Unit 9 Elementary Algorithms

Structure:

9.1 Introduction

In this unit, you are going to study about the concepts that are basically needed to frame an elementary algorithm, which you have already discussed in your earlier classes.

Problem may be a state of mind of a living being, of not being satisfied with some situation. However, for our purpose, we may take the unsatisfactory / unacceptable / undesirable situation itself, as a problem.

One way of looking at a possible solution of a problem is, as a sequence of activities (if such a sequence exists at all), that if carried out using allowed available tools, leads us from the unsatisfactory (initial) position to an acceptable, satisfactory or desirable 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 utensil 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.

Objectives

After studying this unit, you should be able to:

- use notations for expressing algorithms
- apply the different characteristics of an algorithm
- use the different building blocks in algorithms in an effective manner
- use the concept of recursion effectively in mathematical concepts

9.2 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.

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 $m \leq n$, *A[m, n]* is used to denote the variables

A[m], A[m+1], …., A[n]. However, we must note that another similar notation *A[m, n]* is used to indicate the element of the matrix (or two – dimensional array) *A*, which is in m^{th} row and n^{th} column.

9.3 Role and Notation for Comments

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}.

Self Assessment Questions

1. Comments help the human reader of the algorithm to better understand the ————————.
2. The symbol ———————— is used for assignment.

9.4 Example of an Algorithm

Before going in to the detail 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 and n*

El: {Find remainder}. Divide m by n and let r be the (new) remainder

{we have $0 \leq r < n$}

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

Otherwise,

E3: {Interchange} Let the new value of m be the current value of n and the new value of n be the current value of r. Go back to Step E_1.

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. This algorithm is considered as the first among non – trivial algorithms. However, the word 'algorithm' itself came into usage quite late. The word is derived form 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 familiarize 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 integer}
begin {of algorithm}
while $n \neq 0$ do
begin {of while loop}
$r \leftarrow m$ mod $n$;
{a new variable is used to store the remainder which is obtained by dividing m by n, with $0 \leq r < m$}
$m \leftarrow n$;
{the value of n is assigned as new value of m; but at this stage, value of n remains unchanged}
$n \leftarrow r$,
{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}
```

Self Assessment Question

3. “A new variable is used to store the remainder which is obtained by dividing m by n, with $0 \leq r < m$”. This is denoted by ——————— .

9.5 Problems and Instances

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

$$ax^2 + bx + c = 0 \qquad a \neq 0 \qquad (1)$$

are given by the equation

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

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

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

Using *(2)*, the roots of *(3)* are given by

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

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

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 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 in a given computer is a single-instance problem.

Self Assessment Question

4. Four-Colour Problem requires us to find out whether a political map of the world, can be drawn using only ————————.

9.6 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.

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.

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.

However, the method is not definite, as two different executions may yield different outputs.

Inputs

An algorithm has zero or more but only finite, number of inputs.

Examples of algorithm 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 letters in the alphabet of the computer system being used.
iii) Find the sum *S* of the form *I + 2 + 3 +………* Where S is the largest integer less than or equal to MAX defined in example (i) above.

Outputs

An algorithm has one or more outputs. The requirement of at least one output is obviously essential. Otherwise we can not know the answer/ solution provided by the algorithm.

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

Effectiveness

An algorithm should be effective. This means that each of the operations 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 cannot be effective also.

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

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

$e = 1 + \frac{1}{(1!)} + \frac{1}{(2!)} + \frac{1}{(3!)} + ...$ 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 $\frac{1}{n!}$ 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 $\frac{1}{n!}$ is definite (at least for a given machine).

Self Assessment Question

5. An algorithm has one or more ————————

9.7 Building Blocks of Algorithms

Next, we enumerate the basic actions and corresponding instructions used in a computer system based on Von Neumann 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.

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 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 external storage. In the latter case, the identity of the secondary 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 monitory or a secondary storage.

Control Mechanism 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., assignment, 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 instruction occurs 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 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 off the instructions,

- One after the other on successive lines, or even on the same 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 next lines

A;
B;
C;
D;

denotes that the execution of *A* is to be followed by execution of *B*, to be inturn 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 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 (components) 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 *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 situation 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 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: We are to write a program segment that converts % of marks to grades as follows

% of marks *(M)*	Grades *(G)*
$M \geq 80$	*A*
$60 \leq M < 60$	*B*
$50 \leq M < 60$	*C*
$40 \leq M < 50$	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 repetitive execution of a sequence of actions is the basis of expressing long processes by comparatively small number of instructions. 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.

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 begins *S* end;

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

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

Explanation of the algorithm *Sum_First_N_1:*

Initially, an integer value of the variable *n* is read. Just to simplify the argument, we assume that the integer $n \geq 1$. The next two statements assign value *1* to each of the variables *x* and SUM. Next, we get the execution of the while – loop. The while – loop extends from the statement *(α 1)* to *(β 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 *(β 1)* are skipped.

Suppose we read *3* as value of n, and (initially) x equal 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 ← x + 1;
SUM ← SUM + x;
end
```

and as a consequence of execution of this composite statement

The value of *x* becomes *1* 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 *(α1)* and *(β1)*,(including *(α1)* and *(β1)* is again executed).

By our assumption $n = 3$ and it did not change since the initial assumption about *n*; however, *x* has become *2*. Therefore, $x < n$ is again satisfied. Again the rest of the while loop is executed. Because of the execution of the rest of the loop, *x* becomes *3* and *SUM* becomes the algorithm comes to the execution of first statement of while – loop, i.e., while *($x < n$)* do, which tests whether $x < n$. At this stage $x = 3$ and $n = 3$. Therefore, $x < n$ is false. Therefore, all statement upto, and including (β1), are x < n and skipped.

Then the algorithm executes the next statement, viz, write (‘The sum of the first’ n, ‘natural number 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.

Self Assessment Question

6. The variable *x* is called ———————— of the for-loop

9.8 Procedure and Recursion

Though the above mentioned three control structure, 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

First we consider the advanced control structure procedure.

9.8.1 Procedure

Among a number of terms that are used, instead 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.

Under this mechanism, the sequence of instructions expected to be repeatedly used in one or more algorithms, is written only once, 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 >1]
    <declarations>
    <sequence of instructions expected to occur
      repeatedly                                          (1)
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), <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 indented code, a statement of the form

sum – square *(a, b)*; (2)

is written, where values of a and b are defined before the location of the statement under (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)* is replaced by the type of the value to be passed, in this case integer.

In cases of procedure 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.

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:

```
Procedure sum-square (a, b: integer) integer;
{denoted the inputs a and b integers and the output is also an integer}
S: integer;
{to store the required number}
begin
$S \leftarrow a^2 + b^2$
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}
$L_1$, $L_2$ 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
$D \leftarrow$ square – root (sum – square $(L_1, L_2)$)
Write (‘For sides of given length’, $L_1, L_2,$ ‘the required diagonal length is’ $D$);
read $(L_1, L_2)$
end
```

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

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

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

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

9.8.2 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)

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 $\angle 4$ 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 'fac' be the variable which is used to pass the value by the procedure factorial to a calling program. The variable 'fac' is initially assigned value 1, which is the value of factorial (1),

```
 Procedure factorial (n)
 fac; integer:
 begin
          fac ← 1
          if n equals 1 then return fac
          else begin
          fac ← n * factorial (n – 1)
          return (fac)
          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, …… up to n finally deliver the desired result.

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 an in-built mechanism in the computer system that supports the calling of a procedure by itself, eg., there may be an 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)
        Begin s ← n + SUM (n – 1)
        end;
end;
```

Self Assessment Questions

7. ———————— is a self-contained algorithm which is written for the purpose of connecting into another algorithm.
8. A procedure, which can call itself, is said to be ————————

9.9 Outline of Algorithmics

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 our study, we shall be explaining and practising algorithms. We see 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) Bracketing
v) Branch and bound
vi) Searches and traversals.

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.

Understanding the problem

Understanding allows appropriate action. This step forms the basis of the other steps to be discussed. For understanding the problem, we should read the statement of the problem, if required, a number of times. Then we should 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 cannot be non-numeric character strings. 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 cannot give as input a number greater than the maximum number 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.

iii) 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, happened to be zero then usual method of solving quadratic equations, cannot be used for the purpose.

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 regard are discussed below:

i) Whether the problem is decomposable into independent smaller or easier sub problems, 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

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

can be decomposed into smaller and simpler problem viz.,

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

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 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, 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 be in a position, because of the adversary's 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 out tools, techniques and strategies for solving the problem.

Capabilities of the computer system

We know that because of change in computational capabilities, a totally different algorithm has to be designed to solve the problem (eg. 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 powerful machines based on parallel distributed architectures, are also increasingly becoming available. Algorithms, that use 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.

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 (eg., n = 3). We have to determine in advance what approximation is acceptable, eg., 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 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 Traveling 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.

Self Assessment Question

9. In order to find the shortest paths, one should find the cost of covering each of the ———————— different paths covering the n given cities

9.10 Specification Methods for Algorithms

An algorithm is a description 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 meaning, which, 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) Flow chart 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.

Analyzing an Algorithm

Generally, the resources that are taken into consideration for analyzing 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 resources for the purpose.

Self Assessment Question

10. A —— notation is a sort of dialect obtained by mixing some programming language constructs with natural language descriptions of algorithms

9.11 Summary

In this unit we have studied

- 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]*
- The comments may be inserted at any place within an algorithm
- The difference between the two concepts viz., ‘problem’ and ‘instance’
- The characteristics of an algorithm is very important to design it

- The following two building blocks
  i) Basic actions and instructions
  ii) Control Mechanism and control structure
  are very much necessary to frame an algorithm
- The following two advanced control structures
  i) Procedure
  ii) Recursion
  have proved to be quite useful in facilitating the expression of complex algorithms
- The different outline of Algorithms which is very much necessary to solve a problem effectively
- The different specification methods and the resources that are taken into consideration for analyzing algorithms

9.12 Terminal Questions

1. What is the use of comments in an Algorithm?
2. Briefly explain the concept of "Problems and Instances"
3. Give the different characteristics of an algorithm
4. Explain the different control mechanisms and control structures in an Algorithm
5. Explain the concept of "Analyzing the Problem" in Algorithms
6. Explain "Specification Methods" in an Algorithm
7. What do you mean by "Recursion in Algorithms"? Explain?

9.13 Answers

Self Assessment Questions

1. Algorithm
2. $\leftarrow$
3. r $\leftarrow$*m mod n*
4. Four colours
5. Outputs
6. Index varialbe
7. Procedure
8. Recursive procedure/algorithm
9. n!
10. Pseudo code

Terminal Questions

1. Refer to section 9.3
2. Refer to section 9.5
3. Refer to section 9.6
4. Refer to section 9.7
5. Refer to section 9.10
6. Refer to section 9.11
7. Refer to section 9.9