representation:

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$
$$\therefore \quad 1011_2 = 8 + 0 + 2 + 1 = 11_{10}$$

In general,

$$[b_n, b_{n-1} \ldots b_2, b_1, b_0]_2 = b_n 2^n + b_{n-1} 2^{n-1} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Similarly, the binary number 10101.011 can be written as,

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | . | 0 | 1 | 1 |
| $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | . | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ |
| (MSD) | | | | | | (LSD) | | |

$$\therefore \quad 10101.011_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 16 + 0 + 4 + 0 + 1 + 0 + 0.25 + 0.125 = 21.375_{10}$$

NOTES

In each binary digit, the value increases in powers of two starting with 0 to the left of the binary point and decreases to the right of the binary point starting with power –1.

**Use of Binary Number System in Digital Computers**

The binary number system is used in digital computers because all electrical and electronic circuits can be made to respond to the two-state concept. A switch, for instance, can be either opened or closed, only two possible states exist. A transistor can be made to operate either in cut-off or saturation; a magnetic tape can be either magnetized or non-magnetized; a signal can be either High or Low; a punched tape can have a hole or no hole. In all of these illustrations, each device is operated in any one of the two possible states and the intermediate condition does not exist. Thus, 0 can represent one of the states and 1 can represent the other. Hence, binary numbers are convenient to use in analysing or designing digital circuits.

### 4.5.2 Decimal Number System

The number system which utilizes ten distinct digits, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 is known as decimal number system. It represents numbers in terms of groups of ten, as shown in Figure 4.4.

We would be forced to stop at 9 or to invent more symbols if it were not for the use of positional notation. It is necessary to learn only 10 basic numbers and positional notational system in order to count any desired figure.

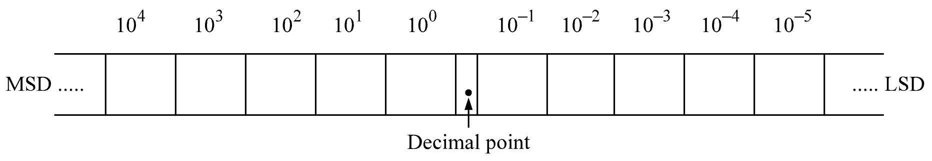


***Figure 4.4*** *Decimal Position Values*

The decimal number system has a base or radix of 10. Each of the ten decimal digits 0 through 9 has a place value or weight depending on its position. The weights are units, tens, hundreds and so on. The same can be written as the power of its base as $10^0$, $10^1$, $10^2$, $10^3$... etc. Thus, the number 1993 represents quantity equal to 1000 + 900 + 90 + 3. Actually, this should be written as $\{1 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 3 \times 10^0\}$. Hence, 1993 is the sum of all digits multiplied by their weights. Each position has a value 10 times greater than the position to its right.

For example, the number 379 actually stands for the following representation.

$$\begin{array}{lll} 100 & 10 & 1 \\ 10^2 & 10^1 & 10^0 \\ 3 & 7 & 9 \end{array}$$

$$3 \times 100 + 7 \times 10 + 9 \times 1$$

$$\begin{aligned} \therefore \quad 379_{10} &= 3 \times 100 + 7 \times 10 + 9 \times 1 \\ &= 3 \times 10^2 + 7 \times 10^1 + 9 \times 10^0 \end{aligned}$$

In this example, 9 is the least significant digit (LSD) and 3 is the most significant digit (MSD).

**Example 4.4:** Write the number 1936.469 using decimal representation.

**Solution:**
$$\begin{aligned} 1936.469_{10} &= 1 \times 10^3 + 9 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} \\ &\quad + 6 \times 10^{-2} + 9 \times 10^{-3} \\ &= 1000 + 900 + 30 + 6 + 0.4 + 0.06 + 0.009 \\ &= 1936.469 \end{aligned}$$

It is seen that powers are numbered to the left of the decimal point starting with 0 and to the right of the decimal point starting with –1.

The general rule for representing numbers in the decimal system by using positional notation is as follows:

$$a_n a_{n-1} \ldots a_2 a_1 a_0 = a_n 10^n + a_{n-1} 10^{n-1} + \ldots a_2 10^2 + a_1 10^1 + a_0 10^0$$

Where $n$ is the number of digits to the left of the decimal point.

### 4.5.3 Binary to Decimal Conversion

A binary number can be converted into decimal number by multiplying the binary 1 or 0 by the weight corresponding to its position and adding all the values.

**Example 4.5:** Convert the binary number 110111 to decimal number.

**Solution:**
$$\begin{aligned} 110111_2 &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1 \\ &= 32 + 16 + 0 + 4 + 2 + 1 \\ &= 55_{10} \end{aligned}$$

We can streamline binary to decimal conversion by the following procedure:

**Step 1:** Write the binary, i.e., all its bits in a row.

**Step 2:** Write 1, 2, 4, 8, 16, 32, ..., directly under the binary number working from right to left.

**Step 3:** Omit the decimal weight which lies under zero bits.

**Step 4:** Add the remaining weights to obtain the decimal equivalent.

The same method is used for binary fractional number.

NOTES

**Example 4.6:** Convert the binary number 11101.1011 into its decimal equivalent.

**Solution:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Step 1:** | 1 | 1 | 1 | 0 | 1 | . | 1 | 0 | 1 | 1 |
| | | | | | | ↑ Binary Point | | | | |
| **Step 2:** | 16 | 8 | 4 | 2 | 1 | . | 0.5 | 0.25 | 0.125 | 0.0625 |
| **Step 3:** | 16 | 8 | 4 | 0 | 1 | . | 0.5 | 0 | 0.125 | 0.0625 |

**Step 4:** $16 + 8 + 4 + 1 + 0.5 + 0.125 + 0.0625 = [29.6875]_{10}$

Hence, $[11101.1011]_2 = [29.6875]_{10}$

Table 1.1 lists the binary numbers from 0000 to 10000. Table 4.2 lists powers of 2 and their decimal equivalents and the number of K. The abbreviation K stands for $2^{10} = 1024$. Therefore, 1K = 1024, 2K = 2048, 3K = 3072, 4K = 4096, and so on. Many personal computers have 64K memory this means that computers can store up to 65,536 bytes in the memory section.

***Table 4.1*** *Binary Numbers*

| *Decimal* | *Binary* |
|---|---|
| 0 | 0 |
| 1 | 01 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |
| 11 | 1011 |
| 12 | 1100 |
| 13 | 1101 |
| 14 | 1110 |
| 15 | 1111 |
| 16 | 10000 |

***Table 4.2*** *Powers of 2*

| *Powers of 2* | *Equivalent* | *Abbreviation* |
|---|---|---|
| $2^0$ | 1 | |
| $2^1$ | 2 | |
| $2^2$ | 4 | |
| $2^3$ | 8 | |
| $2^4$ | 16 | |
| $2^5$ | 32 | |
| $2^6$ | 64 | |
| $2^7$ | 128 | |
| $2^8$ | 256 | |
| $2^9$ | 512 | |
| $2^{10}$ | 1024 | 1K |
| $2^{11}$ | 2048 | 2K |
| $2^{12}$ | 4096 | 4K |
| $2^{13}$ | 8192 | 8K |
| $2^{14}$ | 16384 | 16K |
| $2^{15}$ | 32768 | 32K |
| $2^{16}$ | 65536 | 64K |

### 4.5.4 Decimal to Binary Conversion

There are several methods for converting a decimal number into a binary number. The first method is to simply subtract values of powers of 2 from the decimal number until nothing remains. The value of the highest power of 2 is subtracted first, then the second highest and so on.

**Example 4.7:** Convert the decimal integer 29 to the binary number system.

**Solution:** First, the value of the highest power of 2 which can be subtracted from 29 is found. This is $2^4 = 16$.

Then, $29 - 16 = 13$.

If the value of the highest power of 2 which can be subtracted from 13 is $2^3$, then $13 - 2^3 = 13 - 8 = 5$. The value of the highest power of 2 which can be subtracted from 5 is $2^2$. Then $5 - 2^2 = 5 - 4 = 1$. The remainder after subtraction is 1 or $2^0$. Therefore, the binary representation for 29 is given by,

$$\begin{aligned} 29_{10} &= 2^4 + 2^3 + 2^2 + 2^0 = 16 + 8 + 4 + 0 \times 2 + 1 \\ &= 1 \quad 1 \quad 1 \quad 0 \; 1 \\ [29]_{10} &= [11101]_2 \end{aligned}$$

Similarly,
$$\begin{aligned} [25.375]_{10} &= 16 + 8 + 1 + 0.25 + 0.125 \\ &= 2^4 + 2^3 + 0 + 0 + 2^0 + 0 + 2^{-2} + 2^{-3} \\ [25.375]_{10} &= [11011.011]_2 \end{aligned}$$

This is a laborious method for converting numbers. It is convenient for small numbers and can be performed mentally, but is seldom used for larger numbers.

### 4.5.5 Double-Dabble Method

A popular method called **double-dabble method,** also known as divide-by-two method, is used to convert a large decimal number into its binary equivalent. In this method, the decimal number is repeatedly divided by 2 and the remainder after each division is used to indicate the coefficient of the binary number to be formed. Notice that the binary number derived is written from the bottom up.

**Example 4.8:** Convert $199_{10}$ into its binary equivalent.

**Solution:**
$$\begin{array}{rll} 199 \div 2 & = 99 + \text{remainder} & 1 \quad \text{(LSB)} \\ 99 \div 2 & = 49 + \text{remainder} & 1 \quad \uparrow \\ 49 \div 2 & = 24 + \text{remainder} & 1 \\ 24 \div 2 & = 12 + \text{remainder} & 0 \\ 12 \div 2 & = 6 + \text{remainder} & 0 \\ 6 \div 2 & = 3 + \text{remainder} & 0 \\ 3 \div 2 & = 1 + \text{remainder} & 1 \\ 1 \div 2 & = 0 + \text{remainder} & 1 \quad \text{(MSB)} \end{array}$$

The binary representation of 199 is, therefore, 11000111. Checking the result we have,

$$\begin{aligned} [11000111]_2 &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 128 + 64 + 0 + 0 + 0 + 4 + 2 + 1 \end{aligned}$$

$\therefore \quad [11000111]_2 = [199]_{10}$

Notice that the first remainder is the LSB and the last remainder is the MSB. This method will not work for mixed numbers.

NOTES

### 4.5.6 Decimal Fraction to Binary

The conversion of decimal fraction into binary fractions may be accomplished by using several techniques. Again, the most obvious method is to subtract the highest value of the negative power of 2 from the decimal fraction. Then, the next highest value of the negative power of 2 is subtracted from the remainder of the first subtraction, and this process is continued until there is no remainder or to the desired precision.

**Example 4.9:** Convert decimal 0.875 to a binary number.

**Solution:**

$$\begin{aligned} 0.875 - 1 \times 2^{-1} &= 0.875 - 0.5 = 0.375 \\ 0.375 - 1 \times 2^{-2} &= 0.375 - 0.25 = 0.125 \\ 0.125 - 1 \times 2^{-3} &= 0.125 - 0.125 = 0 \\ \therefore \quad [0.875]_{10} &= [0.111]_2 \end{aligned}$$

A much simpler method of converting longer decimal fractions to binary consists of repeatedly multiplying them by 2 and recording any carriers in the integer position.

**Example 4.10:** Convert $0.6940_{10}$ to a binary number.

**Solution:**

$$\begin{aligned} 0.6940 \times 2 &= 1.3880 = 0.3880 \text{ with a carry of } 1 \\ 0.3880 \times 2 &= 0.7760 = 0.7760 \text{ with a carry of } 0 \\ 0.7760 \times 2 &= 1.5520 = 0.5520 \text{ with a carry of } 1 \\ 0.5520 \times 2 &= 1.1040 = 0.1040 \text{ with a carry of } 1 \\ 0.1040 \times 2 &= 0.2080 = 0.2080 \text{ with a carry of } 0 \\ 0.2080 \times 2 &= 0.4160 = 0.4160 \text{ with a carry of } 0 \\ 0.4160 \times 2 &= 0.8320 = 0.8320 \text{ with a carry of } 0 \\ 0.8320 \times 2 &= 1.6640 = 0.6640 \text{ with a carry of } 1 \\ 0.6640 \times 2 &= 1.3280 = 0.3280 \text{ with a carry of } 1 \end{aligned}$$

We may stop here as the answer would be approximate.

$$\therefore \quad [0.6940]_{10} = [0.101100011]_2$$

If more accuracy is needed, continue multiplying by 2 until you have as many digits as necessary for your application.

**Example 4.11:** Convert $14.625_{10}$ to binary number.

**Solution:** First, the integer part 14 is converted into binary and then, the fractional part 0.625 is converted into binary as follows:

| *Integer part* | *Fractional part* |
|---|---|
| $14 \div 2 = 7 + 0$ | $0.625 \times 2 = 1.250$ with a carry of 1 |
| $7 \div 2 = 3 + 1$ | $0.250 \times 2 = 0.500$ with a carry of 0 |
| $3 \div 2 = 1 + 1$ | $0.500 \times 2 = 1.000$ with a carry of 1 |
| $1 \div 2 = 0 + 1$ | |

$\therefore$ The binary equivalent is $[1110.101]_2$

**CHECK YOUR PROGRESS**

7. What is a base two system?
8. Why is a binary number system used in digital computers?
9. State any one method used for the conversion of a decimal number into a binary number.

## 4.6 RECURSION AND RECURRENCE RELATIONS

The numbers in the sequence 0, 1, 2, 3, 5, 8, 13, 21,..... in which each new term is the sum of the previous two terms are called the Fibonacci numbers. If we denote the $(n + 1)$th Fibonacci number by $f_n$, we have,

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

and $f_0 = 0$ and $f_1 = 1$. This is called a recursive definition in which each element of the sequence is defined in terms of the previous numbers in the sequence.

We can define a function with the set of non-negative intergers and its domain by,

1. Specifying the value of the function at zero
2. giving a rule for finding its value at an integer from its values at smaller integers

Such a function is called a recursively defined function or such a definition is called recursive definition.

For example, consider the sequence of powers of 3 given by,

$$a_n = 3^n \text{ for } n = 0, 1, 2, .....$$

This sequence can also be defined by giving the first term of the sequence, namely $a_0 = 1$ and a rule for finding a term of the sequence from the previous one, namely,

$$a_{n+1} = 3a_n \text{ for } n = 0, 1, 2, ....$$

**Example 4.12:** Find a recursive definition of binomial coefficients.

**Solution:** Denote the binomial coefficient $n_k$ by $C(n, k)$. Then the recursive definition for $C(n, k)$ where $n \geq 0$, $k \geq 0$ and $n \geq k$ is given by,

$$C(n, 0) \quad = \quad 1;$$

$$C(n, n) = 1; \text{ and}$$
$$C(n, k) = C(n-1, k) + C(n-1, k-1) \text{ for } n > k > 0.$$

**Example 4.13:** Suppose that $f$ is recursively defined as follows,

$$f(0) = 2$$
$$f(n+1) = 3f(n)+2$$

Find $f(1), f(2), f(3)$ and $f(4)$.

**Solution:** From the recursive definition, it follows that:

$$f(1) = 3f(0) + 2 = 3.2 + 2 = 8$$
$$f(2) = 3f(1) + 2 = 3.8 + 2 = 26$$
$$f(3) = 3f(2) + 2 = 3.26 + 2 = 80$$
$$f(4) = 3f(3) + 2 = 3.80 + 2 = 242$$

**Example 4.14:** Give a recursive definition of $f(n) = n!$

**Solution:** Since $0! = 1$ and $(n+1)! = (n+1)n!$, the desired rule is,

$$f(0) = 1$$
$$f(n+1) = (n+1) f(n)$$

**Example 4.15:** Give a recursive definition of $a^n$ where $a$ is a non-zero number and $n$ is a non-negative integer.

**Solution:** It can be stated as,

$$a^0 = 1; \text{ and } a^{n+1} = a.a^n \text{ for } n \geq 0.$$

Where $a^n$ uniquely defined is for all non-negative integers $n$.

**Example 4.16:** Give a recursive definition for polynomial expression.

**Solution:** An expression of the form,

$$f(x) = a_0x^n + a_1x^{n-1} + .... + a_{n-1}x + a_n$$

Where $a$'s are constants ($a_0 \neq 0$) and $n \geq 0$, is called polynomial in $x$ of degree $n$.

Let $S$ be a set of coefficients. Then, recursive definition for polynomial is,

1. A zeroth degree (constant) polynomial is an element of $S$.
2. For $n \geq 1$, an $n$th degree polynomial expression is an expression of the form $q(x)x+a$, where $q(x)$ is an $(n-1)$th degree polynomial and $a \in S$.

**Example 4.17:** Using recursive definition for polynomial expression prove that

$f(n) = 3n^3 - 8n^2 + 2n + 4$ is a third-degree polynomial expression

**Solution:** Given
$$f(n) = 3n^3 - 8n^2 + 2n + 4$$
$$= (3n^2 - 8n + 2)\, n + 4 = ((3n - 8)\, n + 2)\, n + 4$$
$$= (((3)\, n - 8)\, n + 2)\, n + 4$$

Now, 3 is a zeroth degree polynomial.

$\Rightarrow$ (3) $n - 8$ is a first degree polynomial.

$\Rightarrow$ ((3) $n - 8$) $n + 2$ is a second degree polynomial

$\Rightarrow f(n) = (((3)\, n - 8)\, n + 2)\, n + 4$ is a third degree polynomial.

***Note:*** The final expression is,

$$f(n) = (((3)\, n - 8)\, n + 2)\, n + 4$$

This is called its telescoping form. If you use it to calculate $f(6)$, you need only three multiplications and three additions or substractions. This is called Horner's method for evaluating a polynomial expression.

**Example 4.18:** Write $p(x) = x^3 - 6x^2 + 11x - 6$ in telescoping form.

**Solution:** The telescoping form is as follows:

$$\begin{aligned} p(x) &= x^3 - 6x^2 + 11x - 6 \\ &= (((1)\, x - 6)\, x + 11)\, x - 6 \end{aligned}$$

**Example 4.19:** Write $g(x) = -x^5 + 3x^4 + 3x^3 + 2x^2 + x$ in telescoping form.

**Solution:** The telescoping form is as follows:

$$\begin{aligned} g(x) &= x^5 + 3x^4 + 3x^3 + 2x^2 + x \\ &= (((((-1)\, x + 3)\, x + 3)\, x + 2)\, x + 1)\, x + 0 \end{aligned}$$

### 4.6.1 Recursion and Iteration

There is another way to evaluate a function from its recursive definition. Instead of sucessively reducing the computation to evaluate the function at small integers, we can start with the basis and successively apply the recursive definition to find the values of the function at successive larger intergers. Such a procedure is called an iterative procedure.

For Example, to find $n!$ using an iterative procedure, we start with 1, the value of the factorial function at 0, and multiply it successively by each positive integer less than or equal to $n$.

***Note:*** Often an iterative approach for the evaluation of a recursively defined sequence requires much less computation than a procedure using recursion. However, is sometimes preferable to use a recursive procedure even if it is less efficient than the iterative procedure.

### 4.6.2 Closed Form Expression

Let $f(x_1, x_2, .... x_n)$ be an algebraic expression involving variables, $x_1, x_2, .... x_n$ which are allowed to take values from some predetermined set. $f$ is a closed form expression if there exists a number $F$ such that the evaluation of $f$ with any allowed values $x_1, x_2, ....x_n$ will take no more than $F$ operations.

For example, (*i*) A closed form expression for $1 + 2 + ... + n$ is,

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

(*ii*) A closed form expression for $1^2 + 2^2 + ... + n^2$ is,

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

### 4.6.3 Sequence of Integers

A sequence of integers is a function from the natural numbers into integers. That is, if $f$ is a sequence of integers, then $f: N \rightarrow Z$ is a function. We use the notation $f_n$ or $f(n)$ to denote the image of any natural number $n$. We call $f_n$ and $n$th term of the sequence.

***Note:*** A sequence is often called a discrete function.

**Example 4.20:** Prove by induction that $f(k) = 3k + 2$, $k \geq 0$ is a closed form expression of the sequence $f$ defined recursively by $f(0) = 2$ and $f(k) = f(k-1) + 3$ for $k \geq 1$.

**Solution:**

1. Basis of induction: If $k = 0$, then $f(0)$. But $f(0) = 3(0) + 2 = 2$ as defined.
2. Induction step: Now assume that for some $k > 0$,

$$f(k) = 3k + 2 \qquad ...(1)$$

We have,

$$f(k+1) = f(k) + 3 \text{ by the recursive definition}$$
$$= 3k + 2 + 3 = 3k + 3 + 2$$
$$= 3(k+1) + 2$$

Therefore, by the principle of mathematical induction the result follows. Hence, $f(k) = 3k + 2$, $k \geq 0$ is a closed form expression for the sequence $f$.

**Example 4.21:** Define the sequence of numbers $f$ by $f_0 = 100$ and $f_n = 1.08\, f_{n-1}$ for $n \geq 1$. Prove by induction that $f_n = 100\,(1.08)^n$, $n \geq 0$ is a closed form of expression for the sequence $f$.

**Solution:**

1. Basis of induction: If $k = 0$, then $f_0 = 100\,(1.08)^0 = 100$ as defined.
2. Induction step: Now assume that for some $k > 0$

$$f_k = 100\,(1.08)^k$$

Then,

$$f_{(k+1)} = (1.08) f_k \text{ by the recursive definition}$$
$$= (1.08)100^k$$
$$= 100\,(1.08)^{k+1}$$

Therefore, by the principle of mathematical induction the result follows. Hence, $f_n = 100\,(1.08)^n$, $n \geq 0$ is closed form expression for the sequence $f$.

***Notes:***

1. The sequence $f:N \rightarrow Z$, defined by $f(n) = n^3 - n$, is a sequence of integers.
2. The sequence $f:N \rightarrow Z$, defined by $f(0) = 1$ and $f(n) = f(n-1) + 2$ for $n \geq 1$ is a sequence of integer.
3. The codomain of a sequence can be any set. We use the notation $\{f_n\}$ to describe the sequence. We describe sequences by listing the terms of the sequence in the order of increasing subscripts.

For example,

(*i*) Consider the sequence $f:N \rightarrow S$ defined by $f(n) = f_n = \dfrac{1}{n+1}$. The list of the terms of this sequence $f_0, f_1, f_2, \ldots$ begins with:

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots.$$

(*ii*) Consider the sequence $f:N \rightarrow Z$ defined by $f_n = 4^n$. The list of the terms of the sequence $f_0, f_1, f_2, \ldots$ begins with:

$$1, 4, 16, 64, \ldots.$$

## 4.6.4 Recurrence Relations

The expressions for permutations and combinations are one of the most fundamental tools for counting the elements of finite sets. They often prove to be inadequate and many problems of computer sciences require a different approach. An important alternative approach uses recurrence relations (often called recurrence equations or difference equation) to define the terms of a sequence. A formal definition of recurrence relations is difficult because of the wide variety of forms in which such relations can be written, but the concept is straightforward. You have already seen an example of a recurrence relation in the definition of the Fibonacci sequence, where for $n \geq 2$, the term $f_n$ is defined by the recurrence relation,

$$f_n = f_{n-1} + f_{n-2}$$

The salient characteristic of a recurrence relation is the specification of the term $f_n$ as a function of the terms $f_0, f_1, \ldots, f_{n-1}$. By itself, however, a recurrence relation is not sufficient to define the terms of a sequence. You must also specify the values of some initial terms of the sequence. Thus, in our definition of the Fibonacci sequence, we set $f_0 = 0$ and $f_1 = 1$. These are called the boundary conditions or initial conditions of the sequence.

Recall that a recursive definition of a sequence specifies one or more initial terms and a rule for determining subsequent terms from those that precede them. Recursive definitions can be used to solve counting problems. When they are, the rule for finding terms from those that precede them is called a recurrence relation.

***Recurrence relation:*** A recurrence relation for the sequence $\{f_n\}$ is a formula that expresses $f_n$ in terms of one or more of the previous terms of the sequence, namely, $f_0, f_1, \ldots f_{n-1}$, for all integers $n$ with $n \geq n_0$, when $n_0$ is a non-negative integer. A sequence is called a solution of a recurrence relation if its terms satisfy the recurrence relation.

**Example 4.22:** Determine whether the sequence $\{f_n\}$ is a solution of the recurrence relation,

$$f_n = 2f_{n-1} - f_{n-2}$$

for $n = 2, 3, 4, \ldots$ Where $f_n = 3n$ for every non-negative integer $n$.

**Solution:** Suppose that, $f_n = 3n$ for every non-negative integer $n$. Then for $n \geq 2$,

$$\begin{aligned} f_n &= 2f_{n-1} - f_{n-2} \\ &= 2[3(n-1)] - 3(n-2) \text{ since } f_n = 3n \\ &= 6n - 6 - 3n + 6 = 3n \end{aligned}$$

Therefore, $\{f_n\}$, where $f_n = 3n$ is a solution of the recurrence relation.

**Example 4.23:** Show that the sequence $\{f_n\}$ is a solution of the recurrence relation $f_n = -3f_{n-1} + 4f_{n-2}$ if $f_n = 2(-4)^n + 3$.

**Solution:** Suppose that, $f_n = 2(-4)^n + 3$.

Then
$$\begin{aligned} f_n &= -3f_{n-1} + 4f_{n-2} \\ &= -3[2(-4)^{n-1} + 3] + 4[2(-4)^{n-2} + 3] \\ &= -6(-4)^{n-1} - 9 + 8(-4)^{n-2} + 12 \\ &= -6(-4)^{n-1} - 2(-4)^{n-1} + 3 \\ &= -8(-4)^{n-1} + 3 \\ &= 2(-4)^n + 3 \end{aligned}$$

Therefore $\{f_n\}$ where $f_n = 2(-4)^n + 3$ is a solution of the recurrence relation.

Now you will study about a class of recurrence relations known as linear recurrence relations with constant coefficients.

***Linear recurrence relation:*** A recurrence relation of the form

$$a_0 f_n + a_1 f_{n-1} + a_2 f_{n-2} + \ldots + a_k f_{n-k} = f(n) \qquad \ldots(4.1)$$

Where $a_1, a_2$ and so on are constants, is called a linear recurrence relation with constant coefficients. The recurrence relation as shown in Equation 4.1 is known as a $k$th-order recurrence relation, provided that both $a_0$ and $a_k$ are non zero.

***Note:*** The phrase '$k$th-order' means that each term in the sequence depends only on the previous $k$ terms.

For example, consider the Fibonacci sequence defined by the recurrence relation $f_n = f_{n-1} + f_{n-2}$, $n \geq 2$ and the initial condition $f_0 = 0$ and $f_1 = 1$. The recurrence relation is called a second-order relation because $f_n$ depends on the two previous terms of $f$.

For example, consider the recurrence relation $f(k) - 5f(k-1) + 6f(k-2) = 4k + 10$ defined for $k \geq 2$, together with the initial condition $f(0) = \frac{7}{3}$ and $f(1) = 5$. Clearly, it is a second-order linear recurrence relation.

***Homogeneous recurrence:*** A $k$th-order linear relation is a homogeneous recurrence relation if $f(n) = 0$ for all $n$. Otherwise, it is called non-homogeneous.

For example, consider the recurrence relation $c(k) - 5c(k-1) + 8c(k-2) = 0$ together with the initial condition $c(0) = 5$ and $c(1) = 2$. It is a second-order homogeneous recurrence relation.

**Example 4.24:** Which of the recurrence relations of the following are homogeneous and which are non-homogeneous?

(*i*) $f_n = nf_{n-2}$ (*ii*) $a_n = a_{n-1} + a_{n-3}$

(*iii*) $b_n = b_{n-1} + 2$ (*iv*) $s(n-2) + s(n-4)$

**Solution:** The relation $f_n = nf_{n-2}$, $a_n = a_{n-1} + a_{n-3}$, $s(n) = s(n-2) + s(n-4)$ are all homogeneous and the relation $b_n = b_{n-1} + 2$ is non-homogeneous.

### 4.6.5 Linear Homogenous Recurrence Relations (LHRR)

Before writing an algorithm for solving a recurrence relations, let us examine a few recurrence relations that arise from certain closed form expressions. The procedure is illustrated by the following examples.

**Example 4.25:** Form the recurrence relation given $f_n = 3.5^n$, $n \geq 0$.

**Solution:** If, $n \geq 1$,

Then $f_n = 3.5^n = 3.5.5^{n-1}$

$= 5.3.5^{n-1} = 5f_{n-1}$

So, the recurrence relation is $f_n - 5f_{n-1} = 0$ with $f_0 = 3$.

**Example 4.26:** Find the recurrence relation which satisfies that

$y_n = A(3)^n + B(-2)^n$.

**Solution:** Given, $y_n = A(3)^n + B(-2)^n$

Therefore, $y_{n+1} = A(3)^{n+1} + B(-2)^{n+1} = 3A(3)^n - 2B(-2)^n$

and $y_{n+2} = A(3)^{n+2} + B(-2)^{n+2} = 9A(3)^n + 4B(-2)^n$

Eliminating $A$ and $B$ from these equations, we get

$$\begin{vmatrix} y_n & 1 & 1 \\ y_{n+1} & 3 & -2 \\ y_{n+2} & 9 & 4 \end{vmatrix} = 0$$

Or $y_{n+2} - y_{n+1} - 6y_n = 0$, which is the required recurrence relation.

**Example 4.27:** Find the recurrence relation which satisfies that $y_n = A(3)^n + B(-4)^n$

**Solution:** Given, $y_{n+1} = A(3)^n + B(-4)^n$

Therefore, $y_{n+1} = 3A\,(3)^n - 4B(-4)^n$

and $y_{n+2} = 9A(3)^n + 16B\,(-4)^n$

Eliminating $A$ and $B$ from these equation, we get

$$\begin{vmatrix} y_n & 1 & 1 \\ y_{n+1} & 3 & -4 \\ y_{n+2} & 9 & 16 \end{vmatrix} = 0$$

Or $y_{n+2} + y_{n+1} - 12y_n = 0$, which is the required recurrence relation.

**Example 4.28:** Find the recurrence relation which satisfies that $y_n = (A + Bn)4^n$

**Solution:** Given, $y_n = (A + Bn)\,4^n$

$= A4^n + nB4^n$

Therefore, $y_{n+1} = 4A4^n + 4\,(n+1)\,B4^n$

and $y_{n+2} = 16A4^n + 16\,(n+2)\,B4^n$

Eliminating $A$ and $B$ from these equation, we get

$$\begin{vmatrix} y_n & 1 & n \\ y_{n+1} & 4 & 4\,(n+1) \\ y_{n+2} & 16 & 16\,(n+2) \end{vmatrix} = 0$$

Or $y_{n+2} - 8y_{n+1} = 0$, which is the required recurrence relation.

### 4.6.6 Solving Linear Homogeneous Recurrence Relations

Consider a linear homogeneous recurrence relations of degree $k$ with constant coefficients.

$$f_n = a_1 f_{n-1} + a_2 f_{n-2} + \ldots + a_k f_{n-k}$$

Where $a_1, a_2, \ldots., a_k$ are real numbers and $a_k \neq 0$. The basic approach for solving linear homogeneous recurrence relations is to look for solutions of the form $f_n = r^n$, where $r$ is a constant. Note that $f_n = r^n$ is the solution of the recurrence relation $f_n = a_1 f_{n-1} + a_2 f_{n-2} + \ldots. + a_k f_{n-k}$ if and only if,

$$r_n = C_1\, r^{n-1} + C_2\, r^{n-2} + \ldots + C_k\, r^{n-k}$$

When both sides of this equation are divided by $r^{n-k}$ and the right-hand side is substracted from the left, we obtain:

$$r^k - C_1\, r^{k-1} - C_2\, r^{k-2} - \ldots. - C_{k-1}\, T - C_k = 0$$

Consequently, the sequence $\{f_n\}$ with $f_n = r^n$ is the solution if and only if $r$ is a solution of this last equation.

***Characteristic equation:*** The characteristic equation of the homogeneous *k*th order linear relation,

$$f_n + a_1 f_{n-1} + a_2 f_{n-2} + ... + a_k f_{n-k} = 0$$

This is the *k*th degree polynomial equation

$$r^k + C_1 r^{k-1} + C_2 r^{k-2} + ... + C_{k-1} r + C_k = 0$$

The solutions of this equation are called the characteristic roots of the recurrence relation.

**Example 4.29:** What is the characteristic equation of,

$$Q(k) + 2Q(k-1) - 3Q(k-2) - 6Q(k-4) = 0$$

**Solution:** The characteristic equation of the given equation

$Q(k) + 2Q(k-1) - 3Q(k-2) - 6Q(k-4) = 0$

is $r^4 + 2r^3 - 3r^2 - 6 = 0$.

Note that the absence of a $Q(k-3)$ term means that there is no $r^{4-3} = r$ term in the characteristic equation.

**Example 4.30:** What is the characteristic equation of $T(k) - 7T(k-2) + 6T(k-3) = 0$?

**Solution:** The characteristic equation is $r^3 - 7r + 6 = 0$ and 1, 2 and $-3$ are the characteristic roots.

**Algorithm for solving *k*th-order homogeneous linear recurrence relation**

**Step 1:** If $f_n + a_1 f_{n-1} + a_2 f_{n-2} + ... + a_k f_{n-k} = 0$ is a given recurrence relation, then write its characteristic equation.

It is, $r^k + C_1 r^{k-1} + C_2 r^{k-2} + ... + C_{k-1} r + C_k = 0$.

**Step 2:** Find all the characteristic roots of this equation.

**Step 3:** Case (*i*) If there are *k* distinct roots, say $c_1, c_2, ... c_k$, then the general solution of the recurrence relation is,

$$f_n = A_1 c_1^k + A_2 c_2^k + ... + A_k c_k^k$$

Case (*ii*) Suppose that $c_1$ is a root of multiplicity *m*. Then the corresponding solution is,

$$f_n = (A_1 r^{m-1} + A_2 r^{m-2} + ... + A_m - 2^{r2} + A_{m-1} T + A_m) c_1^T + ...$$

**Step 4:** Use the boundary conditions to determine the constants $A_1, A_2, .... A_k$.

**Example 4.31:** Solve the Fibonacci sequence $\{f_n\}$ defined by,

$f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$ with the initial conditions $f_0 = 1$ and $f_1 = 1$.

**Solution:** The first step is to form the characteristic equation corresponding to the given difference equation. In this case it is,

$$r^2 - r - 1 = 0$$

Solving, we get $c_1 = \frac{1+\sqrt{5}}{2}$ and $c_2 = \frac{1-\sqrt{5}}{2}$ as the characteristic roots. It follows that the general solution is,

$$f_n = A_1 c_1^n + A_2 c_2^n$$

Where $A_1$ and $A_2$ are constants. Since $f_0 = 1$ and $f_1 = 1$, we get

$$0 = A_1 + A_2$$

and

$$1 = A_1 c_1 + A_2 c_2$$

$$1 = A_1\left(\frac{1+\sqrt{5}}{2}\right) + A_2\left(\frac{1-\sqrt{5}}{2}\right)$$

On solving, we get

$$A_1 = \frac{1}{\sqrt{5}} \text{ and } A_2 = \frac{-1}{\sqrt{5}}$$

Hence, the solution is,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

**Example 4.32:** If the recurrence relation is $u_{n+1} - 2u_n = 0$, then find the closed form expression (solution) for $u_n$.

**Solution:** The characteristic equation of the given recurrence relation is $r - 2 = 0$, i.e., $r = 2$. Therefore, the general solution is $u_n = A2^n$. Hence, $u_n = u_0 \cdot 2^n$ is the closed form expression, where the value of $u_0$ is the initial condition.

**Example 4.33:** Find $f(n)$ if $f(n) = 7f(n-1) - 10f(n-2)$, given that $f(0) = 4$ and $f(1) = 17$.

**Solution:** The characteristic equation of the given recurrence relation is $r^2 - 7r + 10 = 0$.

Its characteristic roots are $r = 2, 5$. So, the general solution of the recurrence relation is,

$$f(n) = A_1 (2)^n + A_2 (5)^n$$

Since, $f(0) = 4,\ 4 = A_1 + A_2$

Again, $f(1) = 17$ implies $17 = 2A_1 + 5A_2$

Solving, we get $A_1 = 1$ and $A_2 = 3$.

Therefore, $f(n) = (2)^n + 3(5)^n$

**Example 4.34:** Find $T(k)$ if $T(k) - 7T(k-2) + 6T(k-3) = 0$, where $T(0) = 8$, $T(1) = 6$ and $T(2) = 22$.

**Solution:** The characteristic equation is $r^2 - 7r + 6 = 0$

Its roots are 1, 2 and –3.

Therefore, the general solution is $T(k) = A_1 (1)^k + A_2 (2)^k + A_3 (-3)^k$.

Now,

$$T(0) = 8 \Rightarrow A_1 + A_2 + A_3 = 8$$

$$T(1) = 6 \Rightarrow A_1 + 2A_2 - 3A_3 = 6$$

$$T(2) = 22 \Rightarrow A_1 + 4A_2 + 9A_3 = 22$$

Solving, we get $A_1 = 5, A_2 = 2$ and $A_3 = 1$.

Hence,

$$T(k) = 5 + 2(2)^k + 1(-3)^k$$

$$= 5 + 2^{k+1} + (-3)^k$$

**Example 4.35:** Solve $f_k - 8f_{k-1} + 16f_{k-2} = 0$, where $f_2 = 16$ and $f_3 = 80$.

**Solution:** The characteristic equation is $r^2 - 8r + 16 = 0$ (or) $(r-4)^2 = 0$

So, $r = 4$ is a double characteristic root.

Therefore, the general solution is $f_k = (A_1 + A_2 k)\, 4^k$

Now, $f_2 = 16 \Rightarrow (A_1 + 2A_2)\, 16 = 16$

And, $f_3 = 80 \Rightarrow (A_1 + 3A_2)\, 64 = 80$

Solving, we get $A_1 = \frac{1}{2}$ and $A_2 = \frac{1}{4}$

Hence, the solution is,

$$f_k = \left(\frac{1}{2} + \frac{1}{4}k\right)4k$$

$$= (2 + k)4^{k-1}$$

**Example 4.36:** Find a solution to the recurrence relation $C_n = -3C_{n-1} - 3C_{n-2} - C_{n-3}$ for $n \geq 3$ with initial conditions $C_0 = 1$, $C_1 = -2$ and $C_2 = 1$.

**Solution:** The characteristic equation is $r^3 + 3r^2 + 3r + 1 = 0$ (or) $(r+1)^3 = 0$.

So $r = -1$ is a characteristic root of multiplicity 3. Therefore, the general solution is,

$$C_n = (A_1 + A_2 n + A_3 n^2)(-1)^n$$

Now, $C_0 = 1 \Rightarrow A_1 = 1$

$$C_1 = 2 \Rightarrow -(A_1 + A_2 + A_3) = -2$$
$$C_2 = 1 \Rightarrow A_1 + 2A_2 + 4A_3 = 1$$

Solving, we get $A_1 = 1, A_2 = 2, A_3 = 1$.

Hence, the solution is $C_n = (1 + 2n - n^2)(-1)^n$.

### 4.6.7 Solving Linear Non-Homogeneous Recurrence Relations

The solution of a linear non-homogeneous recurrence relation with constant coefficient is the sum of the two parts: the homogeneous solution, which satisfies the recurrence relation when the right-hand side of the equation is set 1 to 0, and the particular solution, which satisfies the difference equation with $f(n)$ on the right-hand side.

There is no general procedure for determining the particular solution of a difference equation. However, in simple cases, this solution can be obtained by the method of inspection. To determine the particular solution, the following rules are used:

***Rule* 1:** When $f(n)$ is of the form of a polynomial of degree $m$ in $n$,

$$k_0 + k_1 n + k_2 n^2 + ... + k_{m-1} n^{m-1} + k_m n^m$$

Then the corresponding particular solution will be of the form,

$$Q_0 + Q_1 n + Q_2 n^2 + ... + Q_{m-1} n^{m-1} + Q_m n^m$$

***Rule* 2:** When $f(n)$ is of the form,

$$(k_0 + k_1 n + k_2 n^2 + ... + k_{m+1} n^{m-1} + k_m n^m)a^n$$

Then the corresponding particular solution is of the form,

$$(Q_0 + Q_1 n + Q_2 n^2 + .... + Q_{m-1} n^{m-1} + Q_m n^m)a^n$$

If $a$ is not a characteristic root of the recurrence relation.

**Example 4.37:** Solve $S(k) - S(k-1) - 6S(k-2) = -30$, where $S(0) = 20$, $S(1) = -5$.

**Solution:** The associated homogeneous relation is $S(k) - S(k-1) - 6S(k-2) = 0$.

The characteristic equation is $r^2 - r - 6 = 0$.

Its characteristic roots are $r = -2, 3$.

So, the homogeneous solution is $A_1(-2)^k + A_2(3)^k$.

Since the right-hand side of $S(k) - S(k-1) - 6S(k-2) = -30$ is a constant, by rule 1, the particular solution will also be a constant, say $Q$ into (1).

Thus,

$$Q - Q - 6Q = -30 \Rightarrow -6Q = -30$$
$$\Rightarrow Q = 5$$

Therefore, the general solution is

$$S(k) = A_1(-2)^k + A_2(3)^k + 5$$

Using the initial conditions, $S(0) = 20, S(1) = -5$

$$20 = A_1 + A_2 + 5$$
$$-5 = -2A_1 + 3A_2 + 5$$

This will yield $A_1 = 11$ and $A_2 = 4$.

Hence, the complete solution is $S(k) = 11(-2)^k + 4(3)^k + 5$.

***Rule* 3:** If $a$ is a characteristic root of multiplicity $r - 1$, when $f(n)$ is of the form,

$$(k_0 + k_1 n + k_2 n^2 + ... + k_{m-1} n^{m-1} + k_m n^m)a^n$$

Then the corresponding particular solution is of the form,

$$n^{r-1}(Q_0 + Q_1 n + Q_2 n^2 + ... + Q_{m-1} n^{m-1} + Q_m n^m)a^n.$$

***Note:*** The general solution of the recurrence relation is the sum of the homogeneous solution and particular solution. If no initial conditions are given, then you have obtained the solution. If $m$ initial conditions are given, obtain $m$ linear equations in $m$ unknowns and solve the system, if possible, to get a complete solution.

**Example 4.38:** Solve the recurrence relation $f_n - 5f_{n-1} + 6f_{n-2} = 1$. ...(1)

**Solution:** The associated homogeneous relation is $f_n - 5f_{n-1} + 6f_{n-2} = 0$.

The characteristic equation is $r^2 - 5r + 6 = 0$.

Its characteristic roots are $r = 2, 3$ so the homogeneous solution is,

$$A_1(2)^n + A_2(3)^n$$

Since the right-hand side $f(n) = 1$ is a constant, by rule 1, the particular solution will also be a constant, say $Q$. Substituting $Q$ into Equation (1), we obtain:

$$Q - 5Q + 6Q = 1$$

This implies that $Q = \frac{1}{2}$. Therefore, the complete solution is,

$$f_n = A_1(2)^n + A_2(3)^n + \frac{1}{2}.$$

**Example 4.39:** Find the particular solution of the recurrence relation,

$$f(n) + 5f(n-1) + 6f(n-2) = 3n^2 - 2n + 1 \qquad ...(1)$$

**Solution:** By rule 1, the particular solution is of the form,

$$Q_0 + Q_1 n + Q_2 n^2 \qquad ...(2)$$

Substituting Equation (2) into Equation (1), you obtain:

$$(Q_0 + Q_1 n + Q_2 n^2) + 5(Q_0 + Q_1(n-1) + Q_2(n-1)^2)$$
$$+ 6(Q_0 + Q_1(n-2) + Q_2(n-2)^2) = 3n^2 - 2n + 1$$

**NOTES**

Which simplifies to,

$$(12Q_0 - 17Q_1 + 29Q_2) + (12Q_1 - 34Q_2)n + 12Q_2n^2 = 3n^2 - 2n + 1 \quad (3)$$

Comparing the two sides of Equation (3), you obtain,

$$12Q_2 = 3;\ 12Q_1 - 34Q_2 = -2;\ 12Q_0 - 17Q_1 + 29Q_2 = 1$$

Which yields $Q_2 = \frac{1}{4}, Q_1 = \frac{13}{24}, Q_0 = \frac{71}{288}$.

Therefore, the particular solution is $\frac{71}{288} + \frac{13}{24}n + \frac{1}{4}n^2$.

**Example 4.40:** Solve $a_r + 5a_{r-1} = 9$ with initial condition $a_0 = 6$.

**Solution:** The associated homogeneous recurrence relation is $a_r + 5a_{r-1} = 0$.

The characteristic equation is $r + 5 = 0$.

Therefore, $r = -5$. So, the homogeneous solution is $A_1(-5)^r$. Since the right-hand side of the given relation is a constant, the particular solution will also be a constant $Q$. Substituting in the recurrence relation, we get:

$$Q + 5Q = 9$$

So, $Q = \frac{3}{2}$. Therefore, the general solution is $a_r = A_1(-5)^r + \frac{3}{2}$.

Using the initial condition $a_0 = 6$, we get

$$6 = A_1 + \frac{3}{2} \Rightarrow A_1 = \frac{9}{2}$$

Hence, the complete solution is,

$$a_r = \frac{9}{2}(-5)^r + \frac{3}{2}$$

**Example 4.41:** Solve the recurrence relation $f(n) - 7f(n-1) + 10f(n-2) = 6 + 8n$ with $f(0) = 1$ and $f(1) = 2$. ...(1)

**Solution:** The homogeneous solution is $A_1(2)^n + A(2)^n + A_2(5)^n$. By rule 1, the particular solution is of the form $Q_0 + Q_1{}^n$.

Substituting in Equation (1), we get :

$$(Q_0 + Q_1{}^n) - 7(Q_0 + Q_1(n-1)) + 10(Q_0 + Q_1(n-2)) = 6 + 8n$$

Comparing the two sides, you obtain

$$4Q_0 - 13Q_1 = 6 \text{ and } 4Q_1 = 8$$

Which yields, $Q_0 = 8$, $Q_1 = 2$. Therefore, the particular solution is $8 + 2n$ and the general solution is,

$$f(n) = A_1(2)^n + A_2(5)^n + 8 + 2n$$

Now the initial conditions,

$$f(0) = 1 \Rightarrow A_1 + A_2 + 8 = 1$$

and

$$f(1) = 2 \Rightarrow 2A_1 + 5A_2 + 10 = 2$$

Solving, we get $A_1 = -9$ and $A_2 = 2$. Hence, the complete solution is,

$$f(n) = -9(2)^n + 2(5)^n + 8 + 2n$$

**Example 4.42:** Find the particular solution of the recurrence relation $a_n + 5a_{n-1} + 6a_{n-2} = 42(4)^n$. ...(1)

**Solution:** Now $r^2 + r + 6 = 0$, $r = -2, -3$ are characteristic roots. Since 4 is not a characteristic root, by rule 2, you can assume that the general form of the particular solution is $Q(4)^n$. Substituting in Equation (1), you will obtain

$$Q.(4)^n + 5Q.(4)^{n-1} + 6Q.(4)^{n-2} = 42\,(4)^n$$

$$\Rightarrow Q.4^{n-2}\,[16 + 20 + 6] = 42\,(4)^n \Rightarrow Q.4^{n-2}\,(42) = 42\,(4)^n$$

$$\Rightarrow Q = 4^2 = 16$$

Therefore, the particular solution is $16\,(4)^n = 4^{n+2}$

**Example 4.43:** Find the particular solution of the recurrence relation,

$f_n + f_{n-1} = 3n2^n$.

**Solution:** The characteristic equation is $r + 1 = 0$. Therefore, $r = -1$ is a characteristic root. Since 2 is not a characteristic root, by rule 2, the general form of the particular solution is $(Q_0 + Q_1 n)2^n$. Substituting in Equation (1), we obtain:

$$(Q_0 + Q_1 n)2^n + (Q_0 + Q_1\,(n-1))2^{n-1} = 3n2^n$$

Which simplifies to,

$$Q_0 2^n + Q_1 n2^n + \frac{1}{2}\,Q_0 2^n + \frac{1}{2}Q_1 n2^n - \frac{1}{2}\,Q_1 n2^n = 3\text{n}2^{\text{n}}$$

$$\Rightarrow (\frac{3}{2}\,Q_0 - \frac{1}{2}\,Q_1)2^n + \frac{3}{2}\,Q_1 n2^n = 3n2^n$$

Comparing the two sides, we obtain:

$$\frac{3}{2}\,Q_0 - \frac{1}{2}\,Q_1 = 0$$

$$\frac{3}{2}\,Q_1 = 3$$

Thus, $Q_0 = \frac{3}{2}$ and $Q_1 = 2$ and the particular solution is $(\frac{3}{2} + 2n)2^n$.

**Example 4.44:** Find the particular solution of the recurrence relation $f(n) - 2f(n-1) = 3.2^n$

**Solution:** The characteristic equation is $r - 2 = 0$. Since, $r = 2$ is the charactertistic root of multiplicity 1, by rule 3, the general form of the particular solution is $Qn2^n$. Substituting in Equation (1), you will obtain:

$$Qn2^n - 2Q.(n-1)2^{n-1} = 3.2^n$$
$$\Rightarrow Q.2^n = 3.2^n$$
$$\Rightarrow Q = 3$$

Thus, the particular solution is $3n2^n$.

**Example 4.45:** Find the general solution of $f(n) - 3f(n-1) - 4f(n-2) = 4^n$ ...(1)

**Solution:** The associated homogeneous relation is,

$$f(n) - 3f(n-1) - 4f(n-2) = 0$$

Its characteristic equation is $r^2 - 3r - 4 = 0$

Solving, we get $r = -1, 4$ as characteristic roots. Therefore, the homogeneous solution is $A_1(-1)^n + A_2(4)^n$. Since, 4 is a charactetistic root, by rule 3, we assume that the general form of the particular solution is $Qn4^n$.

Substituting in Equation (1), we get:

$$Qn4^n - 3Q(n-1)4^{n-1} - 4Q(n-2)4^{n-2} = 4^n$$
$$\Rightarrow Qn4^n - 3Qn4^{n-1} + 3Q4^{n-1} - 4Qn4^{n-2} + 8Q4^{n-2} = 4^n$$
$$\Rightarrow (16Qn - 12\,Qn + 12Q - 4Qn + 8Q)4^{n-2} = (16)4^{n-2}$$
$$\Rightarrow 20Q = 16 \Rightarrow Q = \frac{4}{5}$$

Therefore, the particular solution is $\frac{4}{5}n4^n$. Hence, the general solution of the recurrence relation of,

$$f(n) = A_1(-1)^n + A_2(4)^n + \frac{4}{5}n4^n$$

***Note:*** What if the characteristic equation gives rise to complex roots? Here, the methods are still valid, but the method for expressing the solutions of the recurrence relations is different. Since an understanding of these representations require some background in complex numbers, it is suggested that an interested reader refer to a more advanced treatement of recurrence relations.

### 4.6.8 Linear Homogeneous Recurrence Relations with Constant Coefficient (LHRRWCC)

We know that a **recurrence relation** is an equation defining a sequence recursively in which each term of the sequence is defined as a function of the preceding terms. A linear homogeneous recurrence relation of $k$-order with constant coefficients is a recurrence relation of the form,

$$a_n = c_1a_{n-1} + c_2a_{n-2} + ... + c_ka_{n-k}$$

Where $c_1, c_2, \ldots, c_k$ are real numbers and $c_k \neq 0$

- The equation is linear since RHS consists of the sum of the previous terms, each multiplied by a function of $n$.
- The equation is homogeneous since there is no term that is not a multiple of $a_j$ s.
- The equation is order $k$ as $a_n$ is expressed in terms of previous $k$ terms in the sequence.
- The equation has constant coefficients $c_1, c_2, \ldots, c_k$.
- The relation is recurrence of $k$th order and has $k$ initial conditions such as $a_0 = c_0$, $a_1 = c_1$, $\ldots$ $a_{k-1} = c_{k-1}$

Thus, the above relation is a linear homogeneous recurrence relation of order $k$ with constant coefficient. The following is an example on the order of the recurrence relation:

| | |
|---|---|
| • $P_n = 2.5\, P_{n-1}$ | Order one |
| • $f_n = f_{n-1} + f_{n-2}$ | Order two |
| • $a_n = a_{n-5}$ | Order five |
| • $a_n = 2a_{n-1} + 3a_{n-2} + 5a_{n-6}$ | Order six |

***Note*:** Terms in a recurrence relation is written either by subscript notation or functional notation.

For example, $f_n = f_{n-1} + f_{n-2}$ is also written as $f(n) = f(n-1) + f(n-2)$.

**Theorem 4.1:** If $\alpha$ and $\beta$ are two distinct (real or complex) solutions of the equation $x^2 - ax - b = 0$, where $a, b \in$ R and $b \neq 0$, then every relation of LHRRWCC $a_n = a{\cdot}a_{n-1} + b{\cdot}a_{n-2}$ where $a_0 = C_0$ and $a_1 = C_1$ is of the form $a_n = A\alpha^n + B\beta^n$ for some constant $A$ and $B$.

**Proof:** This theorem will be proved in two parts:

(*i*) First it will be proved that $a_n = A\alpha^n + B\beta^n$ is a solution of recurrence relation for constants $A$ and $B$. From initial conditions, given values of $A$ and $B$ is determined.

Since $\alpha$ and $\beta$ are roots of the equation $x^2 - ax - b = 0$, $a^2 = a$a $+ b$ and $\beta^2 = a\beta + b$.

Now, it will be shown that $a_n = A\alpha^n + B\beta^n$ is the solution of the recurrence relation,

$$\begin{aligned} a_n &= a{\cdot}a_{n-1} + b{\cdot}a_{n-2}. \\ a{\cdot}a_{n-1} + b{\cdot}a_{n-2} &= a(A\alpha^{n-1} + B\beta^{n-1}) + b(A\alpha^{n-2} + B\beta^{n-2}) \\ &= A\alpha^{n-2}(a\alpha + \mathrm{b}) + B\beta^{n-2}(a\beta + \mathrm{b}) \\ &= A\alpha^{n-2}\alpha^2 + B\beta^{n-2}\beta^2 \\ &= A\alpha^n + B\beta^{n\,i} \\ &= a_n \end{aligned}$$

This proves that $a_n = A\alpha^n + B\beta^n$ is a solution of recurrence relation,

$$a_n = a \cdot a_{n-1} + b \cdot a_{n-2}$$

If $a_n = A\alpha^n + B\beta^n$ is a solution of the recurrence relation, then values of $A$ and $B$ should be calculated. Initial conditions are used to evaluate these two. Initial given conditions are:

$a_0 = C_0$ and $a_1 = C_1$

Using this, you get the following two equations.

$C_0 = A + B$ and $C_1 = A\alpha + B\beta$.

The values of $A$ and $B$ are as:

$A = (C_1 - C_0\beta)/(\alpha - \beta)$ and $B = (C_0\alpha - C_1)/(\alpha - \beta)$ and $\alpha \neq$ .

After these values are determined, $a_n = A\alpha^n + B\beta^n$ is the unique solution.

However, this formula to determine $A$ and $B$ can not be applied when $\alpha = \beta$.

Solution is looked for the form $a_n = r^n$, where $r$ is constant.

Here, $a_n = r^n$ is a solution of recurrence relation given by,

$a_n = c_1a_{n-1} + c_2a_{n-2} + ... + c_ka_{n-k}$ iff $r^n = c_1r^{n-1} + c_2r^{n-2} + c_3r^{n-3} + .... + c_kr^{n-k}$

Dividing both sides by $r^{n-k}$ and bringing RHS to the left, a characteristic equation is obtained which is shown as follows:

$r^k - c_1r^{k-1} - c_2r^{k-2} - c_3r^{k-3} - .... - c_kr^{k-k} \rightarrow r^k - c_1r^{k-1} - c_2r^{k-2} - c_3r^{k-3} - .... - c_kr$

By solving this equation, you get the characteristics roots.

**Example 4.46:** Solve recurrence relation $a_n = 5a_{n-1} - 6a_{n-2}$ where $a_0 = 4$ and $a_1 = 7$.

**Solution:** To solve this problem, we follow the given steps:

To get general solution of the recurrence relation, find the characteristic equation which is givn by $r^2 - 5r + 6 = 0$.

$r^2 - 5r + 6 = (r - 2)(r - 3) = 0$ that leads to characteristic roots as 2 and 3.

Hence, general solution is given by $a_n = A \cdot 2^n + B \cdot 3^n$, where $A$ and $B$ are constants. These are to be determined from the given initial conditions.

Putting $n = 0$, you get $a_0 = A + B = 4$ and $a_1 = 2A + 3B = 7$. Solving equations $A + B = 4$ and $2A + 3B = 7$, you get $A = 5, B = -1$.

Thus, solution is given as $a_n = 5 \cdot 2^n - 3^n$

**Example 4.47:** Solve the recurrence relation $a_n = 6a_{n-1} - 11a_{n-2} + 6a_{n-3}$ where $a_0 = 2$, $a_1 = 5$, and $a_2 = 15$.

**Solution:** General solution is obtained for the given recurrence relation by finding the characteristic equation first which is given by $r^3 - 6r^2 + 11r - 6 = 0 \rightarrow (r - 1)(r - 2)(r - 3) = 0$. Thus characteristic roots are given as 1, 2 and 3.

General solution is given by: $\boldsymbol{a_n = A\cdot 1^n + B\cdot 2^n + C\cdot 3^n}$.

Now evaluate constants by using initial conditions.

Get equations $a_0 = A + B + C = 4$; $a_1 = A + 2B + 3C = 5$ and $a_2 = A + 4B + 9C = 15$. Solving these three equations you will get values of $A$ $B$ and $C$ as $A = 1, B = -1, C = 2$. Thus, the unique solution is:

$a_n = 1 - 2^n + 2\cdot 3^n$ for $n \geq 0$.

**Example 4.48:** Find $f(n)$ if $f(n) = 7f(n-1) - 10f(n-2)$, given that $f(0) = 4$ and $f(1) = 17$.

**Solution:** The characteristic equation of the given recurrence relation is $r^2 - 7r + 10 = 0$

Its characteristic roots are $r = 2, 5$. So, the general solution of the recurrence relation is,

$$f(n) = A_1 (2)^n + A_2 (5)^n$$

Since, $f(0) = 4, A_1 + A_2 = 4$

Again, $f(1) = 17$ implies $2A_1 + 5A_2 = 17$

Solving these two equations, You will get $A_1 = 1$ and $A_2 = 3$. Therefore, unique solution is:

$$f(n) = (2)^n + 3(5)^n$$

***Note:*** If roots of a characteristic equation are equal, i.e., $\alpha = \beta$, then the general solution is a bit different in form and this has a form $a_n = (A_1 + A_2 n)\,\alpha^n$. This is illustrated in the following examples.

**Example 4.49:** Solve $f_k - 8f_{k-1} + 16f_{k-2} = 0$, where $f_2 = 16$ and $f_3 = 80$.

**Solution:** Characteristic equation is $r^2 - 8r + 16 = 0 \to (r-4)^2 = 0$

So, $r = 4$ is a double characteristic root. Therefore, the general solution is $f_k = (A_1 + A_2 k)\, 4^k$

Now, $f_2 = 16 \Rightarrow (A_1 + 2A_2)\, 16 = 16$

and $f_3 = 80 \Rightarrow (A_1 + 3A_2)\, 64 = 80$

Solving, you get $A_1 = 1/2$ and $A_2 = 1/4$ and hence, unique solution is,

$$fk = (1/2 + k/4)\, 4k = (2 + k)4k-1$$

**Example 4.50:** Find a solution to the recurrence relation

$C_n = -3C_{n-1} - 3C_{n-2} - C_{n-3}$ for $n \geq 3$ with initial conditions $C_0 = 1$, $C_1 = -2$ and $C_2 = 1$.

**Solution:** The characteristic equation is $r^3 + 3r^2 + 3r + 1 = 0$ or $(r+1)^3 = 0$.

So, $r = -1$ is a characteristic root of multiplicity 3. Therefore, the general solution is,

$$C_n = (A_1 + A_2 n + A_3 n^2)\,(-1)^n$$

Now, $C_0 = 1 \Rightarrow A_1 = 1$

$$C_1 = 2 \Rightarrow -(A_1 + A_2 + A_3) = -2$$
$$C_2 = 1 \Rightarrow A_1 + 2A_2 + 4A_3 = 1$$

Solving, we get $A_1 = 1, A_2 = 2, A_3 = 1$. Hence, the solution is,

NOTES

$C_n = (1 + 2n - n^2)(-1)^n$.

### 4.6.9 Divide and Conquer Recurrence Relation (DCRR)

There are many recurrence relations that are not linear or are linear with variable coefficients. Such problems are complex and require some strategy to find a method or design an algorithm to solve such recurrence relations. This strategy is known as 'divide and conquer' strategy. A problem is divided into many smaller problems and such reduction is done repeatedly to find solutions of smaller problems quickly. This procedure is called divide and conquer.

***Note:*** We will use functional form of notation for writing recurrence relation that describes a function of time- complexity for an algorithm, and $f(n)$ will be written instead of $a_n$.

Let the given problem be the computation of $f(n)$ and strategy is 'divide and conquer; so it is divided into $b$ number of small sub-problems of the same type. Each problem size can be given by floor-function $\lfloor n/b \rfloor$ or ceiling-function $\lceil n/b \rceil$. In either case, the result is an integer. The basic idea is to know roughly the time required to compute $f(n)$ using a 'divide and conquer' algorithm. The process of solving the smaller problems may be shown by a function and let this function be $h(n)$. Now, the function $f(n)$ can be written as $f(n) = af(n/b) + h(n)$ to get an idea of the time that may be required to solve $f(n)$ using such strategy. You will be calculating to get an approximation about the time needed to compute $f(n/b)$ and time to compute $h(n)$.

For this an O notation is used and is denoted as $O(f(n))$. It is said that $O(f(n))$ is the greater of $O(f(n/b))$ and $O(h(n))$. If you do not know $O(f(n))$, then you also do not know $O(f(n/b))$; hence you cannot compare $O(f(n/b))$ and $O(h(n))$. Thus, you need tools to help, you find the answer. You may compute $f(n)$ for $n = 1$ and find $f(1) = c$ and $f(n) = af(n/b)+c$, $n = b^k$, $k \in Z^+$.

The basic idea is find the asymptotic bound for $f(n)$.

**Theorem 4.2:**

Let $f$ be an increasing function that satisfies the rec. rel. $f(n) = af(n/b) + c$, whenever $n$ is divisible by $b$, where $a, b \in N$, $b > 1$ and $c \in R$, $c > 0$, then $f(n)$ is $O(n^{\log_b a})$ if $a > 1$ and $O(logn)$ if $a = 1$.

Furthermore, when $n = b^k$, $k \in N$, then $f(n) = C_1 n^{\log_b a} + C_2$, where $C_1 = f(1) + c/(a-1)$ and $C_2 = -c/(a-1)$.

**1. Binary Search**

Binary Search is an example of 'divide and conquer' policy. If a function $f(n)$ denotes numbers of comparisons required for searching an element in the list with, size $n$ then let us take $n$ as even. In this, the search list is reduced to two lists each of size $n/2$. Then there are two types of comparisons needed, one to check that part of the list for use and the other to check whether there is any term remaining in the list. So, $f(n)=f(n/2)+2$ for even $n$.

**2. Finding Maximum and Minimum in a List**

Let $\{a_1, a_2, \ldots, a_n\}$ denote a list. For $n = 1$, there is a single list $a_1$ which is both maximum and minimum for $n > 1$, and $f(n)$ as total numbers of comparisons for finding maximum and minimum elements in the list of $n$ elements. If $n$ is even, then list is reduced to two lists of equal elements, otherwise if $n$ is odd, one sub-list will have one element more than the other one. Here also two comparisons are required, one that makes comparison for maximum and another for minimum of the two sub-lists. Hence, the recurrence relation is $f(n)=2f(n/2)+2$ for even $n$.

**3. Fast Multiplication of Integers**

For this job, 'divide and conquer' strategy is used. Suppose $a$ and $b$ are $2n$-bit integers. You split each into two blocks with each block having $n$-bits. Integers $a$ and $b$ have binary expansions of length $2n$. Further,

$a = (a_{2n-1}a_{2n-2}\ldots a_1a_0)_2$, $\boldsymbol{b} = (b_{2n-1}b_{2n-2}\ldots b_1b_0)_2$ $a = 2^nA_1 + A_0$, $b = 2^nB_1 + B_0$ and where $A_1=(a_{2n-1}\ldots a_{n+1}a_n)_2$, $A_0=(a_{n-1}\ldots a_1a_0)_2$, $B_1=(b_{2n-1}\ldots b_{n+1}b_n)_2$, $B_0=(b_{n-1}\ldots b_1b_0)_2$. So, we can write $ab = (2^{2n} + 2^n)\,A_1B_1 + 2^n(A_1 - A_0)(B_0 - B_1) + (2^n + 1)\,A_0B_0$. Thus multiplication of two $2n$-bit integers may be carried by use of multiplication of three $n$-bit integers combined by some shifts, subtractions and additions. Thus, if $f(n)$ stands for total number of bit operations required to multiply two $n$-bit integers, then this can be mathematically denoted as $f(2n)=3f(n)+Cn$. Here, $Cn$ gives the number of shifts, subtractions and additions required to carry out multiplication of three $n$-bit integers which is $3f(n)$.

**4. Fast Matrix Multiplication**

Matrix multiplications between two $n \times n$ matrices need $n^3$ multiplications and $n^2(n-1)$ additions and such an operation is $O(n^3)$. This obviously is high resource consuming operation. Adopting 'divide and conquer' strategy, multiplication of two $n \times n$ matrices can be reduced to 7 multiplication and 15 additions of two half size matrices. This reduction was used by V. Strassen. If $f(n)$ shows the number of both the operations, multiplications and additions, then $f(n)=7f(n/2)+15\,/4$ for even $n$.

So, in 'divide and conquer' strategy, the recurrence relation of the type $f(n)=af(n/b)+g(n)$ appears mostly. Here, instead of $g(n)$, $h(n)$ can also be written. To solve this, let us assume that $f$ satisfies the recurrence relation whenever, $n = b^k$, $k \in N$, where $k \in N$. Then,

$$f(n) = a\,f(n/b) + g(n)$$

$$= a^2 f(n/b^2) + ag(n/b) + g(n)$$
$$= a^3 f(n/b^3) + a^2 g(n/b^2) + ag(n/b) + g(n)$$
$$\vdots$$
$$= a^k f(n/b^k) + \sum_{j=0}^{k-1} a^j g(n/b^j)$$

Since, $n = b^k$, we have $f(n) = a^k f(1) + \sum_{j=0}^{k-1} a^j g(n/b^j)$

This equation can be used for estimating the size of functions satisfying 'divide and conquer' recursive relations.

**Example 4.51:** Let $f(n)=5f(n/2)+3$ and f(1)=7. Find $f(2^k)$, k $\in$N. Also estimate $f(n)$ assuming $f$ as an increasing function.

**Solution:** By applying theorem 4.2 with $a = 5$, $b = 2$, $c = 3$, if $n = 2^k$ then $f(n) = 5^k(31/4) - 3/4$. If $f$ is increasing, then by theorem 4.2, $f(n)$ is $O(n^{\log 5})$.

**Example 4.52:** Estimate the number of comparisons by a binary search.

**Solution:** As you know, $f(n)=f(n/2)+2$ when $n$ is even. If $f(n)$ denotes the number of comparisons for ascertaining whether an element $x$ exists in a list of size $n$, then by theorem 4.2 where $a = 1$, $b = 2$, $c = 2$, $f(n)$ is O(log$n$).

**Example 4.53:** Estimate the number of comparisons required to find maximum and minimum elements of the list with $n$ elements.

**Solution:** This type of problem has been discussed in earlier examples. Here, $f(n)=2f(n/2)+2$ for even $n$, if $f(n)$ denotes the number of comparisons for finding maximum and minimum on the list. Applying theorem 4.2 where $a = 2$, $b = 2$, $c = 2$, it is clear that $f(n)$ is $O(n^{\log_b a}) = O(n)$.

Hence, a more general version of the theorem 4.2 is obtained known as Master Theorem; it is used in complexity analysis of many 'divide and conquer' algorithms.

**Master Theorem**

Let $f$ be an increasing function that satisfies the recurrence relation

$f(n) = af(n/b) + cn^d$, whenever $n = b^k$, $k \in N$, where $a, b \in N$, $b > 1$ and $c, d \in R$, $c > 0$, $d$ is non-negative. Then $f(n)$ is $O(n^d)$ if $a < b^d$, $O(n^d logn)$ if $a = b^d$ and $O(n^{\log_b a})$ if $a > b^d$.

**Example 4.54:** Estimate the number of bit operations needed to multiply two $n$-bit integers using the fast multiplication algorithm for $f(n) = 3f(n/2) + C_n$ where $n$ is even.

**Solution:** Here, $f(n)=3f(n/2)+Cn$ when $n$ is even and $f(n)$ denotes the number of bit operations required for multiplication of two $n$-bit integers using the fast multiplication algorithm. Applying the Master Theorem for $a = 3$, $b = 2$, $d = 1$, it is clear that f(n) is $O(n^{\log_2 3}) = O(n^{1.6})$. This is a faster method than $O(n^2)$.

According to Master Theorem, let there be two constants $a$ and $b$, where $a \geq 1, b > 1$, and a function $f(n)$ is defined recursively by $f(n) = af(n/b) + h(n)$.

Here $n/b$ is either taken as 'floor function' or 'ceiling function' giving integral value in either case.

Then, $f(n)$ is bounded asymptotically. These may be expresed as follows:

1. If $h(n) \in O(n^{(\log_b a)-\in})$ for some $\in > 0$, then $f(n)$ $O(n^{\log_b a})$.
2. If $h(n) \in O(n^{(\log_b a)})$, then $f(n) \in O(n^{\log_b a} \ln n)$.
3. If $h(n) \in O(n^{(\log_b a)+\in})$ for some $\in > 0$, and if $a\, h(n/b) \leq c\, h(n)$ for some $0 \leq c < 1$ and for sufficiently large $n$, $f(n) \in O(h(n))$.

The role played by the term $n^{\log_b a}$ is important.

If you take $n = b^k$, then $f(n) = f(b^k) =$

$af(b^{k-1}) = a^2f(b^{k-2}) = \cdot \cdot \cdot = akf(1)$. However, $a^k = a^{\log_b n} = (b^{\log_b a})^{\log_b n} = b^{(\log_b n)(\log_b a)} = n^{\log_b a}$

So, it will be $f(n) \in O(n^{\log_b a})$ in the homogeneous case.

Thus, the interpretation of this theorem is that if $h(n)$ grows slower than the homogeneous case, the homogeneous case is dominant and the inhomogeneous $f(n) \in O(n^{\log_b a})$. If the rate of growth is the same then the inhomogeneous $f(n) \in O(n^{\log_b a} \ln n)$. Lastly, if $h(n)$ grows faster than the homogeneous case, then $f(n) \in O(h(n))$.

**CHECK YOUR PROGRESS**

10. What is an iterative procedure used for?
11. When is a linear relation considered to be a homogeneous recurrence relation?
12. What is the 'divide and conquer' strategy?

## 4.7 RECURSIVE PROCEDURES

A recursive procedure is a unique method of defining functions. In it, the function is applied within its own definition. This term also describes the process of repetition in a similar way. An example of recursion is seen when reflecting surfaces of two mirrors are placed parallel to each other. You see the nesting of images and is a form of recursion.

Mathematics and computer science do a lot to define rules and apply these rules in breaking down complex cases into simpler ones. They do it by defining few simple base cases or methods and build recursions on that. These base cases or methods are kept to minimum, preferably just one.

For example, we take the case of defining ancestors.

***Base case:*** Parents are ancestors.

***Recursion step:*** Parents of ancestors are also ancestors.

NOTES

Thus putting this fact in simple words, recursion defines objects in terms of 'previously defined' objects belonging to that class.

Such facts are often seen in mathematics. For example, in a set of natural numbers, 1 is a natural number and each natural number has a successor, which in turn is also a natural number.

Functions, sets, and fractals are examples of mathematical objects defined recursively. A fractal is based on the property of self similarity. If you take a fragmented geometrical shape such that each fragmented part is a reduced size copy of the original whole it is known as fractal. Recursion is in use in India since 5th century when the ancient Indian linguist Pânini used the principle of recursion in framing rules for grammar of Sanskrit language.

### 4.7.1 Functional Recursion

Common examples of functional recursion are Fibonacci number sequence, Ackermann function, Lucas number sequence, etc.

A function that is partly defined in terms of itself is also a recursive function. A familiar Fibonacci number sequence is given by $F(n) = F(n–2) + F(n–1)$ where $n > 2$, is an example. To make it useful, few values are non-recursively defined. In Fibonacci number sequence the initial two values are defined non-recursively. These are $F(0) = 0$ and $F(1) = 1$. Here $f(0)$ is the first term and for this reason the condition $n > 2$ is given in the definition of Fibonacci number sequence.

Ackermann function is not like Fibonacci sequence. It is always expressed with recursion and it is not primitive recursive. If $P$ is a set of primitive recursive functions and $R$ is that of general recursive functions, then $P$ is a subset of $R$.

The Ackermann function, defined recursively for non-negative integers, $m$ and $n$, are given as,

$$A(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1 & \text{if } m > \text{and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Now take an example of Lucas number sequence which is defined as, $L_1 = 1$, $L_2=3$ and $L_{x+1} = L_x + L_{x-1}$ for $x \geq 2$, where $x$ a positive integer. This is like Fibonacci number sequence in which initial two values are defined non-recursively. For example, to find the first six terms of this recursive sequence the follwing method is used.

From the definition of Lucas number sequence, $L_3 = L_2 + L_1 = 1+ 3 = 4$, $L_4 = L_3 + L_2 = 4 + 3 = 7$, $L_5 = L_4 + L_3 = 7 + 4 = 11$ and $L_6 = L_5 + L_4 = 11 + 7 = 18$.

So the first six terms are 1, 3, 4, 7, 11 and 18.

We now give example of factorial function defined as $f(n) = n!$.

This leads to $n! = n(n-1)! = n(n-1)(n-2)! \rightarrow f(n) = n.f(n-1) = n(n-1).$ $F(n-2)$. Here, $n \in N$, set of natural numbers.

Catalan numbers are another example of recursive functions; it is given as:

$C_0 = 1, C_{n+1} = (4n+2)C_n / (n+2)$

Catalan numbers form a sequence of natural numbers that exists in the problems using recursively defined objects in combinational mathematics.

### 4.7.2 Recursive Proofs

New systems in mathematics or logical constructs are defined as 'true' and 'false' or 'all natural numbers'. These are taken as base cases and after this, subsequent computations in the system are made according to predefined rules. If base cases and rules as predefined are computable, then any formula can be computed.

### 4.7.3 The Recursion Theorem

This theorem guarantees the existence of recursively defined functions. Let $X$ be a set whose element is $x$ and a function $f: X \rightarrow X$. The theorem opines the existence of a unique function $F : N \rightarrow X$, where $N$ stands for the set of natural numbers and zero. The function is stated as,

$F(0) = x$

$F(n+1) = f(F(n))$ for any natural number $n$.

**Proof of Uniqueness**

Let there be two functions $f$ and $g$. Domain of $f$ is $N$ and codomain of $g$ is $X$ such that,

$f(0) = x$

$g(0) = x$

$f(n+1) = F(f(n))$

$g(n+1) = F(g(n))$, where $x$ is an element of $X$.

It is required to prove that $f = g$.

Equality of two functions is possible when they have equal domains/ codomains and follow the same curve.

**Recursively Defined Functions and Procedures**

The basic requirement for a function to be recursive is that at least one value $f(x)$ is defined in terms of another value $f(y)$, when $x \neq y$. In a similar way, if there is a procedure $P$, it is defined recursively only when the action of $P(x)$ is defined in terms of another action $P(y)$, provided $x \neq y$.

NOTES

The argument domain is already inductively defined.

Steps in defining recursive functions:

1. A value $f(x)$ or an action $P(x)$ is to be specified for each basis element $x$ of $S$.
2. Rules for each inductively defined element $x$ in $S$ is to be specified. Value $f(x)$ or action $P(x)$ is to be defined in terms of previously defined values.

**Example 4.55:** A function $f : N \to N$ is defined as $f(n) = 0 + 3 + 6 + \ldots + 3n$. Develop a recursive definition for this function.

**Solution:** The set $N$ which is inductively defined should be understood first. This includes 0 too, hence $0 \in N$ and $n \in N \to n + 1 \in N$. Now $f(0)$ should be given a value in $N$ and then define $f(n + 1)$ in terms of $f(n)$. As given, set $f(0) = 0$. Next a definition for $f(n + 1)$ is obtained. According to definition of $f(n)$,

$f(n + 1) = (0 + 3 + 6 + \ldots + 3n) + 3(n + 1) = f(n) + 3(n + 1)$.

Thus, a recursive definition has been developed as, $f(0) = 0$ and $f(n + 1) = f(n) + 3(n + 1)$.

This recursive function can be stated in different ways:

- Putting $n - 1$ in place of $n$, you get $f(0) = 0$ and $f(n) = f(n - 1) + 3n$ ($n > 0$).
- It may be expressed as a conditional statement. If $n = 0$, then $f(n) = 0$ else $f(n) = f(n-1) + 3n$.

**Example 4.56:** A function $f : N \to N$ is defined recursively as $f(0) = 0$, $f(1) = 0$ and $f(x + 2) = 1 + f(x)$. This function can be written as a conditional statement. If $x = 0$ or $x = 1$, then $f(x) = 0$; else $1 + f(x - 2)$. What does $f$ do?

**Solution**: Following the rule of the function, find few values of the function for $x = 0$ to 9. $f(0) = 0, f(1) = 0, f(2) = 1, f(3) = 1$... and like that $f(8) = 4$ and $f(9) = 4$. Now do the mapping $(f(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)) = (0, 0, 1, 1, 2, 2, 3, 3, 4, 4)$ and find $f(x) = [x/2]$.

### 4.7.4 Infinite Sequences

It is possible to define recursive functions for infinite sequences. For this, define a value $f(x)$ in terms of $x$ and $f(y)$ for some value $y$ in a sequence.

**Example 4.57:** An infinite sequence $f(x) = (x, x^2, x^4, x^8, \ldots)$ is to be represented as a recursive function.

**Solution:** Follow the given definition and develop a solution:

The function is defined as,

$f(x) = (x, x^2, x^4, x^8, \ldots) = (x :: (x^2, x^4, x^8, \ldots)) = x :: f(x^2)$.

So the developed definition is,

$f(x) = x :: f(x^2)$.

NOTES

**Example 4.58:** A recursive function is defined as $g(x, c) = x^c :: g(x, c + 1)$. Find the sequence represented by this function.

**Solution:** According to definition,

$g(x, c) = x^c :: g(x, c + 1) = x^c :: x^{c+1} :: g(x, c + 2) = \ldots = (x^c, x^{c+1}, x^{c+2}, \ldots)$.

### 4.7.5 Recursive Function and Primitive Recursive Function

Consider a function, $g(x_1, x_2, \ldots x_n)$ and $h(x_1, x_2, \ldots x_n, y, z)$ of $n$ and $n+2$ variables. Now define a function $f(x_1, x_2, \ldots x_n, y)$ of $n+1$ variables as $f(x_1, x_2, \ldots x_n, 0) = g(x_1, x_2, \ldots x_n,)$ and $f(x_1, x_2, \ldots x_n, y + 1) = h(x_1, x_2, \ldots x_n, ,y, f(x_1, x_2, \ldots x_n, y))$; here $f$ is called a recursive function or simply a recursion. A function $f$ is primitive recursive if it can be obtained from initial functions by a finite number of operations of composition and recursion.

**CHECK YOUR PROGRESS**

13. What do you understand by a recursive procedure?
14. Give some examples of functional recursion.

## 4.8 SUMMARY

In this unit, you have learned that:

- The Mergesort algorithm works according to a 'divide and conquer' strategy in which the sequence is divided into two halves.
- In the merging process, the elements of two arrays are combined, creating a new array.
- In insertion sorting algorithm, the sorted array is built one entry at a time.
- The ordered sequence of inserted elements is stored at the beginning of the array.
- Each iteration of the inner loop scans and shifts the entire sorted subsection of the array before the next element is inserted.
- Sorting can be classified into two types, viz., internal sorting and external sorting.
- A binary system groups numbers by two and by powers of two.
- A binary number with 4 bits is called a nibble and a binary number with 8 bits is called a byte.
- A number system that utilizes ten distinct digits, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 is known as a decimal number system.
- A binary number can be converted into decimal number by multiplying the binary 1 or 0 by the weight corresponding to its position and adding all the values.

NOTES

- The numbers in the sequence 0, 1, 2, 3, 5, 8, 13, 21 … in which each new term is the sum of the previous two terms are called Fibonacci numbers.
- A recursive relation is an equation defining a sequence in which each term of the sequence is defined as a function of the preceding terms.
- A $k$th order linear relation is a homogeneous recurrence relation if $f(n)=0$ for all $n$.
- The general solution of the recurrence relation is the sum of the homogeneous solution and particular solution.

## 4.9 KEY TERMS

- **Selection sort:** It refers to a simple technique of sorting a list of elements by finding the smallest value in an array.
- **Binary number system:** It refers to the number system in which only two digits, viz., 0 and 1, are used.
- **Nibble:** A binary number with 4 bits is called a nibble.
- **Double-dabble method:** It refers to a method that is used to convert a large decimal number into its binary equivalent by repeatedly dividing it by 2.
- **Recursive relation:** It is an equation defining a sequence in which each term of the sequence is defined as a function of the preceding terms.

## 4.10 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. The Mergesort algorithm is based on the merging process where all the elements are copied in one array and kept in a separate new array.
2. Two advantages of insertion sort are as follows:
   (i) Its implementation is simple.
   (ii) It can sort a list even as it receives it.
3. The insert function is designed for inserting a value into a sorted sequence at the beginning of an array.
4. Sorting is a method of arranging keys in a file in the ascending or descending order.
5. Internal sorting occurs when the records in a file that are stored in the main memory are sorted.
6. Two common sorting techniques are:
   (i) Bubble sort
   (ii) Tree sort

7. A base two system is another term for a binary number system. Such a number system uses two digits, 0 and 1, only.
8. A binary number system is used in digital computers because all electrical and electronic circuits can be made to respond to the two-state concept.
9. One method to convert a decimal number into a binary number is to subtract values of power of 2 from the decimal number until nothing remains.
10. An iterative procedure is used to evaluate a function from its recursive definition.
11. A *k*-th order linear relation is considered to be a homogeneous recurrence relation if $f(n)=0$ for all $n$.
12. When a problem is divided into many smaller problems and such reduction is done repeatedly to find solutions of smaller problems, such a procedure is called a 'divide and conquer' strategy.
13. A recursive procedure is a unique method of defining functions. In it, the function is applied within its own definition. This term also describes the process of repetition in a similar way.
14. Common examples of functional recursion are Fibonacci number sequence, Ackermann function, Lucas number sequence, etc.

## 4.11 QUESTIONS AND EXERCISES

**Short-Answer Questions**

1. Give an algorithm illustrating the use of Mergesort mechanism.
2. Write a short note on the Insertion sort algorithm.
3. What are the different types of sorting algorithms?
4. Differentiate between the binary number system and decimal number system.
5. Write a short note on the double-dabble method.

**Long-Answer Questions**

1. Write a program illustrating the use of Insertion sort.
2. Explain the functions of Bubble sort and Selection sort. Write one program each to illustrate their implementation.
3. Explain the various techniques that are used to convert binary numbers into decimal numbers and vice versa.
4. What do you understand by recursion and recurrence relations? Give examples to illustrate both the relations.

## 4.12 FURTHER READING

NOTES

Lipschutz, Seymour and Lipson Marc. *Schaum's Outline of Discrete Mathematics,* 3rd edition. New York: McGraw-Hill, 2007.

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms.* Hyderabad: Orient BlackSwan, 2008.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

Brassard, Gilles and Paul Bratley. *Fundamentals of Algorithms*. New Delhi: Prentice Hall of India, 1995.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. New Jersey: Pearson, 2006.

Baase, Sara and Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. New Jersey: Pearson, 2003.

Mott, J.L. *Discrete Mathematics for Computer Scientists*, 2nd edition. New Delhi: Prentice-Hall of India Pvt. Ltd., 2007.

Liu, C.L. *Elements of Discrete Mathematics*. New Delhi: Tata McGraw-Hill Publishing Company, 1977.

Rosen, Kenneth. *Discrete Mathematics and Its Applications*, 6th edition. New York: McGraw-Hill Higher Education, 2007.

# UNIT 5 NUMBER THEORY

**NOTES**

**Structure**

## 5.0 INTRODUCTION

Numbers form the very basis for any type of calculation. There are primarily two types of numbers—prime and composite numbers. A prime number refer to those numbers which have only 1 and the number itself as its factors and does not contain any multiple. In this unit, you will learn about the various types of numbers, their properties and their applications.

A common calculation that is used frequently is the finding of the greatest common divisor (GCD) and the Euclid's algorithm. You will learn about the application of GCD, least common multiple (LCM), and Euclidean algorithm.

Fibonacci numbers refer to numbers that are a sum of the previous two numbers. Among other things, this unit will talk about Fibonacci numbers, congruent relations, equivalence relations and also encryption schemes.

## 5.1 UNIT OBJECTIVES

NOTES

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

- Understand the basics of the number theory
- Identify the greatest common divisor (GCD), Euclidean algorithm and Fibonacci numbers
- Explain the concepts of congruence and equivalent relations
- Comprehend the public key encryption schemes and their application

## 5.2 NUMBER THEORY: BASICS

### 5.2.1 Fundamental Theorem of Arithmetic

Let $a$, $b$ be two integers such that $a = bc$, where $b$ and $c$ are called factors of $a$ and $a$ is called *multiple* of $b$ and $c$. If $b$ is a factor of $a$, this implies that $b$ divides $a$ and it is written as $b \mid a$. It is easy to prove that $\pm 1$ are factors of $a$ and every non-zero integers is a factor of 0.

For example, $6 = (-2)(-3) \quad \Rightarrow -2$ and $-3$ are factors of 6.

Again $7 = 7.\ 1$, so 7 and 1 are factors of 7.

An integer $n$ different from $\pm 1$ is called a prime integer provided its only factors are $\pm 1$ and $\pm n$.

Thus, $2, -3, 7, 11, -13$ are some prime integers, while $-4, 16, 12$ are not prime integers as $-4 = 2(-2)$, $16 = 4 \,.\, 4$, $12 = 6 \,.\, 2$

A positive prime integer is called a prime number.

**Properties of Prime Numbers**

(*i*) If a prime number $p$ divides $ab$ then either $p \mid a$ or $p \mid b$.

(*ii*) If a prime number $p$ does not divide an integer $a$ then the Highest Comman Factor (HCF) of $p$ and $a$ is 1 (by convention take HCF to be $a$ +ve integer).

**Fundamental Theorem of Arithmetic**

Every integer $n > 1$ can be factorized into a product of finite numbers of prime numbers. This expression is unique except for the order in which the prime factors are written.

For example, $56 = 2 \,.\, 2 \,.\, 2 \,.\, 7 = 2^3 \,.\, 7$

$72 = 2 \,.\, 2 \,.\, 2 \,.\, 3 \,.\, 3 = 2^3 \,.\, 3^2$

### 5.2.2 Prime Numbers

An integer $p > 1$ is called a *prime number* if 1 and $p$ are the only divisors of $p$.

We prove it using Greatest Comnon Divisor (GCD) which is normally represented as g.c.d.

**Theorem 5.1:** *If a prime number p divides ab, then either p divides a or p divides b.*

**Proof:** Let $ab = pc$ for some integer $c$.

Suppose $p$ does not divide $a$.

Then, g.c.d. $(a, p) = 1$

$\therefore$ $p \mid ab$ and g.c.d.$(a, p) = 1$

$\Rightarrow p \mid b$

This result can be generalized in the following theorem.

**Theorem 5.2:** *If p divides $a_1, a_2, ..., a_n$, then p divides $a_i$ for some i.*

**Proof:** This result can be proved by induction on $n$.

If $n = 1$, then this result is clearly true.

If $n = 2$, then the result is true.

Let the result be true for natural numbers less than $n$.

Suppose, $p \mid a_1 \ldots a_n = (a_1 \ldots a_{n-1})a_n$

$\Rightarrow p \mid a_1 \ldots a_{n-1}$ or $p \mid a_n$

If $p$ divides $a_1, ..., a_{n-1}$, then by induction hypothesis, $p$ divides $a_i$ for some $i$.

So, the result is true in this case also.

By induction, result is true for all $n > 1$.

**Composite Numbers**

A composite number is an integer $n > 1$ such that $n$ is not prime. It is a positive integer with a positive divisor other than one or itself. For example, the integer 14 is a composite number as it can be factorized as $2 \times 7$. The integers 2 and 3 are not composite numbers because these can only be divided by one and the number itself. The example of composite numbers are 4, 6, 8, 9, 10, 12, 14, 15, 16 etc.

**Example 5.1:** Prove that if $2^n - 1$ is prime, then $n$ is also prime.

**Solution:** Let $2^n - 1 = p =$ Prime.

Let $n$ is not prime.

Then, $n = rs$, $1 < r, s < n$

$$\begin{aligned} \therefore \quad p &= 2^n - 1 \\ &= 2^{rs} - 1 = (2^r)^s - 1 \\ &= x^s - 1, \ x = 2^r > 2 \text{ as } r > 1 \\ &= (x - 1)(x^{s-1} + x^{s-2} + \ldots + x + 1) \end{aligned}$$

Either, $x - 1 = 1$ or, $x^{s-1} + ... + x + 1 = 1$

$x - 1 = 1 \Rightarrow x = 2$, which is not true.

And $x^{s-1} + ... + x + 1 = 1$

$\Rightarrow$ $x^{s-1} + ... + x = 0$, which is not true.

$\therefore$ $n$ is prime.

**Example 5.2:** Prove that $n^4 + 4$ is composite if $n > 1$.

**Solution:** $n^4 + 4 = (n^2 + 2)^2 - 4n^2$

$= (n^2 + 2 - 2n)(n^2 + 2 + 2n)$

$n > 1 \Rightarrow n \geq 2 \Rightarrow n^2 \geq 2n \Rightarrow n^2 - 2n \geq 0$

$\Rightarrow n^2 - 2n + 2 \geq 2$

Also, $n^2 + 2 + 2n > 1$

$\therefore$ $n^4 + 4$ is composite as both $n^2 + 2 + 2n$ and $n^2 + 2 - 2n < n^4 + 4$.

### 5.2.3 Division Algorithms

In mathematics, the division algorithm is a theorem which expresses the outcome of the usual process of division of integers. It is a well-defined procedure for achieving a specific task and can be used to find the greatest common divisor of two integers. The term 'division algorithm' is the study of algebra. Specifically, the division algorithm states that for given two integers $a$ and $d$ with $d \neq 0$, there exist unique integers $q$ and $r$ such that $a = qd + r$ and $0 \leq r < |d|$, where $|d|$ denotes the absolute value of $d$.

In such a case, the integer:

- $q$ is called the *quotient*
- $r$ is called the *remainder*
- $d$ is called the *divisor*
- $a$ is called the *dividend*

The following examples will make the concept clear:

- If $a = 9$ and $d = 4$, then $q = 2$ and $r = 1$, since $9 = (2)(4) + 1$
- If $a = 9$ and $d = -4$, then $q = -2$ and $r = 1$, since $9 = (-2)(-4) + 1$

**Existence:** Consider the set $S = \{ a - nd : n \in Z \}$

Here, $S$ contains at least one non-negative integer and the following two cases can be considered:

- If $d < 0$, then $-d > 0$, and by the Archimedean property, there is a non-negative integer $n$ such that $(-d)n \geq -a$, i.e., $a - dn \geq 0$.
- If $d > 0$, then again by the Archimedean property, there is a non-negative integer $n$ such that $dn \geq -a$, i.e., $a - d(-n) = a + dn \geq 0$.

Each case shows that $S$ contains a non-negative integer. This means that the well-ordering principle can be applied to deduce that $S$ contains a *least* non-negative integer $r$. Let $q = (a–r)/d$, then $q$ and $r$ are integers and $a = qd + r$. It shows that $0 \leq r < |d|$.

### 5.2.4 Divisibility

The divisibility of any number is determined on the basis whether a given number is evenly divisible by other numbers. There are standard divisibility rules for testing a number's factors without resorting to division calculations. Divisibility rules can be created for any base including decimal numbers. A divisor, in mathematics, is an integer $n$, also termed as factor of $n$, which evenly divides $n$ without leaving any remainder. For example, to divide $m$ by $n$, you can write $m/n$ and read as $m$ divides $n$ for non-zero integers $m$ and $n$, iff there exists an integer $k$ such that $n = km$. The divisors can be both positive and negative. Numbers which are evenly divisible by 2 without leaving any remainder are called even numbers and numbers not evenly divisible by 2 are called odd numbers. Further, a divisor of $n$ that is not $1, –1, n$ or $–n$, which are basically the *trivial divisors*, is known as a *non-trivial divisor*. Prime numbers do not have non-trivial divisors while the composite numbers have non-trivial divisors. The term is derived from the arithmetic operation of division. If $a/b = c$ then $a$ is the dividend, $b$ the divisor, and $c$ the quotient.

The following are some elementary rules of divisibility:

- If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$, in fact, $a \mid (mb + nc)$ for all integers $m, n$.
- According to transitive relation, if $a \mid b$ and $b \mid c$, then $a \mid c$.
- If $a \mid b$ and $b \mid a$, then $a = b$ or $a = –b$.
- According to Euclid's lemma, if $a \mid bc$ and g.c.d.$(a, b) = 1$, then $a \mid c$.

A positive divisor of $n$ which is different from integer $n$ is termed as *proper divisor* or *aliquot part* of $n$ while a number which does not evenly divide $n$, but leaves a remainder is termed as *aliquant part* of $n$. An integer $n > 1$, whose only proper divisor is 1, is called a prime number. Mathematically, a prime number is one which has exactly two factors, i.e., 1 and the number itself. Also, any positive divisor of $n$ is a product of prime divisors of $n$ raised to some power.

*Definition.* A non-zero integer $a$ is said to divide an integer $b$ if $b = ac$ for some integer $c$ and is expressed as $a \mid b$.

The following results can be proved:

(*i*) $a \mid b,\ b \mid c$ then $a \mid c$

(*ii*) $a \mid b,\ a \mid c$ then $a \mid b + c$

(*iii*) $a \mid 0,\ a \mid a$

### 5.2.5 Absolute Value

NOTES

For any $a \in \mathbf{R}$, $|a|$ is defined as follows:

$$|a| = a \quad \text{if } a \geq 0$$
$$= -a \quad \text{if } a < 0$$

Thus, $|-\sqrt{2}| = \sqrt{2}$, $|0| = 0$, $|5.7| = 5.7$

*Definition.* $|a|$ is called *absolute value* of $a$.

***Note:*** It follows from definition that for all $x \in R$, $|x| \geq 0$.

***Properties of Absolute Value Function:***

(1) For all $a \in R$, $|a| \geq a$.

**Proof:** If $a \geq 0$ then, $|a| = a$. As $a \geq a$, you get $|a| \geq a$.

In case $a < 0$ then, $a + (-a) < 0 + (-a) \Rightarrow 0 < -a \Rightarrow -a > 0$

Now $-a > 0$ and $0 > a \Rightarrow -a > a$

But for $a < 0$, $|a| = -a$. Hence $|a| > a$. Then, by definition of '$\geq$', $|a| \geq a$.

(2) For all $a, b \in \mathbf{R}$, $|ab| = |a|\,|b|$

**Proof:** The following four cases occur:

**Case I.**

$a \geq 0, b \geq 0$

By definition, $|a| = a$, $|b| = b$. Further, $a \geq 0, b \geq 0$ imply $ab \geq 0$.

Thus, $|ab| = ab = |a|\,|b|$

**Case II.**

$a < 0, b \geq 0$

In this case $|a| = -a$ and $|b| = b$. But $a < 0, b \geq 0$ imply $ab < 0$.

So, $|ab| = -ab = |a|\,|b|$

**Case III.**

$a \geq 0, b < 0$

This is similar to Case II; There is just an interchange of $a$ and $b$.

**Case IV.**

$a < 0, b < 0$

Here, $|a| = -a$, $|b| = -b$ and $ab > 0 \Rightarrow |ab| = ab$

So, $|ab| = (-a)(-b) = |a|\,|b|$

(3) For all $a, b$ in $\mathbf{R}$, $|a + b| \leq |a| + |b|$

**Proof:** For all $a, b \in \mathbf{R}$, $(|a| + |b|)^2 = |a|^2 + 2\,|a|\,|b| + |b|^2$

$$= a^2 + 2\,|a|\,|b| + b^2.$$

Since, $|a|^2 = a^2$ and $|b|^2 = b^2$

$= a^2 + 2\ |a\,b| + b^2$

$\geq a^2 + 2\,ab + b^2$ by Property (1)

Thus, $(|a| + |b|)^2 \geq (a+b)^2 = |a+b|^2$

$\Rightarrow |a| + |b| \geq |a+b|$

(4) For $a, k \in \mathbf{R}$, $|a| < k \Leftrightarrow -k < a < k$

**Proof:** Suppose $|a| < k$. In case $a \geq 0$, you get $a < k$.

In case, $a < 0$, $|a| = -a \Rightarrow -a < k \Rightarrow a > -k$

$\Rightarrow -k < a$

Hence, $|a| < k$ implies that $-k < a < k$.

Conversely, let $-k < a < k$

If $a \geq 0$ you get $|a| = a < k$

If $a < 0$, then $|a| = -a$.

But, $-k < a \Rightarrow k > -a$

or, $-a < k$.

So, $|a| < k$

### 5.2.6 Order and Inequalities

In mathematics, an *inequality* is a statement that defines the relative size or order of two objects to check whether they are similar or not. The following statements explain the order and inequality relationship between integers:

- The notation $a < b$ means that $a$ is *less than b*.
- The notation $a > b$ means that $a$ is *greater than b*.
- The notation $a \neq b$ means that $a$ is *not equal to b*. It does not mean that $a$ is bigger than $b$ or even that they can be compared in size.

In all these cases, $a$ is not equal to $b$ defines that there is inequality in their relation. The following relations are known as *strict inequality*:

- The notation $a \leq b$ means that $a$ is *less than* or *equal to b* or in other words $a$ is *not greater than b*.
- The notation $a \geq b$ means that $a$ is *greater than* or *equal to b* or in other words $a$ is *not smaller than b*.

When the inequality is same for all the values of the given variables for which it is defined, then the inequality is termed as an 'absolute' or 'unconditional' inequality. Similarly, if an inequality holds only for certain values of the variables involved, but is reversed or destroyed for other values of the variables then it is termed as 'conditional inequality'.

Inequalities are governed by the following properties:

**Trichotomy**: The trichotomy property states that for any real numbers $a$ and $b$ any one of the following is true:

- $a < b$
- $a = b$
- $a > b$

**Transitivity**: The transitivity of inequalities states that for any real numbers $a$, $b$, $c$, any one of the following case may be true:

- If $a > b$ and $b > c$; then $a > c$
- If $a < b$ and $b < c$; then $a < c$
- If $a > b$ and $b = c$; then $a > c$
- If $a < b$ and $b = c$; then $a < c$

**Addition and subtraction:** The properties which deal with addition and subtraction, state that for any real numbers $a$, $b$, $c$, any one of the following case may be true:

- If $a < b$, then $a + c < b + c$ and $a - c < b - c$
- If $a > b$, then $a + c > b + c$ and $a - c > b - c$

This states that the real numbers are an ordered group.

**Multiplication and division**: The properties which deal with multiplication and division, state that for any real numbers $a$, $b$, $c$ any one of the following case may be true:

- If $c$ is positive and $a < b$, then $ac < bc$
- If $c$ is negative and $a < b$, then $ac > bc$

This is basically applied to an ordered field.

The next step is to extend **N** so as to include the solution of the equations of the type $a + x = b$ with $a, b \in \mathbf{N}$. The extended system **Z**, consists of, ..., – 4, – 3, – 2, – 1, 0, 1, 2, 3, ..., etc. Each member of **Z** is called an *integer*. To construct an integer with the help of natural numbers, remember a simple fact that every integer can be written as $m - n$ for some $m, n \in \mathbf{N}$, where '–' stands for the usual difference sign. But, as you have not defined this operation for **N**, hence tactfully avoid it and proceed as follows:

Consider $\mathbf{N} \times \mathbf{N} = \{(m, n) \mid m, n \in \mathbf{N}\}$, i.e., the elements of $\mathbf{N} \times \mathbf{N}$ are ordered pairs of natural numbers.

Define a relation ~ on N as under,

$$(m, n) \sim (p, q) \text{ if and only if } m + q = n + p$$

For example, $(1, 5) \sim (3, 7)$ as $1 + 7 = 5 + 3 = 8$ and $(4, 4) \sim (2, 2)$ as $4 + 2 = 2 + 4 = 6$

It can be verified that '~' is an equivalence relation on **N**. By $[m, n]$ it is meant that all ordered pairs $(p, q)$ such that $(m, n) \sim (p, q)$. This $[m, n]$ will be called an integer. The set of all $[m, n]$, $m, n \in \mathbf{N}$ is denoted by **Z**.

Addition and multiplication on **Z** are defined as follows:

$$[m, n] + [m_1, n_1] = [m + m_1, n + n_1]$$
$$[m, n] \,.\, [m_1, n_1] = [mm + nn_1, mn_1 + m_1n]$$

It can be verified that these operations are well defined, i.e., if

$$(m, n) \sim (p, q) \text{ and } (m_1, n_1) \sim (p_1, q_1)$$

Then, $[m, n] + [m_1, n_1] = [p, q] + [p_1, q_1]$

and $[m, n]\, [m_1, n_1] = [p, q] \,.\, [p_1, q_1]$

Addition and multiplication are motivated by the following observations:

Let $z$ and $z_1$ be two integers and $z = m - n$, $z_1 = p - q$ when $m, n, p, q \in \mathbf{N}$.

Then, $z + z_1 = (m + p) - (n + q)$ and $zz_1 = mp + nq - (mq + np)$

For numerical instances of addition and multiplication, consider integers [3, 5] and [4, 1]. $[3, 5] + [4, 1] = [3 + 4, 5 + 1] = [7, 6]$

$$\therefore \quad [3, 5].[4, 1] = [3 \,.\, 4 + 5 \,.\, 1, 3 \,.\, 1 + 5 \,.\, 4]$$
$$= [17, 23]$$

Compare this with the facts that [3, 5] is actually $-2$ and [4, 1] is 3 then,

$-2 + 3 = 1$

Which is same as $7 - 6$, whereas $(-2)(3) = -6 = 17 - 23$

The following properties hold for integers:

(*i*) Addition is associative and commutative.

(*ii*) Multiplication is associative and commutative.

(*iii*) Multiplication is distributive over addition.

(*iv*) The integer $[n, n]$ is called zero element and is denoted by 0.

It can be easily shown that $0 + z = z = z + 0$ for all $z \in \mathbf{Z}$.

(*v*) For each integer $z = [m, n]$, the integer $z¢ = [n, m]$ is called *negative* of $z$. It can be that $z + z¢ = z¢ + z = 0$

***Note:*** By $z - z'$ we mean $z + (-z')$ for any $z, z' \in \mathbf{Z}$.

The integer $z = [m, n]$ is called *positive* if $m > n$ and is called *negative* in case $n > m$.

Let, $z, z' \in \mathbf{Z}$

You define $z > z'$ if $z = z' + u$ for some positive integer $u$.

(*vi*) Given two integers $z$ and $z'$ for which one and only one of the following holds: Either $z > z'$ or $z = z'$ or $z' > z$.

**Example 5.3:** Show that addition on $Z$ is commutative.

**Solution:** Let, $z = [m, n]$ and $z' = [p, q]$, where $m, n, p, q \in \mathbf{N}$

Now $z + z' = [m + p, n + q] = [p + m, q + n]$

$= [p, q] + [m, n] = z' + z$

**Example 5.4:** Prove that multiplication on $Z$ is associative.

**Solution:** Let, $z = [m, n], z' = [p, q]$ and $z'' = [r, s]$ where $m, n, p, q, r, s \in \mathbf{N}$.

$(zz')\, z' = [mp + nq, mq + np]\, [r, s]$

$= [(mp + nq)\, r + (mq + np)\, s, (mp + nq)\, s + (mq + np)\, r]$

$= [mpr + nqr + mqs + nps, mps + nqs + mqr + npr]$ ... (1)

Further $z\, (z'z'') = [m, n]\, [pr + qs, pr + qr]$

$= [m\, (pr + qs) + n\, (ps + qs), m\, (ps + qr) + n\, (pr + qs)]$

$= [(mpn + mqs + nps + nqr, mps + mqr + npr + nqs]$ ...(2)

Using properties of $\mathbf{N}$, one can easily show that the right sides of Equations (1) and (2) are same as a consequence.

$$(zz')\, z'' = z\, (z'z'')$$

**Example 5.5:** Show that for all $z \in \mathbf{Z}$, $z.\ 0 = 0$

**Solution:** Let $z = [m, n], 0 = [p, p]$

$z \cdot 0 = [mp + np, mp + np] = 0$

**Example 5.6:** Prove that if $z > z'$ then (*i*) For any $z'' \in \mathbf{Z}$, $z + z'' > z' + z''$

(*ii*) For any + ve integer $k$, $zk > z'k$

**Solution:**

(*i*) $z > z' \Rightarrow z = z' + u$, $u$ is a positive integer.

Clearly, $z + z'' = z' + u + z''$

$= z' + z'' + u$

$\Rightarrow$ $z + z'' > z' + z''$

(*ii*) As $zk = (z' + u)\, k = z'k + uk$, it is sufficient to prove that product of two positive integers is a positive integer.

Let, $u = [m, n]$ and $k = [p, q]$

Since $u$ is positive, so $m > n$. In case $n = 1$and $m > 1$, then there exists a natural number $t$ such that $m = t^*$. So $(m, n) = (t^*, 1)$

If $n \neq 1$, $n > 1 \Rightarrow n = 1 + k$, for some $k \in \mathbf{N}$.

Then, $m > n \Rightarrow m = 1 + k + l$ for some $l \in \mathbf{N}$.

Thus, $[m, n] = [1 + k + l, 1 + k] = [1 + l, 1]$

Since, $(1 + k + l) + 1 = (1 + k) + (1 + l) \Rightarrow (1 + k + l, 1 + k) \sim (1 + l, 1)$

This implies that $[m, n] = [l^*, 1]$

So, in each case if $z = [m, n]$, then it is positive.

You can write $u = [t^*, 1]$ for some $t \in \mathbf{N}$.

Similarly,

$k = [s^*, 1]$ for some $s \in \mathbf{N}$

Now, $uk = [t^*s^* + 1. \ t^* + s^*]$

But, $t^*s^* + 1 = (t + 1)(s + 1) + 1 = (t + 1) + (s + 1) + ts$

$= (t^* + s^*) + ts$ and $ts \in \mathbf{N}$

So, $t^*s^* + 1 > (t^* + s^*) + ts$. In other words, $uk$ is + ve. Hence the assertion is follow.

***Notes:***

1. It can be shown that $z \in \mathbf{Z}$ is positive if and only if $z > 0$ and negative if and only if $0 > z$.
2. Let $z$ and $z' \in \mathbf{Z}$, then $z > z'$ if and only if $z - z' > 0$.
3. By $z \geq z'$ it is shown that either $z > z'$ or $z = z'$.
4. $\mathbf{Z}^+$ denotes the set of all positive integers, $\mathbf{Z}^-$ denotes the set of all negative, integers. Law of Trichotomy states that $\mathbf{Z} = Z^+ \cup \mathbf{Z}^- \cup \{0\}$ which is a disjoint union.
5. As a convention, 0 is regarded both positive and negative. When we wish to stress upon a non-zero positive integer, then it is called *strictly positive.* Similarly, the *strictly negative* integer is defined.

**CHECK YOUR PROGRESS**

1. State the properties of prime numbers.
2. Define division algorithm.
3. What are even numbers?
4. Define an inequality.

## 5.3 GREATEST COMMON DIVISOR

The greatest common divisor (g.c.d) is also termed as the greatest common factor (g.c.f.) or highest common factor (h.c.f.). The g.c.d. of two or more non-zero integers is the largest positive integer that divides a number without leaving a remainder.

A special case in Euclid's algorithm arises when the remainder is zero.

*Definition.* An integer $d > 0$ is called g.c.d. of two integers $a$, $b$ (non-zero) if,

(*i*) $d \mid a, \ d \mid b$

(*ii*) If $c \mid a, \ c \mid b$ then $d \mid c$

It is written $d =$ g.c.d.$(a, \ b)$ or simply $d = (a, \ b)$.

***Notes:***

1. $(a,\ 0) = |a|,\ (0,\ b) = |b|$
   Clearly, $|a|\ |\ a,\ |a|\ |\ 0$
   If $c\,|\,a$, then $c\,|\,|a| \Rightarrow (a, 0) = |a|$
   Similarly $(0,\ b) = |b|$
2. If $a\,|\,b$, then $(a,\ b) = |a|$
   $|a|\ |\ a$, and $a\,|\,b \Rightarrow |a|\ |\ b$
   If $c\,|\,a,\ c\,|\,b$, then $c\,|\,|a|$
   $\therefore (a,\ b) = |a|$
3. The g.c.d. of $a$ and $b$ does not depend on signs of $a$ and $b$.
   i.e., $(a,\ b) = (-a,\ b) = (a, -b) = (-a, -b)$
   Let, $d = (a,\ b)$.
   Then, $d\,|\,a,\ d\,|\,b \Rightarrow d\,|-a,\ d\,|\,b$
   $c\,|-a,\ c\,|\,b \Rightarrow c\,|\,a,\ c\,|\,b \Rightarrow c\,|\,d$
   $\therefore d = (-a,\ b)$.

The following theorem will prove the existence and uniqueness of g.c.d. of integers $a$ and $b$.

**Theorem 5.3** *Let a, b be two integers. Suppose either* $a \neq 0$ *or* $b \neq 0$. *Then for some* $(\exists)$*, the greatest common divisor d of a, b such that,* $d = ax + by$ *for some integers x, y d is uniquely determined by a and b.*

**Proof:** Let $S = \{au + bv\ |\ u, v$ are integers and $au + bv > 0\}$.

If $a > 0$, then $a = a.1 + b.0 > 0 \Rightarrow a \in S$.

If $a < 0$, then $-a = a(-1) + b.0 > 0 \Rightarrow -a \in S$.

Similarly, if $b > 0$ then $b \in S$ and if $b < 0$ then $-b \in S$. Since one of $a$ and $b$ is non-zero, either $\pm a \in S$ or $\pm b \in S$. In any case $S \neq \varphi$.

By well ordering principle, $S$ has a least element, say $d$.

Now $d \in S \Rightarrow d = ax + by$ for some integers $x$ and $y$. Also, $d > 0$.

Let, $a = dq + r,\ 0 \leq r < d$

Let, $r \neq 0$. Since $r = a - dq$

$$= a - (ax + by)q$$

$$= a(1 - xq) + b(-yq) > 0$$

$$\Rightarrow r \in S$$

But $r < d$, which contradicts the fact that $d$ is the least element of $S$. So, $r = 0$.

Therefore, $a = dq \Rightarrow d\,|\,a$

Similarly, $d\,|\,b$

Suppose, $c\,|\,a,\ c\,|\,b \Rightarrow c\,|\,ax + by = d$

So, $d$ is the greatest common divisor of $a$ and $b$.

If $d'$ is also the greatest common divisor of $a$ and $b$, then $d' \mid a, d' \mid b, \Rightarrow d \mid d'$. Similarly, $d \mid a, d \mid b \Rightarrow d' \mid d$. Since $d, d' > 0, d = d'$. So, $d$ is uniquely determined by $a$ and $b$.

***Note:*** $x$ and $y$ in the preceding theorem need not be unique.

For, $\quad d = ax + by$

$\Rightarrow d = a(x - b) + b(a + y)$

If $x - b = x$, $a + y = y \Rightarrow b = 0 = a$, which is not true. So either, $x - b \neq x$ or $a + y \neq y$.

**Definition:** If g.c.d.$(a, b) = 1$, then $a$ and $b$ are said to be *relatively prime* or *coprime.*

**Corollary 1.** Two integers $a$, $b$ are relatively prime, if and only if $\exists$ integers $x$, $y$ are such that $ax + by = 1$.

**Proof:** Suppose, $a$, $b$ are relatively prime. Then g.c.d.$(a, b) = 1$. By the preceding theorem, $\exists$ integers $x, y$ such that $ax + by = 1$.

*Conversely*, let $ax + by = 1$ for some integers $x, y$.

Let, $d =$ g.c.d.$(a, b)$.

Then, $d \mid a, d \mid b \Rightarrow d \mid ax, d \mid by \Rightarrow d \mid ax + by = 1 \Rightarrow d = 1$.

So, $a, b$ are relatively prime.

**Corollary 2.** If g.c.d.$(a, b) = d$, then g.c.d.$\left(\frac{a}{d}, \frac{b}{d}\right) = 1$.

**Proof:** Given that g.c.d.$(a, b) = d$

$\Rightarrow \exists$ integers $x, y$ such that,

$d = ax + by$

$\Rightarrow 1 = \frac{a}{d}x + \frac{b}{d}y$

$\Rightarrow$ g.c.d. $\left(\frac{a}{d}, \frac{b}{d}\right) = 1$, by Corollary 1.

**Corollary 3.** If $a \mid bc$, with g.c.d.$(a, b) = 1$, then $a \mid c$

**Proof:** As g.c.d.$(a, b) = 1 \Rightarrow \exists$ integers $x, y$ such that,

$ax + by = 1 \quad \Rightarrow acx + bcy = c$

Now, $a \mid ac, a \mid bc \Rightarrow a \mid acx, a \mid bcy$

$\Rightarrow a \mid acx + bcy = c$

**Corollary 4.** If g.c.d.$(a, b) = 1$ and g.c.d.$(a, c) = 1$, then g.c.d.$(a, bc) = 1$.

**Proof:** Since g.c.d.$(a, b) = 1$, $\exists$ integers $x$, $y$, such that $ax + by = 1$. Also, g.c.d.$(a, c) = 1$, $\exists$ integers $u$, $v$, such that $au + cv = 1$.

$\therefore \quad 1 = (ax + by)(au + cv)$

$\quad = a(axu + cxv + byu) + bc(yv)$

By Corollary 1, g.c.d.$(a, bc) = 1$.

The following lemman defines the practical method of finding the greatest common divisor of two integers. First, prove the following result:

**Lemma.** If $a = bq + r$, then g.c.d.$(a, b)$ = g.c.d.$(b, r)$.

**Proof:** Let, g.c.d.$(a, b) = d$.

Then, $d \mid a, d \mid b \Rightarrow d \mid a, d \mid bq \Rightarrow d \mid a - bq = r$.

Suppose $c \mid b, c \mid r$.

Then, $c \mid bq, \; c \mid r \Rightarrow c \mid bq + r \Rightarrow c \mid a, \; c \mid b \Rightarrow c \mid d$.

Thus, $d$ = g.c.d. $(b, r)$.

Let $a, b$ be two integers.

Since, g.c.d.$(a, b)$ = g.c.d.$(|a|, |b|)$, let $a \geq b > 0$.

Let, $a = bq_1 + r_1$, $0 \leq r_1 < b$.

If $r_1 = 0$, then $b \mid a$ and g.c.d. $(a, b) = b$.

Let $r_1 \neq 0$. Divide $b$ by $r_1$ to get integers $q_2$ and $r_2$ such that,

$b = r_1 q_2 + r_2$, $0 \leq r_2 < r_1$

If $r_2 = 0$, then g.c.d.$(b, r_1) = r_1$ and so by above lemma, g.c.d.$(a, b) = r_1$.

If $r_2 \neq 0$, then proceed as above till we get remainder as zero,

Given that,

$$\begin{aligned} a &= q_1 b + r_1, & 0 < r_1 < b \\ b &= q_2 r_1 + r_2, & 0 < r_2 < r_1 \\ r_1 &= q_3 r_2 + r_3, & 0 < r_3 < r_2 \\ &\cdots\cdots\cdots\cdots & \\ r_{n-2} &= q_n r_{n-1} + r_n, & 0 < r_n < r_{n-1} \\ r_{n-1} &= q_{n+1} r_n + 0 & \end{aligned}$$

By above lemma,

g.c.d.$(a, b)$ = g.c.d.$(b, r_1)$ = ...... = g.c.d.$(r_n, 0) = r_n$

So, the last remainder $r_n$ is g.c.d. of $a$ and $b$.

For example, to determine g.c.d. (56, 72), divide 72 by 56 to get,

$72 = 56 + 16$

$56 = 16 \times 3 + 8$

$16 = 8 \times 2 + 0$

Since the last non-zero remainder is 8 hence g.c.d.(56, 72) = 8.

Also, $8 = 56 - 16 \times 3$

$= 56 - (72 - 56) \times 3$

$= 56 \times (4) + 72\,(-3)$

$= 56x + 72y$ where $x = 4$, $y = -3$.

Which shows us the way to find $x, y$ such that,

g.c.d. $(a, b) = ax + by$

**Theorem 5.4:** Let $k > 0$. Then g.c.d.$(ka, kb) = k$ g.c.d.$(a, b)$.

**Proof:** Let, g.c.d.$(a, b) = d$

Then, $d \mid a,\ d \mid b \Rightarrow kd \mid ka,\ kd \mid kb$

Also, $\exists$ integers $x, y$ such that,

$$d = ax + by$$

$$\Rightarrow kd = kax + kby$$

Let, $c \mid ka,\ c \mid kb$ then, $c \mid kax,\ c \mid kby$

$\Rightarrow c \mid kax + kby = kd$

$\Rightarrow$ g.c.d. $(ka, kb) = kd = k\, g.c.d.\, (a, b)$

***Note:*** As $k > 0$, $d > 0$ we get $\Rightarrow kd > 0$

**Corollary:** For any integer $k \neq 0$, g.c.d. $(ka, kb) = |k|$ g.c.d.$(a, b)$.

**Proof:** For $k > 0$, the result follows from the preceding theorem.

Let, $k < 0$. Then, g.c.d.$(ka, kb)$

= g.c.d.$(-ka, -kb)$

$= -k$ g.c.d.$(a, b)$ by above theorem.

$= |k|$ g.c.d.$(a, b)$

**Definition:** The *least common multiple* (*l.c.m.*) of two non-zero integers $a$ and $b$, denoted as l.c.m.$(a, b)$ is the positive integer $m$ such that,

(*i*) $a \mid m,\ b \mid m$

(*ii*) If $a \mid c,\ b \mid c$, with $c > 0$, then $m \mid c$

**Theorem 5.5:** For positive integers $a$ and $b$,

g.c.d.$(a, b) \times$ l.c.m.$(a, b) = ab$

**Proof:** Let, $d =$ g.c.d.$(a, b)$

Now, $\dfrac{ab}{d} = a.\dfrac{b}{d} \Rightarrow a \left| \dfrac{ab}{d} \right.$ as $\dfrac{b}{d}$ is integer.

Also, $\dfrac{ab}{d} = b.\dfrac{a}{d} \Rightarrow b \left| \dfrac{ab}{d} \right.$ as $\dfrac{a}{d}$ is integer.

Let, $m = \frac{ab}{d}$, then $a \mid m$ and $b \mid m$.

Suppose now $a \mid c$, $b \mid c$. Since $(a, b) = d$, $\exists$ integers $x, y$ such that, $d = ax + by$.

$$\therefore \qquad \frac{c}{m} = \frac{cd}{ab} = \frac{c(ax+by)}{ab} = \left(\frac{c}{b}\right)x + \left(\frac{c}{a}\right)y = \text{Integer}$$

$\Rightarrow m \mid c$

Thus, $m$ = l.c.m. $(a, b)$, i.e., $\frac{ab}{d}$ = l.c.m. $(a, b)$

Or, $ab$ = g.c.d. $(a, b) \times$ l.c.m. $(a, b)$.

**Example 5.7:** Let g.c.d. $(a, b) = 1$.

Show that g.c.d.$(a + b, a^2 - ab + b^2) = 1$ or 3.

**Solution:** Let, g.c.d.$(a + b, a^2 - ab + b^2) = d$

Then, $d \mid a + b$, $d \mid a^2 - ab + b^2$

$\Rightarrow d \mid (a + b)^2 = a^2 + b^2 + 2ab$, $d \mid a^2 - ab + b^2$

$\Rightarrow d \mid 3ab$

Let, g.c.d. $(d, a) = e$

Then, $e \mid d \mid a + b \Rightarrow e \mid a + b$ and $e \mid a$

$\therefore$ $e \mid (a + b) - a = b$

So, $e \mid$ g.c.d.$(a, b) = 1 \Rightarrow e = 1$

$\therefore$ g.c.d.$(d, a) = 1$

Similarly, g.c.d.$(d, b) = 1$

$\therefore$ $d \mid 3 \Rightarrow d = 1$ or 3.

**Example 5.8:** Let g.c.d.$(a, b) = 1$. Show that g.c.d.$(a^n, b^n) = 1$ for every integer $n \geq 1$.

**Solution:** Since, g.c.d. $(a, b) = 1$, $\exists$ integers $x, y,$ such that $ax + by = 1$.

$\Rightarrow (ax + by)(ax + by) = 1$

$\Rightarrow a^2x^2 + 2abxy + by^2 = 1$

$\Rightarrow a^2x^2 + b(2axy + y^2) = 1$

$\Rightarrow$ g.c.d.$(a^2, b) = 1$

In this way you will get,

g.c.d.$(a^n, b) = 1$ or g.c.d.$(b, a^n) = 1$

Proceeding as above, we get

g.c.d.$(b^n, a^n) = 1$

### 5.3.1 Linear Diophantine Equation

*Definition.* Linear Diophantine equation is an equation $ax + by = c$ in two unknowns $x$ and $y$, where $a$, $b$, $c$ are given integers and one of $a$, $b$ is not zero. The name is due to the mathematician Diophantus. A natural question arises as to when such an equation would have a solution? The following theorem is helpful in finding the answer.

**Theorem 5.6:** *The linear Diophantine equation $ax + by = c$ has a solution if and only if $d \mid c$, where $d = \text{g.c.d.}\ (a, b)$. If $x_0$, $y_0$ is a particular solution, then the other solutions are given by,*

$$x = x_0 + \frac{b}{d}t,\ y = y_0 - \frac{a}{d}t \ \ \textit{for varying integer } t.$$

**Proof:** Suppose, $ax + by = c$ has a solution.

Let, $x = x_0$, $y = y_0$ be a solution.

Then, $ax_0 + by_0 = c$.

Let, $d = \text{g.c.d.}(a, b)$.

$$\therefore \quad d \mid a, d \mid b \quad \Rightarrow d \mid ax_0,\ d \mid by_0$$
$$\Rightarrow d \mid ax_0 + by_0 = c$$

Conversely, let $d \mid c$. Let, $c = dk$.

Since, $d = \text{g.c.d.}\ (a, b)$, $\exists$ integers $x_0$, $y_0$ such that

$$ax_0 + by_0 = d \Rightarrow a(x_0k) + b(y_0k) = dk = c$$
$$\Rightarrow ax + by = c \text{ has a solution } x = x_0k,\ y = y_0k$$

To prove the second assertion, let $x_0$, $y_0$ be a given solution of,

$ax + by = c$.

Let $x'$, $y'$ be any solution of $ax + by = c$

$$\therefore \quad ax_0 + by_0 = ax' + by' = c$$
$$\Rightarrow a(x_0 - x') = b(y' - y_0)$$
$$\Rightarrow \frac{a}{d}(x_o - x') = \frac{b}{d}(y' - y_0), \text{ where } d = \text{g.c.d.}\ (a, b)$$
$$\Rightarrow \frac{b}{d} \Big| \frac{a}{d}(x_0 - x')$$

Since, $\text{g.c.d.}\left(\frac{a}{d}, \frac{b}{d}\right) = 1.$

$$\frac{b}{d} | x_0 - x' \Rightarrow \frac{b}{d} | x' - x_0$$
$$\Rightarrow x' - x_0 = \frac{b}{d}t, \text{ where } t \text{ is an integer.}$$

NOTES

$$\Rightarrow \; x' = x_0 + \frac{b}{d}t \;\Rightarrow\; \frac{a}{d}\frac{b}{d}t = \frac{b}{d}(y_0 - y')$$

$$\Rightarrow \; \frac{a}{d}t = y_0 - y' \;\Rightarrow\; y' = y_0 - \frac{a}{d}t$$

It can be easily seen that for all values of $t$, $x' = x_0 + \frac{b}{d}t, y' = y_0 - \frac{a}{d}t$ is a solution of $ax + by = c$ as,

$$a\left(x_0 + \frac{b}{d}t\right) + b\left(y_0 - \frac{a}{d}t\right)$$

$$= ax_0 + by_0 = c$$

**Example 5.9:** Determine all the solutions in the integers of the Diophantine equation $56x + 72y = 40$.

**Solution:** First find g.c.d.(56, 72).

Now, $72 = 56 \times 1 + 16$

$56 = 3 \times 16 + 8$

$16 = 2 \times 8$

Hence, g.c.d.(56, 72) = 8

$8 = 56 - 3 \times 16$

$= 56 - 3 \times (72 - 56)$

$= 4 \times 56 - 3 \times 72$

$\Rightarrow 40 = 56 \times 20 + 72 \times (-15)$

$\Rightarrow x = 20,\ y = -15$, is a solution of $56x + 72y = 40$

By the preceding theorem, any other solution is given by

$$\left(20 + \frac{72}{8}t, -15 - \frac{56}{8}t\right)$$

$= (20 + 9t,\ -15 - 7t)$ for any integer $t$.

**CHECK YOUR PROGRESS**

5. What is the greatest common divisor (GCD)?
6. State the linear Diophantine equation.

## 5.4 EUCLIDEAN ALGORITHM

In number theory, the Euclidean algorithm also termed as Euclid's algorithm, is a very important algorithm that is used to determine the greatest common divisor of two elements of any Euclidean domain. It does not need factoring the two integers and is one of the oldest algorithms known, dating back to the ancient Greeks.

These algorithms are used when division with remainder is possible. It also includes rings of polynomials over a field and the ring of Gaussian integers for all Euclidean domains.

NOTES

**Theorem 5.7:** *Euclid's algorithm*

*Let $k > 0$ be an integer and $j$ be any integer. Then $\exists$ unique integers $q$ and $r$ such that $j = kq + r$, where $0 \le r < k$.*

**Proof:** Let $S = \{j - kq \mid q$ is an integer, $j - kq \ge 0\}$.

Then, $S \ne \varphi$, if you take $q = -\,|j|$.

When, $j > 0$, then $j - kq = j + kj > 0 \Rightarrow j - kq \in S$

and if $j < 0$, then $j - kq = j - kj$

$= j\,(1 - k) \ge 0$

$\Rightarrow j - kq \in S$

If $j = 0$, then $j - kq = j - k.0$

$= j = 0$

$\Rightarrow j - kq \in S$

In any case, $S \ne \varphi$.

By well ordering principle, $S$ has least element, say $r \in S$.

$r \in S \Rightarrow r = j - kq$ for some integer $q$.

$\Rightarrow j = kq + r$.

Also, $r \ge 0$

Suppose, $r \ge k$

Then, $j - kq \ge k$

$\Rightarrow j - k\,(q + 1) \ge 0$

$\Rightarrow j - k\,(q + 1) \in S$

But, $j - k\,(q + 1) < j - kq$ as $k > 0$, contradicts $r = j - kq$ which is least the element of $S$.

$\therefore\ 0 \le r < k$.

***Uniqueness***: Suppose, $j = kq + r = kq' + r'$, $0 \le r, r' < k$. Then, $k(q-q') = r' - r$. Suppose, $r' > r$. Then, $r' - r > 0$. But $k \mid r' - r \Rightarrow k \le r' - r$. Since, $r, r' < k$, $r' - r < k$, a contradiction.

$\therefore\ r' \not> r$. Similarly $r \not> r'$ $\quad \therefore\ r = r' \Rightarrow kq = kq' \Rightarrow q = q'$.

An important application of this result is the basis representation theorem.

**Theorem 5.8:** *Basis Representation Theorem*

*Let $b > 0$ be an integer and let $N > 1$ be also any integer. Then $N$ can be expressed as,*

$$N = a_m b^m + a_{m-1} b^{m-1} + \ldots + a_1 b + a_0,$$

*Where $m$ and $a_i$s are integers such that $m > 0$ and $0 \le a_i < b$. Also then, these $a_i$s are uniquely determined. Here, $b$ is called the base of representation of $N$.*

**Proof:** If $N < b$

Then, $N = 0b^m + 0b^{m-1} + ... + 0b + N$ is the representation of $N$ as required.

Let $N \geq b > 0$.

By Euclid's algorithm, $\exists$ integers $q, r$ such that,

$$N = bq + r, \quad 0 \leq r < b \leq N$$

Since, $N - r > 0$, $bq > 0 \Rightarrow q > 0$ as $b > 0$.

If $q < b$, then $N = bq + r$ is the required representation of $n$.

If $q \geq b > 0$, then by Euclid's algorithm $\exists$ integers $q_1$, $r_1$ such that,

$$q = bq_1 + r_1, \quad 0 \leq r_1 < b \leq q$$

Since, $q - r_1 > 0$, $bq_1 > 0 \Rightarrow q_1 > 0$ as $b > 0$

Now, $N = bq + r = b(bq_1 + r_1) + r$

$\Rightarrow N = b^2q_1 + br_1 + r$

If $q_1 < b$, then it is the required representation of $N$. In this way after a finite number of steps, you get,

$$N = a_m b^m + a_{m-1}b^{m-1} + ... + a_1b + a_0$$

Where $a_i's$ are integers such that,

$$0 \leq a_i < b \quad \text{for all } i = 1,..., m$$

Uniqueness of $a_is$ follows as:

Suppose $N = c_m b^m + c_{m-1} b^{m-1} + ... + c_1 b + c_0$ where each $c_i$ is an integer such that $0 \leq c_i < b$. You can select the same $m$ in both the representations of $N$ because if one representation of $N$ has lesser terms then you can always insert zero coefficients and make the number of terms to be same.

$\therefore \quad 0 = (a_m - c_m)b^m + ... + (a_1 - c_1)b + (a_0 - c_0)$

Let, $a_i - c_i = d_i$

Then, $d_mb^m + ... + d_1b + d_0 = 0$

Show that $d_i = 0$ for all $i$.

Suppose for some $i$, $d_i \neq 0$. Let $k$ be the least subscript such that $d_k \neq 0$

Then, $d_kb^k + d_{k+1}b^{k+1} + ... + d_mb^m = 0$

$\Rightarrow d_kb^k = -(d_{k+1}b^{k+1} + ... + d_mb^m)$

$\Rightarrow d_k = -(d_{k+1}b + d_{k+2}b^2 + ... + d_mb^{m-k})$

$\Rightarrow d_k = -b(d_{k+1} + d_{k+2}b ... + d_mb^{m-k-1})$

$\Rightarrow b \mid d_k$

$\Rightarrow b \mid |d_k|$

$\Rightarrow b \leq |d_k|$

But, $a_k, c_k < b \Rightarrow |a_k - c_k| < b$

$\Rightarrow |d_k| < b$

So, The contradiction is as follows:

$\therefore \quad d_i = 0$ for all $i = 1,..., m$

$\therefore \quad a_i = c_i$ for all $i = 1,..., m$

When the integer $N$ is expressed as follows:

$N = a_m b^m + ... + a_1 b + a_0, \quad 0 \le a_i < b,$

It is represented as, $\quad N = (a_m\, a_{m-1} \,...\, a_1 a_0)b$

And say that $N$ is $a_m a_{m-1} \,...\, a_0$ to the base $b$.

For example,

$132 = 1.10^2 + 3.10 + 2$. Here base is 10.

Then, according the preceding statement

$132 = (132)_{10}$

So, numbers that you usually write are to the base 10.

Again, if you want to write 132 to the base 2, then first you write,

$132 = 2^7 + 2^2 = 2^7 + 0.2^6 + 0.2^5 + 0.2^4 + 0.2^3 + 1.2^2 + 0.2 + 0$

By basis representation theorem,

$132 = (10000100)_2$

**Example 5.10:** If $a, b$ are integers with $b \neq 0,$ show that there exist unique integers $q$ and $r$ satisfying $a = bq + r,$ where $-\frac{1}{2}|b| < r \le \frac{1}{2}|b|$.

**Solution:** By Euclid's algorithm, there exist unique integers $q'$, $r'$ such that,

$a = q'|b| + r', \quad$ where $0 \le r' < |b|$

(As $|b| > 0$ when $b \neq 0$).

**Case 1**. $\quad 0 \le r' \le \frac{1}{2}|b|$

Take, $\quad r' = r$, $q' = q$ (if $b > 0$),

$q' = -q$ (if $b < 0$)

Since, $\quad -\frac{1}{2}|b| < 0 \le r' \frac{1}{2}|b|,$

$-\frac{1}{2}|b| < r \le \frac{1}{2}|b|$

Also, $\quad a = q'\ |b| + r'$ becomes,

$a = qb + r \qquad$ if $b > 0$

And, $\quad a = (-q)(-b) + r \quad$ if $b < 0$

$= qb + r$

where, $\quad \frac{1}{2}|b| < r \le \frac{1}{2}|b|$

NOTES

**Case 2.** $\frac{1}{2}|b| < r < |b|$

Take, $r' = r + |b|$

$q' = q - 1 \quad \text{if } b > 0$

$= -q - 1 \text{ if } b < 0$

Now, $\frac{1}{2}|b| < r' = r + |b|$

$\Rightarrow -\frac{1}{2}|b| < r$

$\therefore \quad -\frac{1}{2}|b| < r < \frac{1}{2}|b|$

Again, $a = |b|\ q + r'$ becomes,

$a = b(q - 1) + r + \text{b}, \qquad \text{when } b > 0$

$= bq + r$

Also, when $b > 0$, $a = |b|\ q + r'$ becomes,

$a = -b\,(-q - 1) + r - b$

$= bq + r$

Where, $-\frac{1}{2}|b| < r < \frac{1}{2}|b|$

## 5.5 FIBONACCI NUMBERS

In mathematics, the following number sequence is termed as Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …………………

The first two Fibonacci numbers are 0 and 1, and all other numbers are the sum of the previous two numbers.

The Fibonacci number sequence is named after Leonardo of Pisa, who was famous as Fibonacci.

The Fibonacci sequence has its origin in ancient India and was used in the metrical sciences (prosody). The inspiration came from Sanskrit prosody in which the length of long syllables was 2 and the short syllables, 1. Any configuration of length $n$ was created by simply adding a short syllable to a design of length $n - 1$ or a long syllable to a design of length $n - 2$. Hence, the number of designs of length $n$ was the sum of the two previous numbers in the sequence. The first 21 Fibonacci numbers denoted by $F_n$ are represented in Table 5.1, where $n = 0, 1, 2, \ldots\ldots, 20$:

***Table 5.1*** *First 21 Fibonacci Numbers*

| | |
|---|---|
| $F_0$ | 0 |
| $F_1$ | 1 |
| $F_2$ | 1 |
| $F_3$ | 2 |
| $F_4$ | 3 |
| $F_5$ | 5 |
| $F_6$ | 8 |
| $F_7$ | 13 |
| $F_8$ | 21 |
| $F_9$ | 34 |
| $F_{10}$ | 55 |
| $F_{11}$ | 89 |
| $F_{12}$ | 144 |
| $F_{13}$ | 233 |
| $F_{14}$ | 377 |
| $F_{15}$ | 610 |
| $F_{16}$ | 987 |
| $F_{17}$ | 1597 |
| $F_{18}$ | 2584 |
| $F_{19}$ | 4181 |
| $F_{20}$ | 6765 |

NOTES

The numbers sequence $F_n$ in Fibonacci can also be explained by using the recurrence relation as, $F_n = F_{n-1} + F_{n-2}$ with seed values $F_0 = 0$ and $F_1 = 1$.

The Fibonacci recursion expressed as $F(n+2) - F(n+1) - F(n) = 0$ is akin to the significant equation of the golden ratio form $x^2 - x - 1 = 0$.

It is also termed as the generating polynomial of the recursion. The Fibonacci numbers also form a Lucas sequence $U_n$ (1, –1) and are termed as Lucas companions that satisfy the similar recurrence equation.

**CHECK YOUR PROGRESS**

7. What is Euclid's algorithm?
8. How are the Fibonacci numbers calculated?

## 5.6 CONGRUENCES AND EQUIVALENCE RELATIONS

### 5.6.1 Congruences Relations

Let $a$, $b$, $c$ $(c > 0)$ be integers. It is said that $a$ is congruent to $b$ modulo $c$ if $c$ divides $a - b$ and is written as $a \equiv b \pmod{c}$.This relation '$\equiv$' on the set of integers is an equivalence relation.

Addition, subtraction and multiplication in congruences behave naturally.

Let, $a \equiv b \pmod{c}$

$a_1 \equiv b_1 \pmod{c} \Rightarrow c \mid a - b,\ c \mid a_1 - b_1$

$\Rightarrow c \mid (a + a_1) - (b + b_1)$

$\Rightarrow a + a_1 \equiv b + b_1 \pmod{c}$

Similarly, $a - a_1 \equiv b - b_1 \pmod{c}$

Also, $c \mid a - b,\ c \mid a_1 - b_1$

$\Rightarrow c \mid aa_1 - ba_1,\ c \mid ba_1 - bb_1$

$\Rightarrow c \mid (aa_1 - ba_1) + (ba_1 - bb_1)$

$\Rightarrow c \mid aa_1 - bb_1$

$\Rightarrow aa_1 \equiv bb_1 \pmod{c}$

However, it is not possible to achieve the above result in case of division.

Indeed $\frac{a}{a_1}$ or $\frac{b}{b_1}$ may not even be integers.

Again, cancellation in congruences in general may not hold.

i.e., $ad \equiv bd \pmod{c}$ need not essentially imply.

$a \equiv b \pmod{c}$

For example, $2.2 \equiv 2.1 \pmod{2}$

But, $2 \not\equiv 1 \pmod{2}$

However, cancellation holds if g.c.d.$(d, c) = 1$.

i.e., if $ad \not\equiv bd \pmod{c}$

And, g.c.d.$(d, c) = 1$

Then, $a \equiv b \pmod{c}$

**Proof:** $ad \equiv bd \pmod{c}$

$\Rightarrow c \mid ad - bd$

$\Rightarrow c \mid d\,(a - b)$

$\Rightarrow c \mid a - b$ as g.c.d.$(c, d) = 1$

$\Rightarrow a \equiv b \pmod{c}$

**Example 5.11:** If $a \equiv b \pmod{n}$, prove that g.c.d.$(a, n) = (b, n)$.

**Solution:** Let, $d =$ g.c.d.$(a, n)$

Then, $d \mid a,\ d \mid n$. But, $n \mid a - b$

$\therefore$ $d \mid a - b,\ d \mid a$

$\Rightarrow d \mid a - (a - b) = b$

$\therefore$ $d \mid b,\ d \mid n$

Let, $c \mid b,\ c \mid n \Rightarrow c \mid b,\ c \mid a - b$ as $n \mid a - b$

$\Rightarrow c \mid a - b + b = a$

$\Rightarrow c \mid a,\ c \mid n$

$\Rightarrow c \mid d$ as $d =$ g.c.d.$(a,\ n)$

$\Rightarrow$ g.c.d.$(b,\ n) = d$

**Example 5.12:** Establish that if $a$ is an odd integer, then

$a^{2^n} \equiv 1 \pmod{2^{n+2}}$ for any $n \geq 1$.

**Solution:** The result can be prove by induction on $n$.

Let, $n = 1$.

Then, $a^{2^n} = a^2$

And $2^{n+2} = 2^3 = 8$

Let, $a = 2k + 1$

Then, $a^2 = 4k^2 + 4k + 1$

$= 4k\,(k + 1) + 1$

$\therefore\ a^2 - 1 = 4k\,(k + 1)$

= Multiple of 8, as either $k$ is even or $k + 1$ is even.

$\therefore\ a^2 \equiv 1 \pmod 8$

So, result is true for $n = 1$.

Assume that the result is true for $n = k$.

Then, $a^{2k} \equiv 1 \pmod{2^{k+2}}$

Now, $a^{2k+1} - 1 = (a^{2k})^2 - 1$

$= (a^{2k} - 1)\,(a^{2k} + 1)$

= (Multiple of $2^{k+2}$) $(a^{2k} + 1)$ by induction hypothesis.

But, $a =$ Odd $\Rightarrow a^{2k} =$ Odd $\Rightarrow a^{2k} + 1 =$ Even

$\therefore\ a^{2k+1} - 1 =$ Multiple of $2^{k+3}$

$\therefore\ a^{2k+1} \equiv 1 \pmod{2^{k+3}}$

So, result is true for $n = k + 1$.

By induction, result is true for all $n \geq 1$.

**Example 5.13:** Show that for any integer $a$,

$a^3 \equiv 0,\ 1,$ or $8 \pmod 9$

**Solution:** Let, $a = 3k + r,\ 0 \leq r < 3$

If, $r = 0$, then $a = 3k$

$\Rightarrow a^3 = 27k^3 \equiv 0 \pmod 9$

If, $r = 1$, then $a = 3k + 1$

$\therefore\ a^3 = 27k^3 + 1 + 9k^2 + 9k$

$\Rightarrow \quad a^3 \equiv 1 \pmod 9$

If , $\quad a = 3k + 2$, then $a^3 = 27k^3 + 8 + 27k^2 + 36k^2$

$\Rightarrow a^3 \equiv 8 \pmod 9$

**NOTES**

$\therefore \quad a^3 \equiv 0, 1$ or $8 \pmod 9$

**Example 5.14:** If $ca \equiv cb \pmod n$, then $a \equiv b \left(\text{mod}\, \frac{n}{d}\right)$, where $d$ = g.c.d. $(c, \; n)$.

**Solution:** Given that, $d$ = g.c.d. $(c, \; n)$

$$\Rightarrow \qquad 1 = \text{g.c.d.}\left(\frac{c}{d}, \frac{n}{d}\right)$$

Also, $\quad ca \equiv cb \pmod n$

$\Rightarrow ca - cb = nk$ for some integer $k$.

$$\Rightarrow \frac{c}{d}a - \frac{c}{d}b = \frac{n}{d}k$$

$$\Rightarrow \frac{n}{d}\left|\frac{c}{d}(a-b)\right.$$

$$\Rightarrow \frac{n}{d}\left| a-b \right. \text{ as g.c.d.}\left(\frac{c}{d}, \frac{n}{d}\right) = 1$$

$$\Rightarrow a \equiv b \left(\text{mod}\, \frac{n}{d}\right)$$

**Example 5.15:** Find the remainder obtained by dividing 1! + 2! + 3! + 4! + ... + 100! by 12.

**Solution:** Each number 4! onwards is a multiple of 12.

$\therefore$ 1! + 2! + 3! + 4! + ... + 100! $\equiv$ 1! + 2! + 3! + 0 + ... + 0 (mod 12)

$\Rightarrow$ 1! + 2! + 3! + 4! + ... + 100! $\equiv$ 9 (mod 12)

$\Rightarrow$ 9 is the required remainder.

**Example 5.16:** Find the remainder when $2^{50}$ is divided by 7.

**Solution:** Now $2^3 \equiv 1 \pmod 7$

$\Rightarrow (2^3)^{16} \equiv 1^{16} \equiv 1 \pmod 7$

$\Rightarrow 2^{48} \equiv 1 \pmod 7$

$\Rightarrow 2^{48} \times 2^2 \equiv 2^2 \pmod 7$

$\Rightarrow 2^{50} \equiv 4 \pmod 7$

$\therefore$ 4 is the remainder.

**Example 5.17:** What is the remainder when the sum $1^5 + 2^5 + 3^5 + ... + 99^5 + 100^5$ is divided by 4?

**Solution:** $1 \equiv 1 \pmod 4 \Rightarrow 1^5 \equiv 1 \pmod 4$

$2^2 \equiv 0 \pmod 4 \Rightarrow 2^5 \equiv 0 \pmod 4$

$3^2 \equiv 1 \pmod 4 \Rightarrow 3^5 \equiv 3 \pmod 4$

$\Rightarrow 3^5 \equiv -1 \pmod 4$

$4^2 \equiv 0 \pmod 4 \Rightarrow 4^5 \equiv 0 \pmod 4$

$\therefore\ 1^5 + 2^5 + 3^5 + 4^5 \equiv 1 + 0 - 1 + 0 \equiv 0 \pmod 4$

Any number after these will be of the form $2k + 1,\ 2k + 2,\ 2k + 3,\ 2k + 4,\ k > 1$.

Now, $(2k + 1)^2 \equiv 1 \pmod 4 \Rightarrow (2k + 1)^5 \equiv 2k + 1 \equiv 1 \pmod 4$

$(2k + 2)^2 \equiv 0 \pmod 4 \Rightarrow (2k + 2)^5 \equiv 0 \pmod 4$

$(2k + 3)^2 \equiv 1 \pmod 4 \Rightarrow (2k + 3)^5 \equiv 2k + 3 \equiv -1 \pmod 4$

$(2k + 4)^2 \equiv 0 \pmod 4 \Rightarrow (2k + 4)^5 \equiv 0 \pmod 4$

$\therefore\quad 1^5 + 2^5 + 3^5 + 4^5 + ...\ 99^5 + 100^5 \equiv 0 \pmod 4$

So, remainder is 0 when the given number is divided by 4.

### 5.6.2 Equivalence Relations

A relation $R$ on a set $A$ is called an equivalence relation if $R$ is reflexive, symmetric and transitive.

For example, let $N$ be set of natural numbers. Define $R$ on $N$ as,

$$R = \{(x,y) : x + y \text{ is even}, x, y \in N\}$$

**Proof:** Let, $x \in N$. Now, $x + x = 2x$.

Clearly $2x$ is even. Therefore, $R$ is reflexive. Let $x, y \in N$ and $x + y$ be even. Clearly $y + x$ is also even and hence, $R$ is symmetric.

Now, if $x + y$ is even and $y + z$ is even then prove that $x + z$ is even.

Since, $x + y$ and $y + z$ are even, both $(x + y)$ and $(y + z)$ are divisible by 2.

$\therefore\ (x + y) + (y + z)$ is also divisible by 2, i.e., $x + (y + y) + z$ is divisible by 2.

$\therefore\ (x + z)$ is divisible by 2.

Hence, $R$ is transitive. So, $R$ is an equivalence relation.

***Note:*** From the relation graph or relation matrix the kind of relation can be identified.

**Example 5.18:** The relation $R$ on a set is represented by,

$$M_R = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Is $R$ reflexive, symmetric or antisymmetric?

**Solution:** In the matrix $M_R$, the diagonal elements are 1. Therefore, $R$ is reflexive. Since, the matrix $M_R$ is symmetric hence, the relation $R$ is also symmetric.

**Example 5.19:** The relation $R$ and $R1$ on a set is represented by,

$$(i)\ M_R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (ii)\ M_{R1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Are the relations $R$ and $R1$ reflexive, symmetric, antisymmetric and/or transitive?

**Solution:**

(*i*) Matrix $M_R$ is symmetric. Its diagonal entries are 1. Hence, relation $R$ is symmetric and reflexive. Since $R$ is not antisymmetric, $R$ is transitive.

(*ii*) The relation $R1$ is not reflexive as all diagonal entries are not 1.

$R1$ is symmetric [$\because$ $M_{R1}$ is symmetric] and $R1$ is transitive.

**Example 5.20:** Draw the relation graph for the following relations.

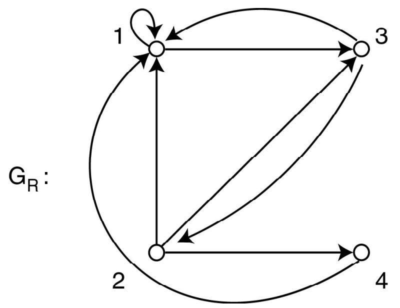
(*i*) $R = \{(1, 1), (1, 3), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (4, 1)\}$ on the set $X = \{1, 2, 3, 4\}$.

(*ii*) $R1 = \{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}$ on the set $Y = \{1, 2, 3\}$.

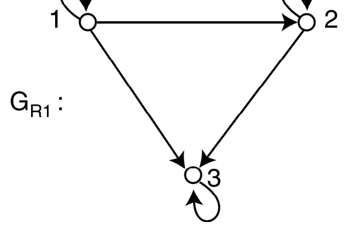
**Solution:**

(*i*) The relation graph $G_R$ of $R$ is drawn as follows.

The vertices of $G_R$ are 1, 2, 3 and 4.



(*ii*) The relation graph $G_{R1}$ of $R1$ is drawn as:



**Example 5.21:** Let $R$ be the relation represented by:

$$M_R = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Find the relation matrices representing (*i*) $R^{-1}$ (*ii*) $R^c$ (*iii*) $R^2$.

**Solution:**

(*i*) To get the inverse relation matrix $(M_{R^{-1}})$ of a relation matrix $(M_R)$ just write the transpose of $M_R$.

$$\therefore \qquad M_{R^{-1}} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

(*ii*) To find the complement relation matrix, replace 0 by 1 and 1 by 0 in the given relation matrix.

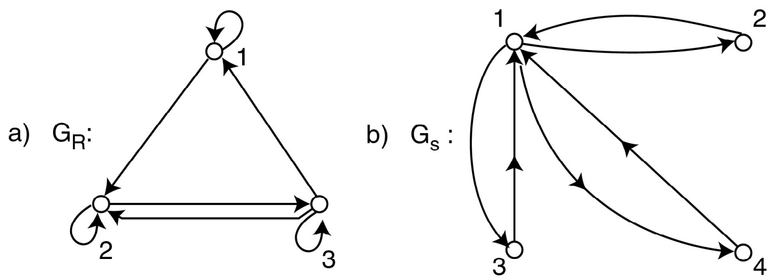
$$\therefore \qquad M_{R^c} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

(*iii*) To find the relation matrix of $R^2$ when $R^2 = R\ o\ R$.

If the relation matrix $M_R$ is known, then $M_{R^2} = M_R \times M_R$, i.e., the matrix multiplication

$$\therefore \qquad M_{R^2} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

**Example 5.22:** Find whether the relations for the directed graphs shown in the following figures are reflexive, symmetric, antisymmetric and/or transitive.



**Solution:**

(*i*) In $G_R$, there are loops at every vertex of the relation graph and hence it is reflexive.

NOTES

It is neither symmetric nor antisymmetric since there is an edge between 1 and 2 but not from 2 to 1. There are edges connecting 2 and 3 in both directions.

Moreover, the relation is not transitive, since there is an edge from 1 to 2 and 2 to 3, but no edge from 1 to 3.

(*ii*) Since loops are not present in $G_S$, this relation is not reflexive. Further it is symmetric and not antisymmetric.

Moreover, the relation is not transitive.

## 5.7 PUBLIC KEY ENCRYPTION SCHEMES

Public key encryption refers to a sort of cipher architecture recognized as public key cryptography that uses two keys or a key pair for encrypting and decrypting data. One of the two keys is a public key, which is used to encrypt a message. When this encrypted message is sent to the recipient, he or she uses his or her private key to decrypt it. This is the basic theory of public key encryption. Cryptography issues are important in many security services, such as peer entity authentication, data origin authentication, data confidentiality, data integrity, message and selected fields as sent by genuine person.

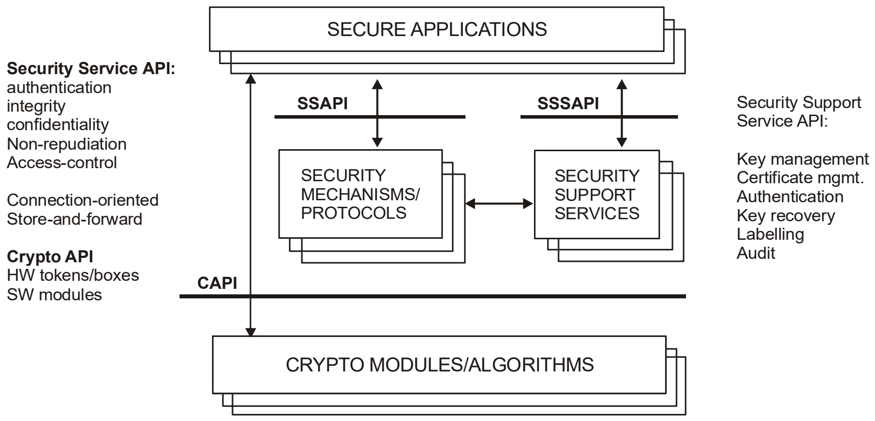


***Figure 5.1*** *Architecture Cryptography Based Security*

In Figure 5.1, the cryptography based security provides a layered architecture in which security service applications program interfaces (APIs), crypto API and security support service API provide secure applications. These services provide cryptographic modules and algorithms. The security service API provides authentication, integrity, confidentiality, non-repudiation, access control, connectionoriented method and 'store and forward' method. The crypto API provides HW tokens and SW modules. HW tokens are hardware cryptographic services, for example, accelerator boards and SW modules are kernel modules providing cryptographic services, such as implementation of cryptographic algorithm. The security support service API provides certificate management authentication, key recovery, labelling and auditing. The prime aim of this hierarchy level is to provide the following mechanisms:

- This level promotes the development of security services and cryptographic API.
- This level also identifies the area of security support service APIs along with certificate management, key management and authentication APIs.
- This level demonstrates the uses of cryptographic APIs products and services.
- This level identifies the steps of demonstrations and experiments in the field of crypto services.

Table 5.2 shows the relationship between security services and security mechanisms:

***Table 5.2*** *Relationship between Security Mechanism and Security Services*

| Service | Encipherment | Digital Signature | Access Control | Data Integrity | Authentication Exchange | Traffic Padding | Routing Control | Notarization |
|---|---|---|---|---|---|---|---|---|
| Peer entity authentication | √ | √ | | | √ | | | |
| Data origin authentication | √ | √ | | | | | | |
| Access control | | | √ | | | | | |
| Data integrity | √ | √ | | √ | | | | |
| Traffic flow confidentiality | √ | | | | | √ | √ | |
| Nonrepudiation | | √ | | √ | | | | √ |
| Availability | | | | √ | √ | | | |

### 5.7.1 Message Authentication Code

Message Authentication Code (MAC) follows the algorithm given as follows:

```
Hash(MAC_write_secret||pad_2||
hash(MAC_write_secret||pad_1||seq_num||SSLCompressed.type|
|SSLCompressed.length||SSLCompressed.fragment))
```

This code is written and computed over compressed data. A shared secret key is used for this algorithm. Table 5.3 shows the keywords used in this code:

**NOTES**

***Table 5.3*** *Keywords Used in Crypto Algorithm and Their Functions*

| **Keywords** | **Function** |
|---|---|
| `||` | Concatenation operator. |
| `MAC_write_secret` | Shared secret key. |
| `hash` | Cryptograph hash algorithm. |
| `pad_1` | By 0×36 (binary value: 0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1. |
| `pad_2` | The byte 0×5C (binary value: 0101 1100) repeated 48 times and for MD5 and 40 times for SHA-1. |
| `seq_num` | It is the sequence number that is used for this message. |
| `SSLCompressed.type` | It is the higher level protocol used to process this fragment. |
| `SSLCompressed.length` | The length of compressed fragment. |
| `SSLCompressed.fragment` | The compressed fragment. |

***Note:*** The `SSLCompressed.fragment` is used if the compression message is not used.

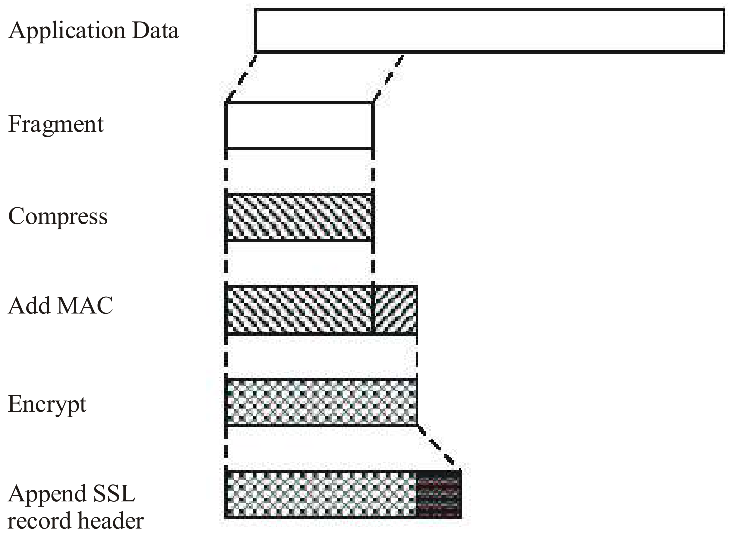


***Figure 5.2*** *Add MAC Protocol*

Figure 5.2 shows the process of hierarchy in which application data is fragmented and compressed. The message authentication code is added to encrypt the SSL record header. The `SSLCompressed.type` is the higher level protocol used in the fragment. The role of SSL record header is to compress the message that is later used to append the encrypted message.

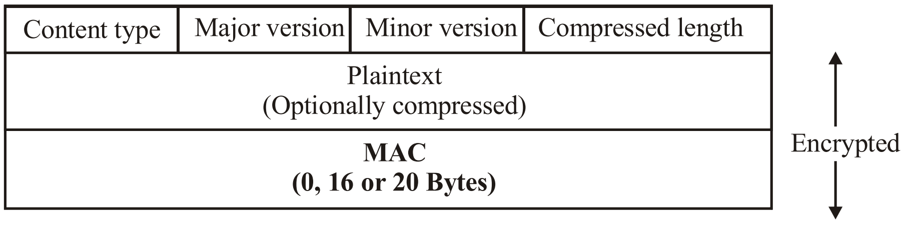


***Figure 5.3*** *Format of Record Header*

NOTES

In Figure 5.3, the record header contains the content type, major and minor versions of SSL and compressed length. The compressed length is the length (bytes) of the plain text fragment. In the client and server computing, MAC operation is used. For this, the client and server share a secret key that is used to perform the functions of `master_secret` referring initial random values from both client and server sides. The algorithm used by client and server computing for message authentication is as follows:

```
key_block=MD5(master_secret||SHA1('A'||premaster_secret||
Clienthello.random||ServerHello.random))||MD5(master_
secret||
SHA1('BB'||premaster_secret||Clienthello.random||
ServerHello.random))||MD5(master_secret||SHA1('CCC'||
premaster_secret||Clienthello.random||ServerHello.
random))…
```

This coding is done if the number of bytes is generated for `client_write MAC, server_write MAC, client_write key and server_write key.` The client sends the premaster key as `'premaster_secret'` that is encrypted with server's public key. The server decrypts the premaster key that is encrypted with Pseudo Random Function (PRF) value and server's public key. The PRF value contains the parameters `(Master_key, Input and Output_Length)` in message authentication code. The server decrypts the message to check the validity of PRF value that is exchanged with the client. If both the values match, the server uses transition of the state that would be pending to current derived keys during the authentication of the message.

**Server Authentication**

Server authentication is a part of client–server computing. SSL/TLS is generally used for authentication. A web server acquires digital certificate from available server using Certification Authority (CA). CA is a third party authority that issues digital certificates for authentication. A Digital Certificate (DC) authenticates the signature that is in fact digitally signed message. DC uses SSL/TLS (Secured Socket Layer/ Transport Layer Security) in X.509 public key infrastructure a defined by International Telecommunication Standardization Sector.

NOTES

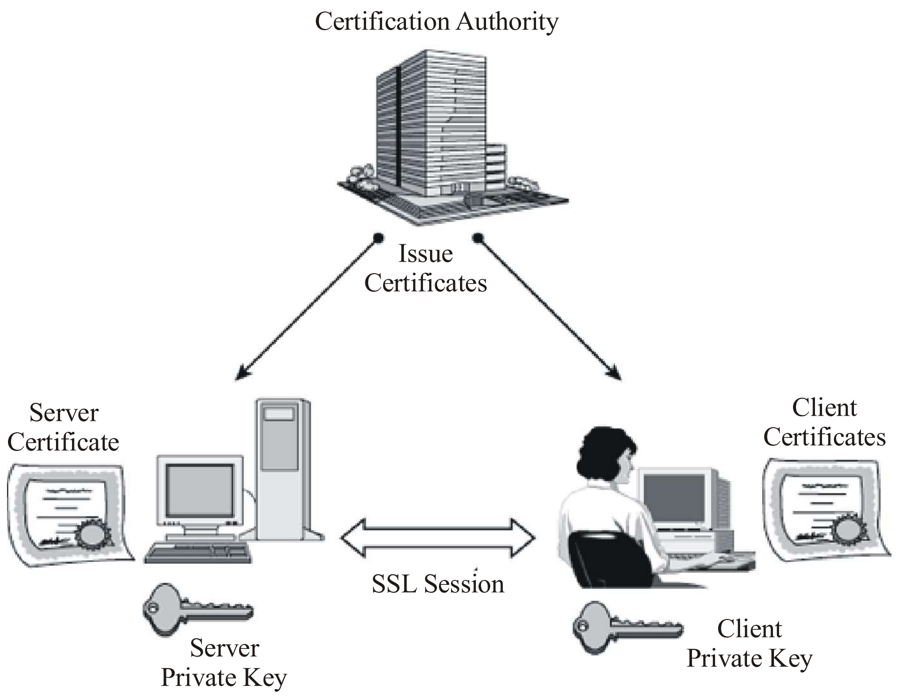


***Figure 5.4*** *Authentication in SSL/TLS*

In Figure 5.4, if client connects to the server using SSL/TLS then both client and server follow strong cryptographic algorithm. Then, the server sends X.509 certificate that contains the server's public key. The client then generates a 48-byte random number and a premaster secret key after encrypting the number used by the server's public key. The encrypted premaster secret key is sent to the server by the client. After getting premaster secret key, the server decrypts the message using the private keys. Then, both the client and the server share the same premaster secret key which is basically symmetric key used to encrypt the message. Subsequently they start communicating via generated keys. In this mechanism, only server knows the private key that decrypts the encrypted premaster secret key. Clients know the message after sending the decrypted message by server. It proves that the client is talking to the correct server. This whole mechanism represents the complete scenario of authenticating the server.

**Client Authentication**

In SSL/TSL, client authentication is not required; instead it is optional. A client stays anonymous while communicating between a web server and a browser in B2B business transaction. Therefore, they use HTTP authentication methods.

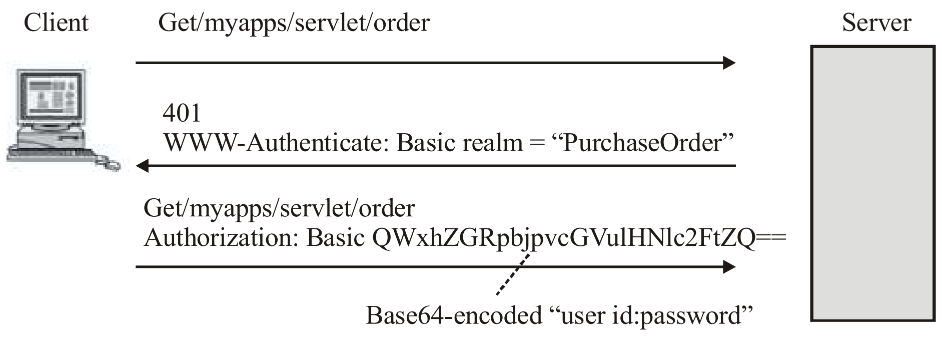

Client
Get/myapps/servlet/order
Server
401
WWW-Authenticate: Basic realm = "PurchaseOrder"
Get/myapps/servlet/order
Authorization: Basic QWxhZGRpbjpvcGVulHNlc2FtZQ==
Base64-encoded "user id:password"

***Figure 5.5*** *HTTP Authentication*

In Figure 5.5, the HTTP authentication known as RFC 2617, represents the HTTP protocol in which client and server communicate between each other via HTTP protocol. It basically considers two factors, user-id and password, to authenticate the users/clients. Sometimes, **user-id** might be user's **email-id** also. Both values are sent to authenticate without encryption and hence they are not considered as secure methods of authentication in cryptography. In this mechanism, client sends Base64-encoded user-id and password in HTTP header. If data is sent through SSL/TLS connection, it is not altered or stolen during transmission. A malicious server cannot disguise itself as genuine web server and cannot steal the password of users. For client authentication, SSL/TLS certificate is used to obtain an appropriate digital certificate before connecting to the server. A client generates the private key/public key pair to obtain the client certificate. A private key is kept as the secret key and is protected by passphrase. A passphrase works as the password with added security. It is a sequence of words used to control access to the system. The application does not maintain the database of user-id and password. It verifies the certificate that is signed by trusted CA.

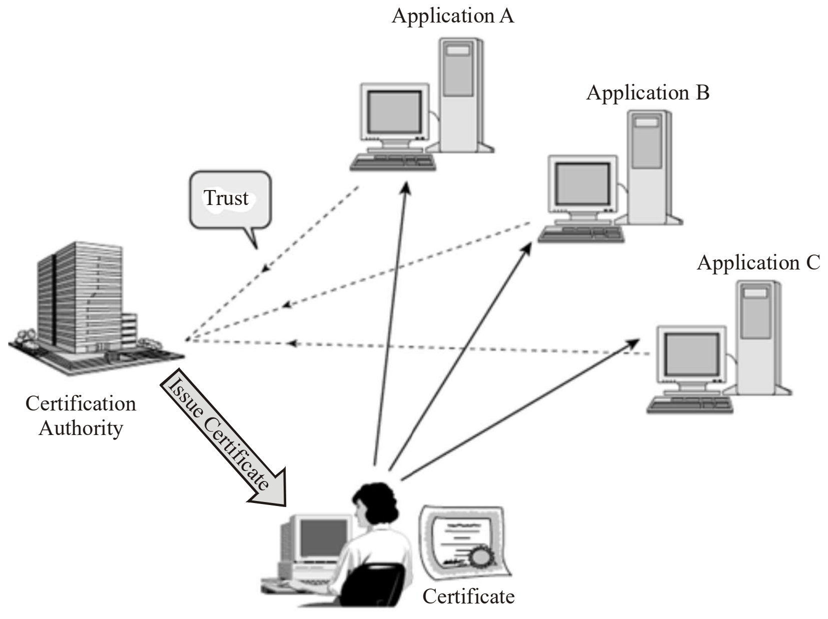


***Figure 5.6*** *Uses of Client Certificates*

Figure 5.6 shows the complete mechanism of using the client certificates. Take an example of a customer who manages ten passwords in which company **'XXX'** uses a specific password to access the system and company **'YYY'** also uses the service. Once certificate-based authentications are used by applications 'A', 'B' and 'C', the companies issue CA where they trust legitimate users. In this way, client certificates are used to authenticate the message. Tables 5.4 and 5.5 show a comparison list of various cryptographic functions and techniques used in a cryptographic algorithm:

***Table 5.4*** *Comparison List of Encryption Speed of Block Cipher*

| **Algorithm** | **Encryption Speed** | **Key Length** |
|---|---|---|
| DES | 35KB/s | 56 bits |
| 3DES | 12KB/s | 112 bits |
| IDEA | 70KB/s | 128 bits |

***Table 5.5*** *Comparison List of Hash Function of Block Cipher*

| **Algorithm** | **Encryption Speed** | **Key Length** |
|---|---|---|
| MD5 | 174KB/s | 128bits |
| SHA | 75KB/s | 160bits |

***Note:*** MD5 and SHA are hash algorithms used for authenticating the packet data. These two mechanisms provide an additional level of hashing.

## Cryptographic Protocols

Cryptographic protocols exchange messages over insecure communication medium ensuring authentication and secrecy of data. Kerberos, IPSec, SET protocol and Pretty Good Privacy (PGP) are popular examples of cryptographic protocol. Kerberos is a network authentication system used for insecure networks. PGP protocol is used for file storage applications and email services that provide authenticable and confidential services. Encryption encodes file storage locally and transmits email messages. The email service enables PGP to be used for private exchange over network. IPSec follows security architecture to the Internet. This protocol formats IP security protocol to lead the cryptographic algorithm. This protocol basically provides subnet-to-subnet and host-to-subnet topologies.

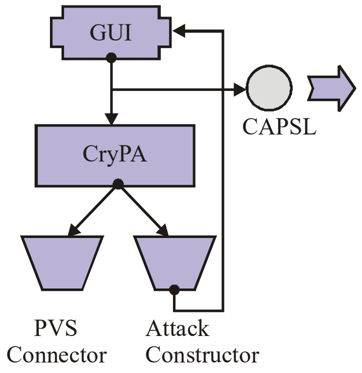


***Figure 5.7*** *Cryptographic Protocol Analyser (CRYPA) Tool*

Figure 5.7 shows that **CRY**ptographic **P**rotocol **A**nalyser (CRYPA) is based on graphics user interface specification of cryptographic protocols using construction of attack on protocol. PGP supports digital signature and encryption. This tool provides a virtual distributed environment system that provides a secure chain of handling and controlling the crypto message.

### 5.7.2 Digital Signature

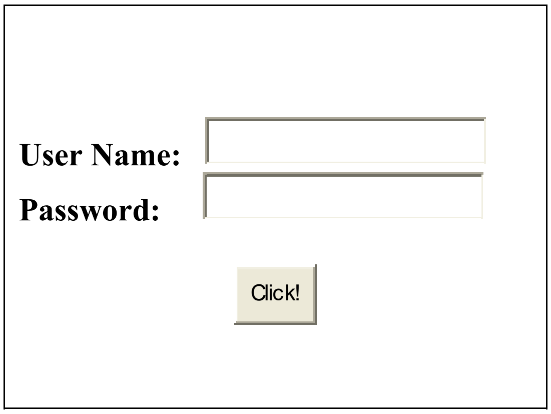
Digital Signature (DS) follows authentication mechanism. A code is attached with messages in DS. Primarily, the signature is generated by hashing the message and later this message is encrypted with the sender's private key. DS is based on public key encryption. A signature confirms that integrity and source of message is correct. **NIST** (National Institute of Standards and Technology) recognized the DS standard that uses the Secure Hash Algorithm (SHA). Message authentication protects digital signature as this way the messages are exchanged by a third party. DS is analogous to manual signature. The characteristics of DS are as follows:

- It attaches date and time along with the author of the signature.
- It authenticates the contents while the signature is being completed.
- It solves the disputes using a third party (generally in online payment by PayPal).
- It ensures that the message is not altered. The message can be in the form of electronic documents, such as email, text file, spreadsheet, etc.

A person or information is authenticated on the computer by using various techniques. Brief descriptions of these techniques are as follows:

**Password**

User name and password provides authentication. When the user logs on the system unit or application, the system asks for user name and password for checking authentication. Generally, the following type of password authentication is provided to users in which two prime fields, namely **'User Name'** and **'Password'** are required to access the system.



If two requirements do not match then authentication fails and , users are not allowed to access the system.

**Checksum**

The checksum provides a form of authentication where an invalid checksum is not recognized. If the packet of checksum is one byte long, it will have a maximum value of 255. If the sum of other bytes of the packet is 255 or less than that, the checksum contains exact value. However, if the sum of other bytes is more than 255, the checksum gives the remainder of total value.

***Table 5.6*** *Checksum Calculation*

| Byte1 | Byte2 | Byte3 | Byte4 | Byte5 | Byte6 | Byte7 | Byte8 | Total | Checksum |
|---|---|---|---|---|---|---|---|---|---|
| 212 | 232 | 54 | 135 | 244 | 15 | 179 | 80 | 1151 | 127 |

For example, in Table 5.6, 1151 is divided by 256 returning a remainder of 4.496 (round value is taken as 4). It is then multiplied with 4× 256 that equals to 1024. The value 1024 is subtracted from 1151 returning 127. In this way, the total checksum value is calculated.

**Cyclic Redundancy Check (CRC)**

The process of CRC is same as checksum. In this method, the polynomial division determines the value of CRC, which can be equal in length of 16 or 32 bits. The one difference between CRC and checksum is that CRC is more accurate. If a single bit is taken as incorrect, the CRC value does not match.

**Private Key Encryption**

A private key encryption contains a secret key that is taken as code. This mechanism encrypts a packet of information if it passed across network to the other computers. A private key requires installing the key which is essentially the same as the secret code. The code provides the key to decoding the message. For example, a coded message A is substituted by C and B is substituted by D. Therefore, A becomes C and B becomes D. If a clue is given for the message, that code is shifted by 2. The message can be decoded by your friend. If any person wants to see the message, he/she cannot get the message until and unless he/she knows the secret key. Let us take a good example of how a message is decoded. A message **'aectac'** is written as ciphertext. This is a hidden message between two persons. Person 'A' sends the message to person 'B' that he is coming to meet him but the place has not been decided. Person B does not know where they have to meet. The secret key is decided by A; he decides that **'n'** for **'a'** and **'t'** for **'c'** will be used while decoding the message. Person 'B' decodes the message. He reaches the sea coast and meets 'A' to discuss his housing plan at sea coast. The deciphered message as understood by person B is **'natant'**. Natant means 'floating in water'. It is the name of the ship by which A is coming to meet B. Therefore, it proves that digital signature issue can be solved by the mediator theory.

## Mediator Concept

Mediator concept is associated with digital signature because of the central authority involved in sending and receiving messages from person A to person B. Person M scrutinizes the message to find out where and to whom the message is being sent and received. Figure 5.8 shows how a message is sent and received between two persons.

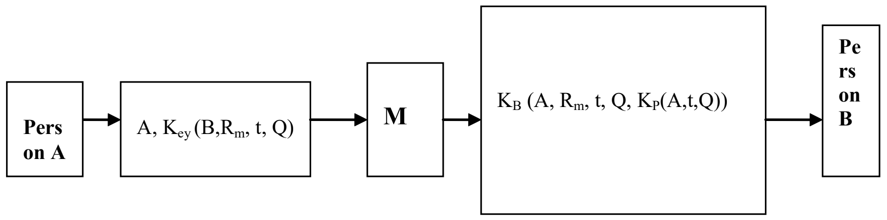

Pers on A
A, $K_{ey}$ (B,$R_m$, t, Q)
M
$K_B$ (A, $R_m$, t, Q, $K_P$(A,t,Q))
Pe rs on B

***Figure 5.8*** *Message Passing Between A and B via M*

In the figure, alphabets are used as codes when the message is sent from one person to another. Person M plays the role of a mediator who is the genuine message sender and receiver. However, Person M could also be a hacker who can alter the message or if he/she knows the secret key. Table 5.7 shows the details of the alphabets used in the message sent from A to B via M:

***Table 5.7*** *Used Alphabets and Their Functions*

| **Alphabets** | **Function** |
|---|---|
| A | Person A's entity. |
| $K_{ey}$ | Secret key code. |
| B | Person B's entity. |
| $R_m$ | Random number chosen by A. |
| t | Time at which signer signs the message. |
| Q | Attachment shows that the message has not been altered. |
| $K_B$ | Key for person B. |
| M | Mediator who scrutinizes the system (scrutinizer). |

A sends a message to B. P checks the message and decrypts it and sends it to B. B decodes the message and sends back the message to P as $K_P$(A,t,Q). This illustrates the use of symmetric key signature as B receive the correct message. Now, the whole concept of sending and receiving message depends on P and attachment Q. If person B realizes that the Q value is still attached with the message, it means that the message has not been altered. P is not a hacker. In this way, DS authenticatation might solve financial legal and commercial transactions in the presence or absence of an authorized handwritten signature.

## Public Key Encryption

A public key encryption uses private and public keys. A private key is restricted to the individual systems, whereas public key can be accessed by any system where message is to be communicated securely with the individual system. Decoding of an encrypted message can be done by a public key that is provided by the individual system as well as its own private key. Basically, the key is based on the hash value. For example, Table 5.8 shows the hash value of input number:

***Table 5.8*** *Hash Value of Input Number*

| Input Number | Hashing Algorithm | Hash Value |
|---|---|---|
| 12421 | Input # × 131 | 1627151 |

In the table, the value 1627151 is the result of multiplication of 131 and 12421. If the multiplier is 131, the value comes as 1627151. The public key can use large values to encrypt, such as 40bit and 128bit. The value 128bit can have $2^{128}$ combinations.

A digital signature follows the following operations:

**Key pair generation:** In this process, a public and a private key pair is generated.

**Generation operation:** In this process, a signature is produced for a message with private key.

**Verification operation:** In this process, a signature is checked with the public key.

Digital signature provides data integrity, signer authenticity, authorization, security, accountability and non-repudiation. These are the mechanisms that are frequently associated with digital signatures. These mechanisms are inter-related with each other and hence are popular in transaction of digital cash, e-money transfer across net, etc.

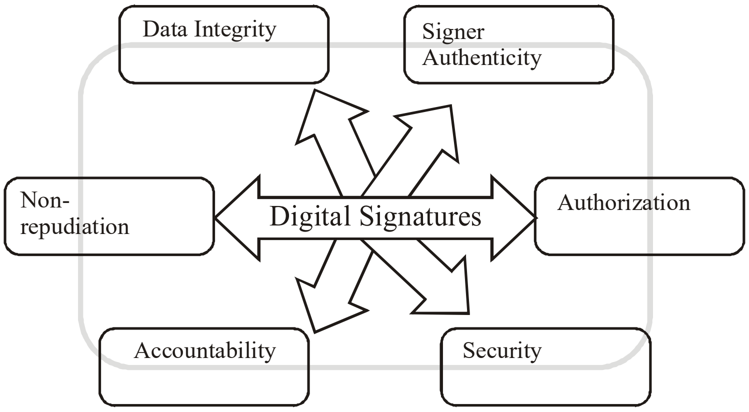


***Figure 5.9*** *Mechanisms of Digital Signature*

Figure 5.9 shows how various mechanisms are associated with digital signatures.

## Properties of Digital Signature

The properties of digital signature are as follows:

- Digital signature cannot be forged by person.
- Once signer signs the document or message, it cannot be forged.
- Signer cannot replace the sign once the message is signed.

The concept of digital signature can be explained with the help of example. Assuming there are two persons and a message is being sent from person 'A' to person 'B'. With reference to cryptography, person **'A'** encrypts the message to person **'B'** using public key. The message is signed by person **'A'** with a secret key. The secret is the code in which the ciphertext can be decoded. Person **'B'** decrypts the message with a personal secret key and then verifies it with **A**'s public key. If the code is matched, **'B'** gets the correct message. The hash coding condenses the message into 100 to 200 bits range. Signing of hash message is faster than signing the whole message. The one-way hash function ensures that no two messages will have the same value.

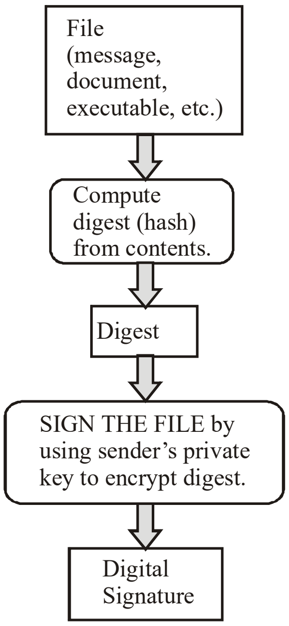

File (message, document, executable, etc.)
Compute digest (hash) from contents.
Digest
SIGN THE FILE by using sender's private key to encrypt digest.
Digital Signature

***Figure 5.10*** *An Encrypted Digest Using Digital Signature*

In Figure 5.10, a digital signature is taken as document, message, driver or program that is being signed. Then the message is encrypted using the public and/or private key. The document or message is signed by using the sender's private key that encrypts the digest. Once the message is encrypted, the file cannot be altered by an attacker.

### Verifying a Digital Signature

Once signer signs, the data is verified. Verifying signature confirms that the signed data has not been altered. If the digital signature is verified, it can be decrypted by using a public key that produces the original hash value. If the two hash values match, the signature is exactly same.

Various software, such as Multilevel Digital Signateure System, MyLiveSignature, Random Signature Changer, SignetSure, SignaturePilot, SignaturePilotPro, 602XMLFormFilter, AzSDK MD5Sum, Signit, etc., are used by various commercial transaction hubs for signing and verifying the digital signature.

**CHECK YOUR PROGRESS**

9. What do you understand by an equivalence relation?
10. What do you understand by public key encryption?
11. What role does the certification authority (CA) play in data authentication?
12. What do cryptographic protocols do?
13. How can the decoding of an encrypted message be done?

## 5.8 SUMMARY

In this unit, you have learned that:

- There are two types of numbers: prime numbers and composite numbers.
- The division algorithm is a well-defined procedure for achieving a specific task and can be used to find the greatest common divisor of two integers.
- Numbers can be divided into two based on their divisibility by 2:
  - (i) Even numbers: They do not leave any remainder.
  - (ii) Odd numbers: They are not evenly divisible by 2.
- An inequality statement identifies the relative size or order of two objects to confirm whether they are similar or not.
- Inequalities are influenced by these properties: trichotomy, transitivity, addition and subtraction and multiplication and division.
- The greatest common divisor (G.C.D) is the largest positive integer that divides a number entirely, i.e., without leaving any remainder.
- Euclid's algorithm determines the G.C.D of two elements of any Euclidean domain.
- The Fibonacci number sequence is named after Leonardo of Pisa, who was famously known as Fibonacci.
- In the Fibonacci series, 0 and 1 are the first two Fibonacci numbers; all the other numbers of the series are the sum of the previous two numbers.
- Congruent relation states that if *a, b, c* (*c* > 0) are integers, then *a* is congruent to *b* modulo *c* if *c* divides $a - b$.
- Equivalence relation refers to a relation *R* on a set *A* which is in equivalence in case *R* is reflexive, symmetric and transitive.

- Public key encryption refers to a sort of cipher architecture recognized as public key cryptography that uses two keys, or a key pair for encrypting and decrypting data.

NOTES

## 5.9 KEY TERMS

- **Prime number:** An integer $p > 1$ is called a *prime number* if 1 and $p$ are the only divisors of $p$.
- **Composite number:** A composite number is an integer $n > 1$ such that $n$ is not prime.
- **Divisor:** A divisor, in mathematics, is an integer $n$, also termed as factor of $n$, which evenly divides $n$ without leaving any remainder.

## 5.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Properties of prime numbers are as follows:
   (*i*) If a prime number $p$ divides $ab$, then either $p \mid a$ or $p \mid b$.
   (*ii*) If a prime number $p$ does not divide an integer $a$, then the highest comman factor (HCF) of $p$ and $a$ is 1 (by convention HCF is taken to be a + ve integer).
2. In mathematics, the division algorithm is a theorem which expresses the outcome of the usual process of division of integers.
3. Numbers which are evenly divisible by 2 without leaving any remainder are called even numbers.
4. In mathematics, an inequality is a statement that defines the relative size or order of two objects to check whether they are similar or not.
5. The GCD of two or more non-zero integers is the largest positive integer that divides a number without leaving a remainder.
6. Linear Diophantine equation is an equation $ax + by = c$ in two unknowns, $x$ and $y$, where $a$, $b$, $c$ are given integers and one of $a$, $b$ is not zero.
7. In number theory, the Euclidean algorithm also termed as Euclid's algorithm, is a very important algorithm that is used to determine the greatest common divisor of two elements of any Euclidean domain.
8. The first two Fibonacci numbers are 0 and 1, and all other numbers are the sum of the previous two numbers.
9. A relation $R$ on a set $A$ is called an equivalence relation if $R$ is reflexive, symmetric and transitive.
10. Public key inscription refers to a sort of cipher architecture recognized as public key cryptography that uses two keys for encrypting and decrypting data.

NOTES

11. Certification Authority (CA) serves as a third party that issues digital certificates for data authentication.
12. Cryptographic protocols exchange messages over insecure communication medium ensuring authentication and secrecy of data.
13. Decoding of an encrypted message can be done by a public key that is provided by the individual system as well as its own private key. Basically, the key is based on the hash value.

## 5.11 QUESTIONS AND EXERCISES

**Short-Answer Questions**

1. What does the concept of strict inequality state?
2. Briefly explain the fundamental theorem of arithmetic.
3. What do you understand by prime numbers?
4. Differentiate between trivial and non-trivial divisors.
5. What are the various keywords that are used in crypto algorithm? Also state their functions.
6. Write a short note on private key encryption.

**Long –Answer questions**

1. Explain the properties of inequalities.
2. Prove the Euclidean algorithm.
3. Explain the basis representation theorem.
4. Explain how http authentication is done in business-to-business transactions.
5. Describe the various techniques used for the authentication of data on computer.

## 5.12 FURTHER READING

Lipschutz, Seymour and Lipson Marc. *Schaum's Outline of Discrete Mathematics,* 3rd edition. New York: McGraw-Hill, 2007.

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms.* Hyderabad: Orient BlackSwan, 2008.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

Brassard, Gilles and Paul Bratley. *Fundamentals of Algorithms*. New Delhi: Prentice Hall of India, 1995.

Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*. New Jersey: Pearson, 2006.

Baase, Sara and Allen Van Gelder. *Computer Algorithms – Introduction to Design and Analysis*. New Jersey: Pearson, 2003.

Mott, J.L. *Discrete Mathematics for Computer Scientists*, 2nd edition. New Delhi: Prentice-Hall of India Pvt. Ltd., 2007.

Liu, C.L. *Elements of Discrete Mathematics*. New Delhi: Tata McGraw-Hill Publishing Company, 1977.

Rosen, Kenneth. *Discrete Mathematics and Its Applications*, 6th edition. New York: McGraw-Hill Higher Education, 2007.

## NOTES

## NOTES

## NOTES