UNIT 1 ELEMEMTARY ALGORITHMICS

Structure **Page Nos.**

1.0 Introduction 7
1.1 Objectives 9
1.2 Example of an Algorithm 9
1.3 Problems and Instances 10
1.4 Characteristics of an Algorithm 12
1.5 Problems, Available Tools & Algorithms 14
1.6 Building Blocks of Algorithms 17
1.6.1 Basic Actions & Instructions
1.6.2 Control Mechanisms and Control Structures
1.6.3 Procedure and Recursion
1.7 Outline of Algorithmics 27
1.7.1 Understanding the Problem
1.7.2 Analyzing the Problem
1.7.3 Capabilities of the Computer System
1.7.4 Approximate vs Exact Solution
1.7.5 Choice of Appropriate Data Structures
1.7.6 Choice of Appropriate Design Technology
1.7.7 Specification Methods for Algorithms
1.7.8 Proving Correctness of an Algorithm
1.7.9 Analyzing an Algorithm
1.7.10 Coding the Algorithm
1.8 Summary 32
1.9 Solutions/Answers 33
1.10 Further Readings 40

1.0 INTRODUCTION

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

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

David Berlinski in The Advent of the Algorithm, 2000.

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

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

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

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

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

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

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

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

Notation for expressing algorithms

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

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

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

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

Role and Notation for Comments

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

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

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

Mathematical Notations shall be introduced in Section 2.2.

1.1 OBJECTIVES

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

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

1.2 EXAMPLE OF AN ALGORITHM

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

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

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

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

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

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

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

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

Algorithm GCD-Euclid (m, n)

```
{This algorithm computes the greatest common divisor of two given positive
    integers}
    begin {of algorithm}
      while $n \neq 0$ do
       begin {of while loop}
           $r \leftarrow m \bmod n$;
    {a new variable is used to store the remainder which is obtained by dividing
    m by n, with $0 \le r < m$}
```

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

1.3 PROBLEMS AND INSTANCES

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

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

are given by the equation

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

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

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

we get the *particular* equation

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

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

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

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

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

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

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

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

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

with $a \neq 0$, b and c as real numbers, has *infinitely* many *instances*, each obtained by giving some specific real values to a, b and c, taking care that the value assigned to *a* is not zero. However, all problems may not be of generic nature. For *some problems, there may be only one instance/question* corresponding to each of the problems. For example, the problem of finding out the largest integer that can be stored or can be arithmetically operated on, in a given computer, is a single-instance problem. Many of the interesting problems like the ones given below, are just **single-instance problems**.

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

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

$$x^5 - 32 = 0$$

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

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

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

1.4 CHARACTERISTICS OF AN ALGORITHM

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and add it to x.

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

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

Ex. 2) For each of the following, give one example of a method, which is not an algorithm, because
(i) the method is not finite
(ii) the method is not definite
(iii) the method is not effective but finite.

1.5 PROBLEMS, AVAILABLE TOOLS & ALGORITHMS

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

First Algorithm:

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

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

$$\begin{array}{r} 4\,2\,6 \\ \underline{3\,7} \\ 2\,9\,8\,2 \\ \underline{1\,2\,7\,8\,0} \\ \underline{1\,5\,7\,6\,2} \end{array}$$

Second Algorithm:

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

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

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

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

The algorithm constitutes the following steps:

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

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

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

Third Algorithm:

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

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

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

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

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

Then

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

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

For example, m = 7 and n = 12

Then

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

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

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

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

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

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

[*] If, initially, Second $\neq$ 0, then Second $\neq$ 0 in the subsequent calculations also.

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

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

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

Let First = 9 and Second = 16

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

Let us take First_1 = [First/2] = 4
Second_1 = 2 * Second = 32 and
Partial-Result = First_1 * Second $_1$.
Then from the above argument, we get
First * Second = First_1 * Second_1 + Second
= Partial-Result_1 + Second.

Here, we may note that as First = 9 is odd and hence Second is added to Partial-Result. Also
Partial-Result_1 = 4*32 = (2 * 2 + 0) * 32 = (2 * 2) * 32 + 0 * 32
= 2* (2 * 32) = First $_2$ * Second $_2$.

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

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

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

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

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

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

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

1.6 BUILDING BLOCKS OF ALGORITHMS

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

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

1.6.1 Basic Actions & Instructions

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

variable ← *expression*;

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

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

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

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

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

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

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

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

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

read (i,j, ,...);

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

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

write (i, j ,….);

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

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

the algorithm gives the following output:

This equation has no real roots.

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

1.6.2 Control Mechanisms and Control Structures

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

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

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

For example, the sequence of the three next lines

```
A; B;
C;
D;
```

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

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

Begin A;B;C;D end.

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

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

If Q then do A else do B,

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

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

This new situation may be denoted by

If Q then do A

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

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

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

The above intention can be expressed through the following notation:

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

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

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

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

Then the corresponding notation may be:

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

where M is an integer variable

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

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

While (Q) do begin S end;

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

```
algorithm Sum_First_N_1
begin
read (n); {assuming value of n is an integer $\ge 1$}
        $x \leftarrow 1$  ;  $SUM \leftarrow 1$;
        while (x < n) do ......................................... ($\alpha$1)
        begin
                $x \leftarrow x + 1$;
                $SUM \leftarrow SUM + x$
        end; {of while loop}................................... ($\beta$1)
        write ('The sum of the first', n, 'natural numbers is' SUM)
        end. {of algorithm}
```

Explanation of the algorithm Sum_First_N_1:

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

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

```
begin
        $x \leftarrow x+1;$
        $SUM \leftarrow SUM + x;$
end
```

and as a consequence of execution of this composite statement

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

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

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

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

The sum of the first 3 natural numbers is 6.

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

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

Do S while (Q)

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

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

The above instruction is denoted in the programming language Pascal by

Repeat S until (not Q)

Example 1.6.2.3: Algorithm Sum_First_N_2

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

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

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

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

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

__OR__

for v ← i to (i + − 1) do

begin S end;

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

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

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

```
algorithm Sum_First_N_3
begin
    read (n);

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

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

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

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

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

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

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

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

1.6.3 Procedure and Recursion

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

(i) Procedure
(ii) Recursion

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

1.6.3.1 Procedure

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

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

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

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

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

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

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

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

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

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

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

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

Example 1.6.3.1.1

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

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

Program Diagonal-Length

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

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

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

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

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

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

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

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

We summarise the above discussion about procedure as follows:

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

1.6.3.2 Recursion

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

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

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

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

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

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

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

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

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

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

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

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

fact = 3 * factorial (2)

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

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

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

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

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

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

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

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

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

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

1.7 OUTLINE OF ALGORITHMICS

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

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

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

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

(i) Sorting problems

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

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

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

1.7.1 Understanding the Problem

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

(i) **the type of problem,** so that if a method of solving problems of the type, is already known, then the known method may be applied to solve the problem under consideration.

(ii) **the type of inputs and the type of expected/desired outputs,** specially, the illegal inputs, i.e., inputs which are not acceptable, are characterized at this stage. For example, in a problem of calculating income-tax, the income can not be non-numeric character strings.

(iii) **the range of inputs,** for those inputs which are from ordered sets. For example, in the problem of finding whether a large number is prime or not, we can not give as input a number greater than the Maximum number (Max, mentioned above) that the computer system used for the purpose, can store and arithmetically operate upon. For still larger numbers, some other representation mechanism has to be adopted.

(iv) **special cases of the problem,** which may need different treatment for solving the problem. For example, **if** for an expected quadratic equation $ax^2+bx+c=0$, a, the coefficient of x^2, happens to be zero **then** usual method of solving quadratic equations, discussed earlier in this unit, can not be used for the purpose.

1.7.2 Analyzing the problem

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

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

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

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

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

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

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

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

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

2	8	7
1	3	5
	6	4

to some final state say

1	2	3
8		4
7	6	5

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

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

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

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

There are many other characteristics of a problem viz.,

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

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

1.7.3 Capabilities of the Computer System

We have already discussed the importance of the step in Section 1.5, where we noticed how, because of change in computational capabilities, a totally different algorithm has to be designed to solve the same problem (*e.g., that of multiplying two natural numbers*).

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

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

However, we restrict ourselves to sequential algorithms only.

1.7.4 Approximate vs Exact Solution

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

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

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

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

1.7.5 Choice of Appropriate Data Structures

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

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

1.7.6 Choice of Appropriate Design Technology

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

We have already enumerated various design techniques and also various problem domains which have been rigorously pursued for computational solutions. For each problem domain, a particular set of techniques have been found more useful, though other techniques also may be gainfully employed. A major part of the material of the course, deals with the study of various techniques and their suitability for various types of problem domains. Such a study can be a useful guide for solving new problems or new problems types.

1.7.7 Specification Methods for Algorithms

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

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

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

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

1.7.8 Proving Correctness of an Algorithm

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

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

1.7.9 Analyzing an Algorithm

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

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

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

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

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

1.7.10 Coding the Algorithm

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

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

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

1.8 SUMMARY

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

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

2. The differences between the related concepts of

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

,

   are explained

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

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

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

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

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

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

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

are discussed

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

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

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

1.9 SOLUTIONS/ANSWERS

Ex.1)

Example Problem 1: Find the roots of the Boolean equation

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

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

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

The problem has only four instances viz

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

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

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

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

Ex.2)

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

Consider the Alternating series

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

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

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

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

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

```
Begin

    If Color-ball= 'black' then
              x← x+1;
        else
              If color-ball= 'white' then
              x←x+2;
        else if colour-ball = 'blue' then x←x+3
        else
                x←x+4;
    end.
```

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

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

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

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

Ex.3)

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

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

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

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

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

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

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

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

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

Let first=22 and second=16

Then

*first*second = 22*16= (7*3+1)*16*
*=7*3*16+1*16 = 7*(3*16)+1*16*
=[first/3](3*second)+1*second*
= [first/3] (3* second)+ remainder * second*

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

Let first=23 and second=16. Then
*First * Second = 23*16=(7*3+2)*16*
*=(7*3)* 16+2*16*
=7(3*16)+2*16=[first/3]*(3*second)+2*second*
= [first/3] (3* second)+ remainder * second*

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

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

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

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

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

```
[**]$\text{First}_1$=[First/3] ;  $\text{Remainder}_1 \leftarrow$ First – $\text{First}_1$*3;
Partial-$\text{Result}_1 \leftarrow$  $\text{First}_1$*$\text{Second}_1$;
Partial-Result$\leftarrow$Partial-$\text{Result}_1$+$\text{Remainder}_1$*Second;
```

Step 3:
{For computing Partial-Result$_1$, replace First by First$_1$; Second by Second$_1$, and Partial-Result by Partial-Result$_1$ in Step 2 and repeat Step 2}

```
First$\leftarrow$ $\text{First}_1$ ;  Second=Second
Partial-Result $\leftarrow$ Partial-$\text{Result}_1$
And Go To Step2
```

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

[**] *First = 3*First$_1$+Remainder$_1$*

with $0 \leq \text{Remainder}_1 < 2$
$\text{Second}_1 = 3{*}\text{Second}$
$\text{Partial-Result}_1 = \text{First}_1 * \text{Second}_1$

$\text{Partial-Result} = \text{First} * \text{Second} = (\text{First}_1{*}3+\text{Remainder}_1){*}(\text{Second})$
$=(\text{First}_1{*}3){*}\text{Second}+\text{Remainder}_1{*}\text{Second}$
$=\text{First}_1{*}(3{*}\text{Second})+\text{Remainder}_1{*}\text{Second}$
$=\text{First}_1{*}\text{Second}_1+\text{Remainder}_1{*}\text{Second}$
$=\text{Partial-Result}_1+\text{Remainder}_1{*}\text{Second}$
Where $0 \leq \text{Remainder}_1 < 2$

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

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

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

Ex. 4)

Algorithm Real-Roots-Quadratic

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

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

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

Ex. 5)

Algorithm Real-Roots-General-Quadratic

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

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

Ex.6) (i)

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

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

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

The sum of the first 3 natural numbers is 6.

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

Ex 6 (ii)

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

SUM← Sum+x

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

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

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

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

Ex.7)

For computing SUM (5) by the algorithm
As n=5 ≠ 0
therefore
$S_5 \leftarrow$ n+ SUM (n-1) = 5+SUM(4)
{It may be noted that in different calls of the procedure SUM, the variable names occurring within the procedure, denote different variables. This is why instead of S we use the names S_i for i=5,4,3,………}

Therefore, in order to compute SUM(5), we need to compute SUM (4)
n=4 ≠ 0, therefore
$S_4 \leftarrow$ 4 + SUM (3)
Continuing like this, we get
$S_3 \leftarrow$ 3 + SUM (2)
$S_2 \leftarrow$ 2 + SUM (1)
$S_1 \leftarrow$ 1 + SUM (0)

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

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

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

1.10 FURTHER READINGS

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

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

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

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

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

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

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