CHAPTER 1

Flowchart Techniques

1.1 Programming Aids

Programmers use different kinds of tools or aids which help them in developing programs faster and better. Such aids are studied in the following paragraphs. Important aids available to a programmer are flowcharting and decision tables which help him in constructing programs very fast and very easily.

1.1.1 Flowcharts

A flowchart is a graphical representation of the sequence of operations in an information system or program. Program flowcharts show the sequence of instructions in a single program or subroutine.

Flowchart uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having *arrow* marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. Since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected easily. Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.

A flowchart is basically the plan to be followed when the program is written. It acts like a road map for a programmer and

guides him in proceeding from the starting point to the final point while writing a computer program.

For a beginner it is strongly recommended that a flowchart be drawn first in order to reduce the number of errors and omissions in the program. Moreover, it is a good practice to have a flowchart along with a computer program as it is very helpful during the testing of the program and also in incorporating any modifications in the program.

1.1.2 Flowchart Symbols

Only a few symbols are needed to indicate the necessary operations in a flowchart. These symbols have been standardized by the American National Standards Institute (ANSI). These symbols are shown in Figure 3.1 and their functions are discussed below.

Terminal

The terminal symbol, as the name implies, is used to indicate the starting (BEGIN), stopping (END), and pause (HALT) in the program logic flow. It is the first symbol and the last symbol in the program logic. In addition, if the program logic calls for a pause in the program, that also is indicated with a terminal symbol. A pause is normally used in the program logic under some error conditions or in case the forms had to be changed in the computer's line printer during the processing of that program.

Input/Output

The input/output symbol is used to denote any function of an input/output device in the program. If there is a program instruction to input data from a disk, tape, card reader, terminal, or any other type of input device, that step will be indicated in the flowchart with an input/output symbol. Similarly, all output instructions, whether it is output on a printer, magnetic tape, magnetic disk, terminal screen, or any output device, are indicated in the flowchart with an input/output symbol.

Processing

A processing symbol is used in a flowchart to represent arithmetic and data movement instructions. Thus, all arithmetic

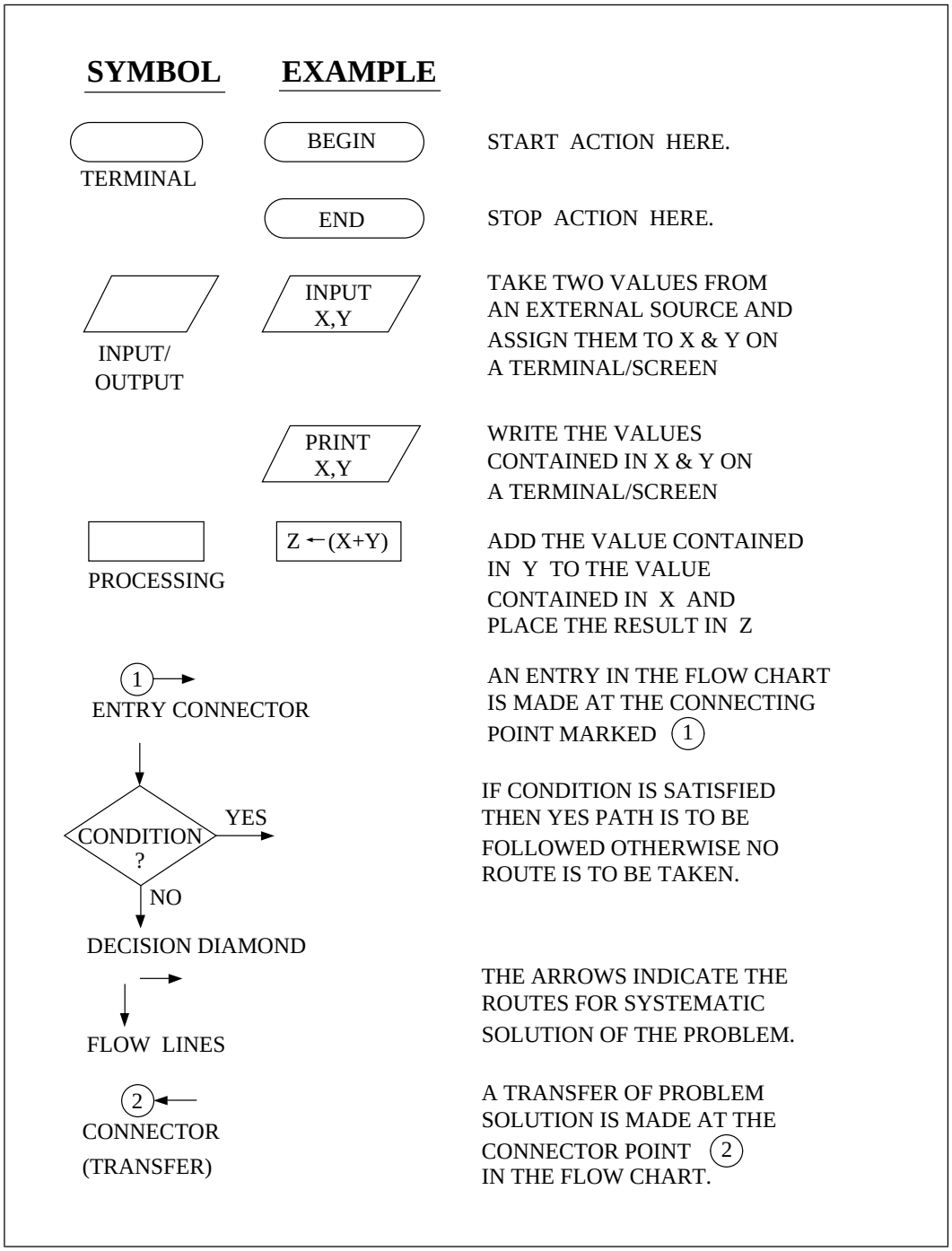

Figure 1.1 Flowchart symbols

processes such as adding, subtracting, multiplying and dividing are shown by a processing symbol. The logical process of moving data from one location of the main memory to another is also denoted by this symbol. When more than one arithmetic and data movement instructions are to be executed consecutively, they are normally placed in the same processing box and they are assumed to be executed in the order of their appearance.

Flow lines

Flowlines with arrowheads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right. Arrowheads are required only when the normal top to bottom flow is not to be followed. However, as a good practice and in order to avoid confusion, flow lines are usually drawn with an arrowhead at the point of entry to a symbol. Good practice also dictates that flow lines should not cross each other and that such intersections should be avoided whenever possible.

Decision

The decision symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternative points is possible. Figure 1.2 shows three different ways in which a decision symbol can be used. It may be noted from these examples that the criterion for making the decision should be indicated clearly within the decision box. Moreover, the condition upon which each of the possible exit paths will be executed, should be identified and all the possible paths should be accounted for. During execution, the appropriate path is followed depending upon the result of the decision.

Connector

If a flowchart becomes very long, the flow lines start *crisscrossing* at many places that causes confusion and reduces the clarity of the flowchart. Moreover, there are instances when a flowchart becomes too long to fit in a single page and the use of flow lines becomes impossible. Thus, whenever a flowchart becomes too complex that the number and direction of flow lines is confusing or it spreads over more than one page, it is useful to utilize the

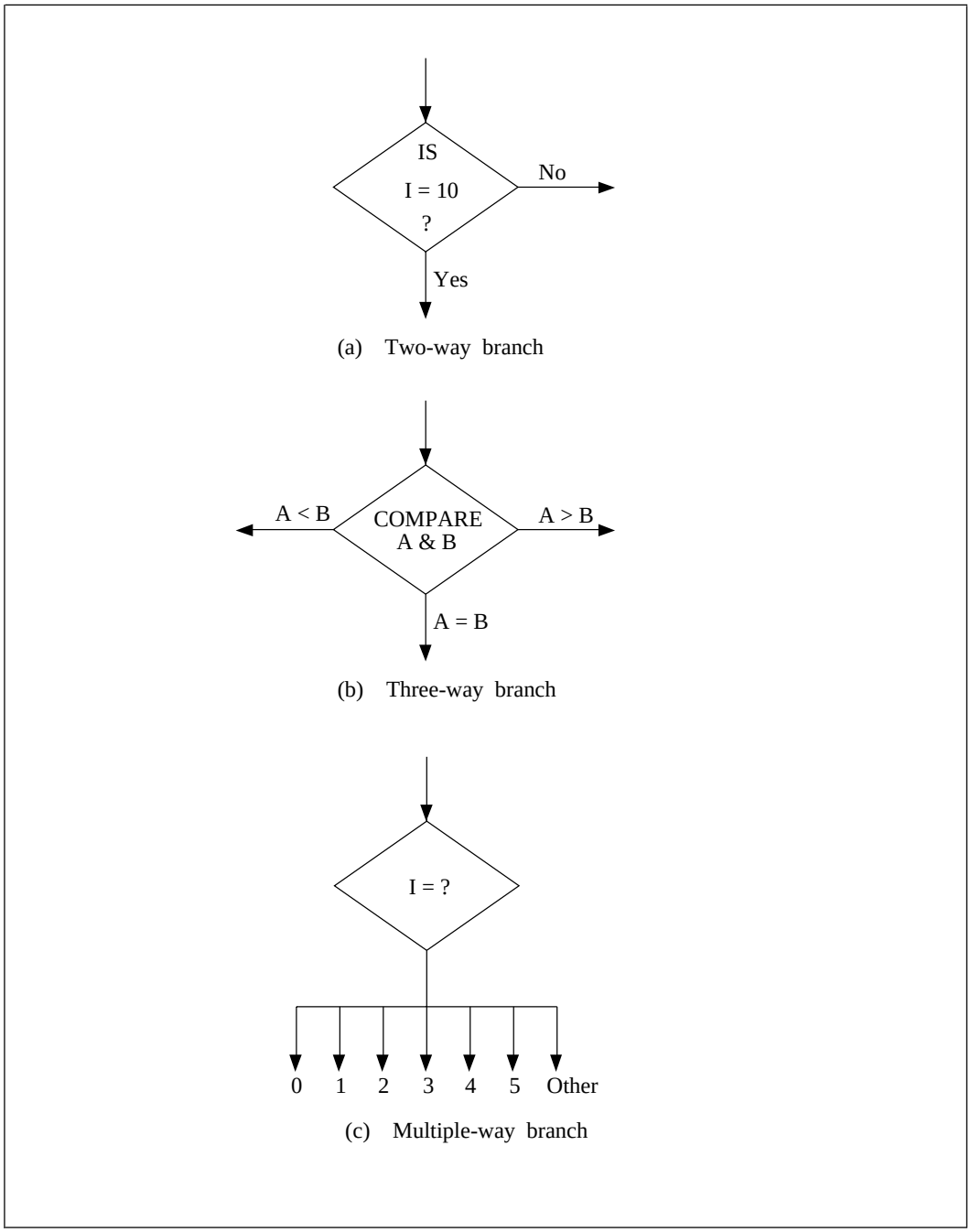

Figure 1.2 Different decision symbols with different ways of branching

connector symbol as a substitute for flow lines. This symbol represents an entry from, or an exit to another part of the flowchart.

A connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link. A pair of identically labeled connector symbols are commonly used to indicate a continued flow when the use of a line is confusing. So two connectors with identical labels serve the same function as a long flow line. That is, they show an exit to some other chart section, or they indicate an entry from another part of the chart. How is it possible to determine if a connector is used as an entry or an exit point? It is very simple: if an arrow enters but does not leave a connector, it is an exit point and program control is transferred to the identically labeled connector that does have an outlet. It may be noted that connectors do not represent any operation and their use in a flowchart is only for the sake of convenience and clarity.

Example 1.1

Draw a flowchart for adding marks in ten subjects obtained by a student in an examination. The output should print the percentage of marks of the student in the examination.

The flowchart for this problem is given in Figure 1.3

The first symbol is a terminal labeled START. It shows that this is the starting point or beginning of our flowchart logic. The second symbol is an input/output (I/O) symbol that is labeled specifically to show that this step is to read input data. This step will input the roll number, name, and the marks obtained by the student from an input device into the main storage of the computer system. The third symbol is a processing symbol which is suitably labeled to indicate that at this step, the computer will add the marks obtained by the student in various subjects and then store the sum in a memory location which has been given the name TOTAL. The fourth symbol is again a processing symbol. The label inside it clearly indicates that the percentage marks obtained by a student is calculated at this stage by dividing TOTAL by 10 and the result is stored in a memory location which has been given the name PERCENTAGE. The fifth symbol is an input/output (I/O) symbol and is labeled WRITE OUTPUT DATA.

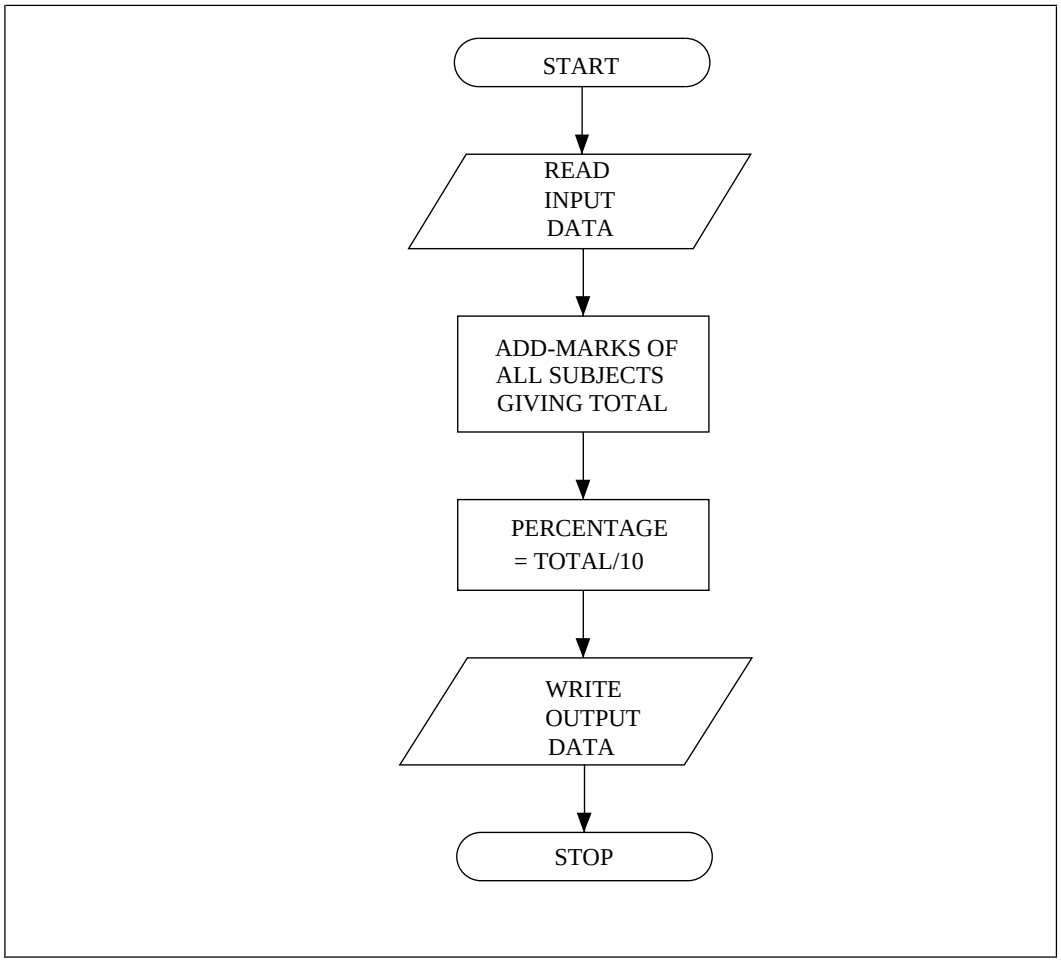

Figure 1.3 Flowchart for calculating percentage of marks

Example 1.2

Draw a flowchart for calculating the average percentage marks of 50 students. Each student appeared in ten subjects. The flowchart should show the counting of the number of student who have appeared in the examination and the calculation should stop when the number of counts reaches the number 50.

Since all the students have appeared in the same examination, so the process of calculation and printing the percentage marks obtained by each student will basically remain the same. The process of reading the input data, adding the marks of all

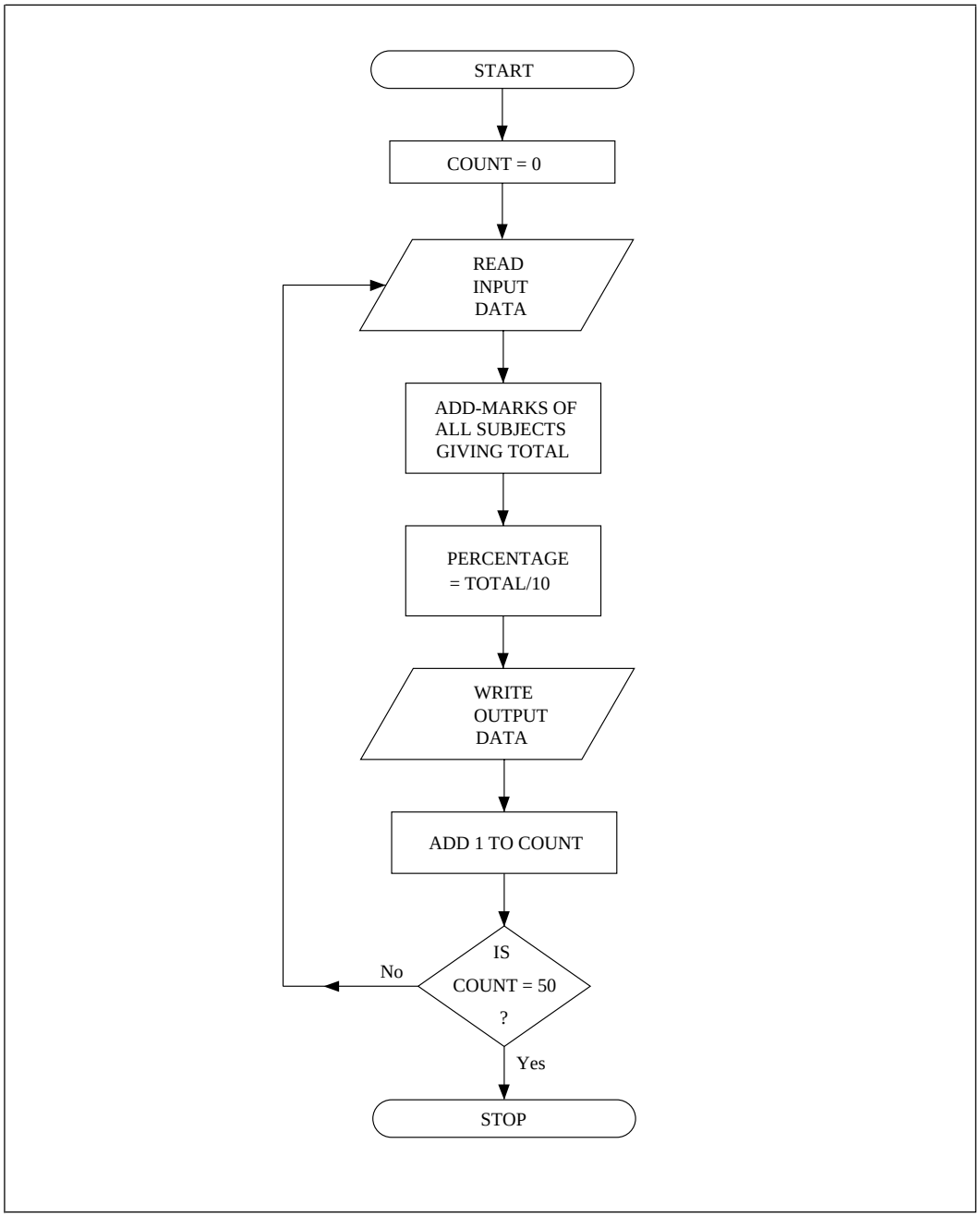

Figure 1.4 Flowchart for example 1.2

subjects, calculating the percentage, and then writing the output data, is to be repeated for all the 50 students. In this situation where the same logical steps can be repeated, the flow line symbols are used in a flowchart to indicate the repetitive nature of the logic in the form of a loop.

Figure 1.4 shows a flowchart which uses a decision step to terminate the algorithm. In this flowchart, another variable COUNT has been introduced which is initialized to zero outside the process loop and is incremented by 1 after processing the data for each student. Thus, the value of COUNT will always be equal to the number of students whose data has already been processed. At the decision step, the value of COUNT is compared with 50 which is the total number of students who have appeared for the examination. The steps within the process loop are repeated until the value of COUNT becomes equal to 50. As soon as the value of COUNT becomes equal to 50, the instruction at the decision step causes the control to flow out of the loop and the processing stops because a terminal symbol labeled STOP is encountered.

Designing a General Flowchart

The flowchart shown in Figure 1.4 is not a general flowchart for the Example 1.2 for calculating the percentage of marks of any number of students appearing in the examination. A good program should be general in nature. For example, in this case we should write a program that need not be modified every time, even if the total number of students changes.

To overcome these drawbacks, another method can be adopted to control the loop. In this method, the end of input data is marked by a trailer record, that is, the last data record in the input is followed by a record whose main purpose is to indicate that the end of the input data has been reached. Suppose the first 7 characters of the input record of a student represents his roll number (ROLLNO). Since 0000000 is never used as a roll number, a value of 0000000 as the first 7 characters can be used to represent the *trailer* record. As each input record is processed, the ROLLNO can be compared with 0000000 to determine if processing is complete. The logic of this process is illustrated in the flowchart given in Figure 1.5.

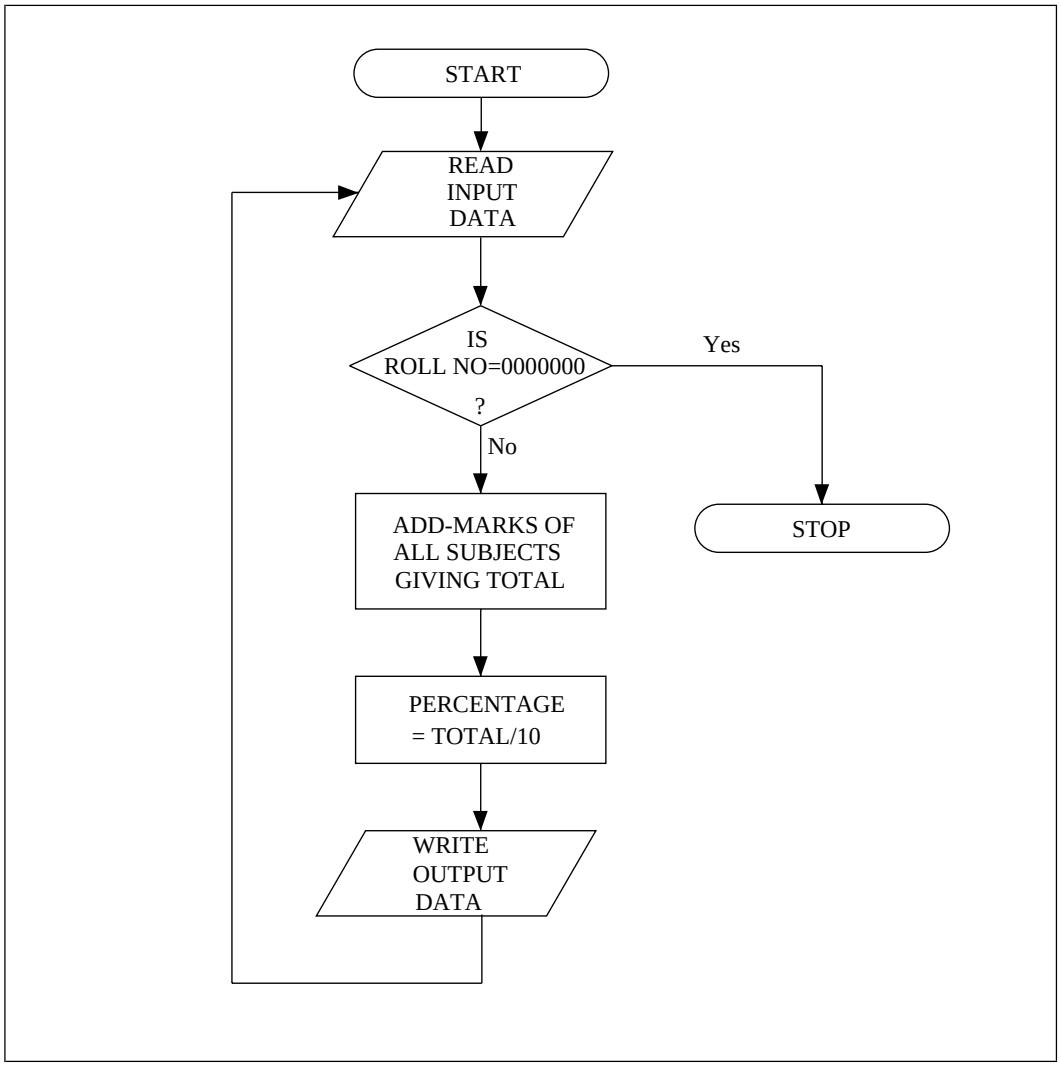

Figure 1.5 General flowchart for example 1.2

What is Trailer Record?

The concept of a trailer record centers around the notion of selecting a field (a particular item of data) in the input record which will be used to indicate the end of data, and then selecting a trailer value also known as *sentinel* value which will never occur as normal data value for that field. The roll number 0000000 is a trailer record for Example 1.2.

Example 1.3

Extend the flowchart represented in Figure 1.5 to count the number of students who have scored percentage marks more than 30. The flowchart should give the result of the students scoring the percentage marks equal to or more than 30.

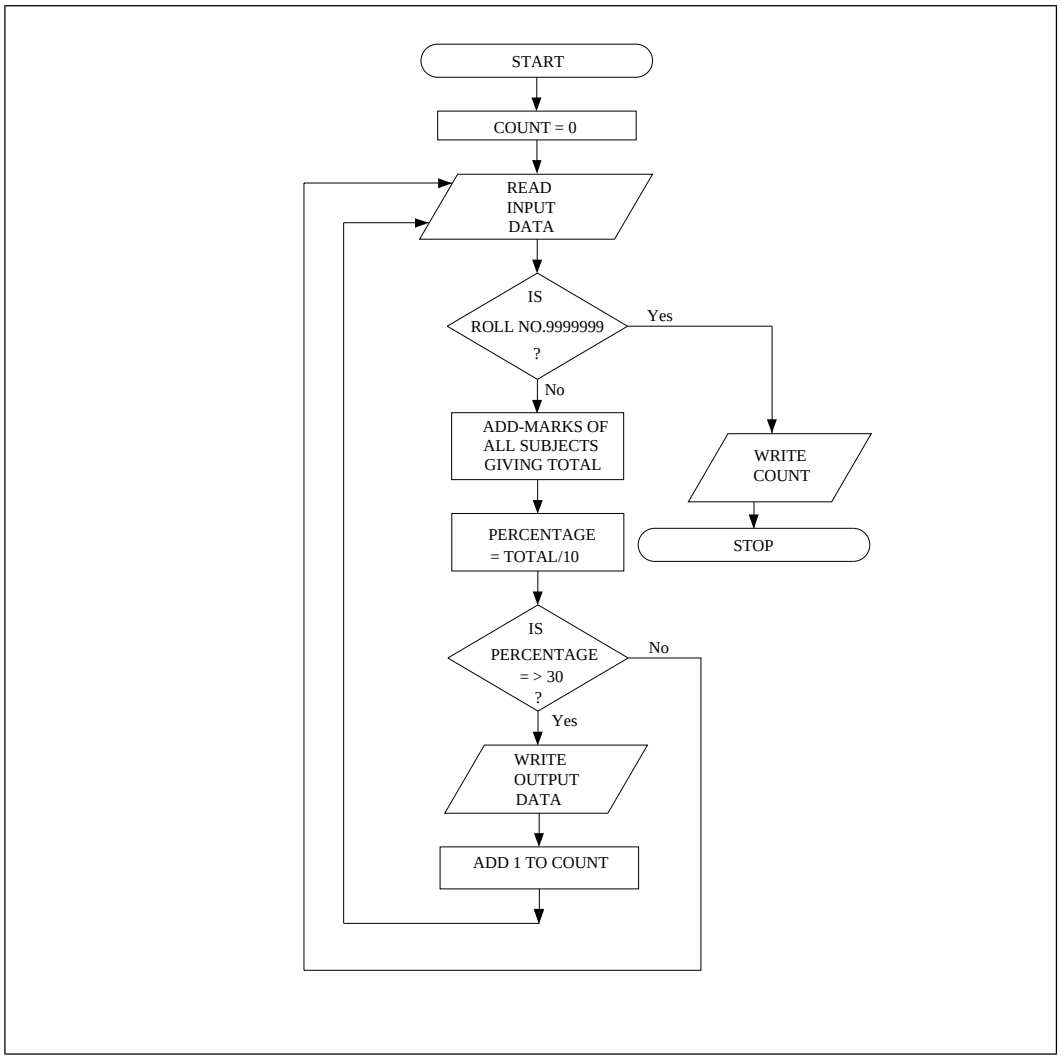

Figure 1.6 Flowchart for example 1.3

The flowchart in Figure 1.6 is a solution to this problem. There are two decision symbols in this flowchart. The first decision symbol checks for a trailer record by comparing ROLLNO against the value 9999999 to determine if processing is complete. The second decision symbol is used to check whether the student has passed or failed by comparing the percentage marks obtained by him against 30. If the student's PERCENTAGE is equal to or more than 30, then he has passed, otherwise he has failed. Note from the flowchart that the operation WRITE OUTPUT DATA is performed only if the student has passed. If he has failed, we directly perform the operation READ INPUT DATA without performing the WRITE operation. This ensures that the output list provided by the computer will contain the details of only those students who have passed in the examination.

Another point to be noted in this flowchart is the use of variable COUNT. This variable has been initialized to zero in the beginning and is increased by 1 every time the operation WRITE OUTPUT DATA is performed. But we have seen that the operation WRITE OUTPUT DATA is performed only for the students who have passed. Hence, the variable COUNT will be increased by 1 only in the case of those students who have passed. Thus, the value of COUNT will always be equal to the number of students whose data has already been processed and who have been identified as passed. Finally, when the trailer record is detected, the operation WRITE COUNT will print out the final value of COUNT that will be equal to the total number of students who have passed the examination.

1.1.3 Flowcharting Rules

(a) First formulate the main line of logic, then incorporate the details.
(b) Maintain a consistent level of detail for a given flowchart.
(c) Do not give every detail on the flowchart. A reader who is interested in greater details can refer to the program itself.
(d) Words in the flowchart symbols should be common statements and easy to understand.
(e) Be consistent in using names and variables in the flowchart.
(f) Go from left to right and top to bottom in constructing the flowchart.

(g) Keep the flowchart as simple as possible. The crossing of flow lines should be avoided as far as possible.
(h) If a new flowcharting page is needed, it is recommended that the flowchart be broken at an input or output point. Moreover, properly labeled connectors should be used to link the portions of the flowchart on different pages.

1.1.4 Advantages of Flowcharts

Conveys Better Meaning

Since a flowchart is a pictorial representation of a program, it is easier for a programmer to understand and explain the logic of the program to some other programmer.

Analyses the Problem Effectively

A macro flowchart that charts the main line of logic of a software system becomes a system model that can be broken down into detailed parts for study and further analysis of the system.

Effective Joining of a Part of a System

A group of programmers are normally associated with the design of large software systems. Each programmer is responsible for designing only a part of the entire system. So initially, if each programmer draws a flowchart for his part of design, the flowcharts of all the programmers can be placed together to visualize the overall system design. Any problem in linking the various parts of the system can be easily detected at this stage and the design can be accordingly modified. Flowcharts can thus be used as working models in the design of new programs and software systems.

Efficient Coding

Once a flowchart is ready, programmers find it very easy to write the concerned program because the flowchart acts as a roadmap for them. It guides them in proceeding from the starting point of the program to the final point ensuring that no steps are omitted. The ultimate result is an error free program developed at a faster rate.

Systematic Debugging

Even after taking full care in program design, some errors may remain in the program because the designer might have never thought about a particular case. These errors are detected only when we start executing the program on a computer. Such type of program errors are called *bugs* and the process of removing these errors is known as *debugging*. A flowchart is very helpful in detecting, locating, and removing mistakes (bugs) in a program in a systematic manner.

Systematic Testing

Testing is the process of confirming whether a program will successfully do all the jobs for which it has been designed under the specified constraints. For testing a program, different sets of data are fed as input to that program to test the different paths in the program logic.

1.1.5 Limitations of Flowcharts

Takes More Time to Draw

Flowcharts are very time consuming and laborious to draw with proper symbols and spacing, especially for large complex programs.

Difficult to Make Changes

Owing to the symbol-string nature of flowcharting, any changes or modifications in the program logic will usually require a completely new flowchart. Redrawing a flowchart is tedious and many companies either do not change them or produce the flowchart by using a computer program to draw it.

Non-standardization

There are no standards determining the amount of detail that should be included in a flowchart.

1.1.6 Levels of Flowcharts

A flowchart that outlines the main segments of a program and shows lesser detail is a *macro-flowchart*. On the other hand, a

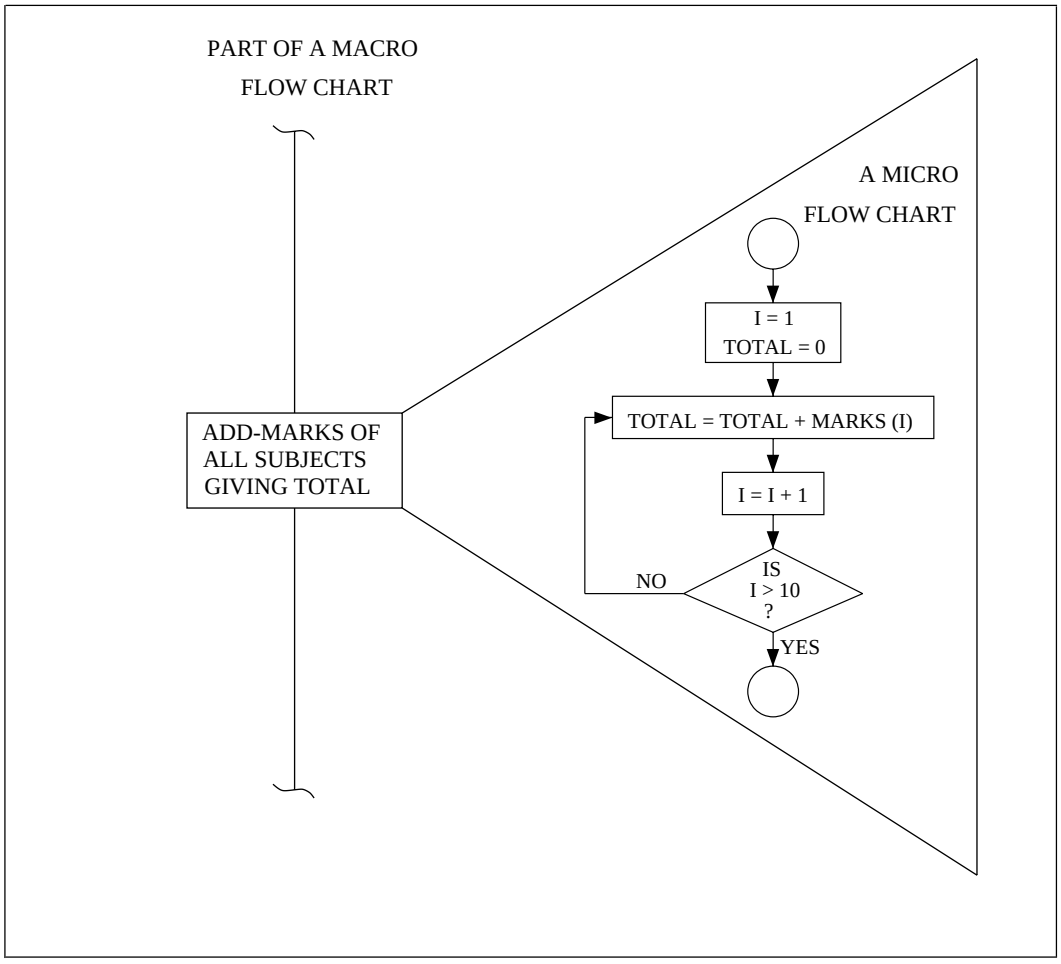

Figure 1.7 Details of the processing block of Figure 1.3 for adding marks in 10 subjects

flowchart with more detail is a *micro-flowchart*. For example, let us consider the examination problem that we have already discussed in Examples 1.3 and 1.4. In all the flowcharts of the examination problem, there is a processing box having the instruction "ADD MARKS OF ALL SUBJECTS GIVING TOTAL". In order to display how the values of TOTAL is calculated, a detailed flowchart can be drawn as shown in Figure 1.7. In a similar manner, the input/output (I/O) boxes for the READ and WRITE operations can also be converted to a detailed flowchart.

1.2 Pseudocode or Metacode or PDL

Pseudocode is another program analysis tool that is used for planning program logic. "Pseudo" means imitation or false and "Code" refers to the instructions written in a programming language. Pseudocode, therefore, is an imitation of actual computer instructions. These pseudo instructions are phrases written in ordinary natural language (e.g., English, French, German, etc.). Instead of using symbols to describe the logic steps of a program, as in flowcharting, pseudocode uses a structure that resembles computer instructions. Because it emphasises the design of the program, pseudocode is also called *Program Design Language (PDL).*

Pseudocode is made up of the following basic logic structures that have been proved to be sufficient for writing any computer program :

1. Sequence
2. Selection (IF...THEN...ELSE
   or IF....THEN)
1. Iteration (DO...WHILE or REPEAT...UNTIL)

1.2.1 Sequence

Sequence logic is used for performing instructions one after another in sequence. Thus, for sequence logic, pseudocode instructions are written in the order, or sequence, in which they are to be performed. The logic flow of pseudocode is from the top to the bottom. Figure 1.8 shows an example of sequence logic structure.

1.2.2 Selection

Selection logic, also known as decision logic, is used for making decisions. It is used for selecting the proper path out of the two or more alternative paths in the program logic. Selection logic is depicted as either an IF...THEN...ELSE or IF.....THEN structure. The flowcharts shown in Figures 1.9 and 1.10 illustrate the logic of these structures. Their corresponding pseudocode is also given in these figures.

The IF...THEN...ELSE construct says that if the condition is true, then do process 1, else (if the condition is not true) do process 2. Thus, in this case either process 1 or process 2 will be executed depending on whether the specified condition is true or false. However, if we do not want to choose between two processes and we simply want to decide if a process is to be performed or not, then the IF...THEN structure is used.

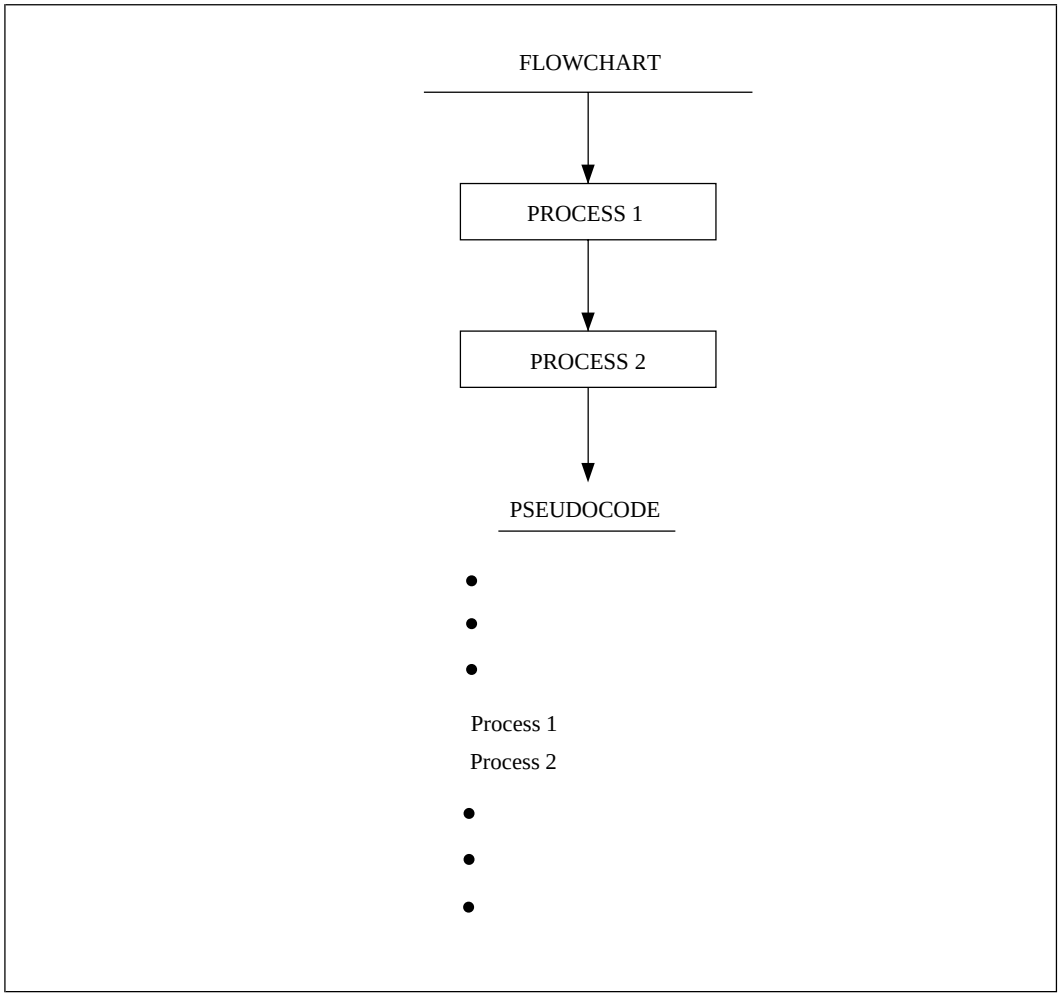

Figure 1.8 Pseudocode for sequence structure

The IF...THEN structure says that if the condition is true, then do process 1; and if it is not true, then skip over process 1. In both the structures, process 1 and process 2 can actually be one or more processes. They are not limited to a single process. END IF is used to indicate the end of the decision structures.

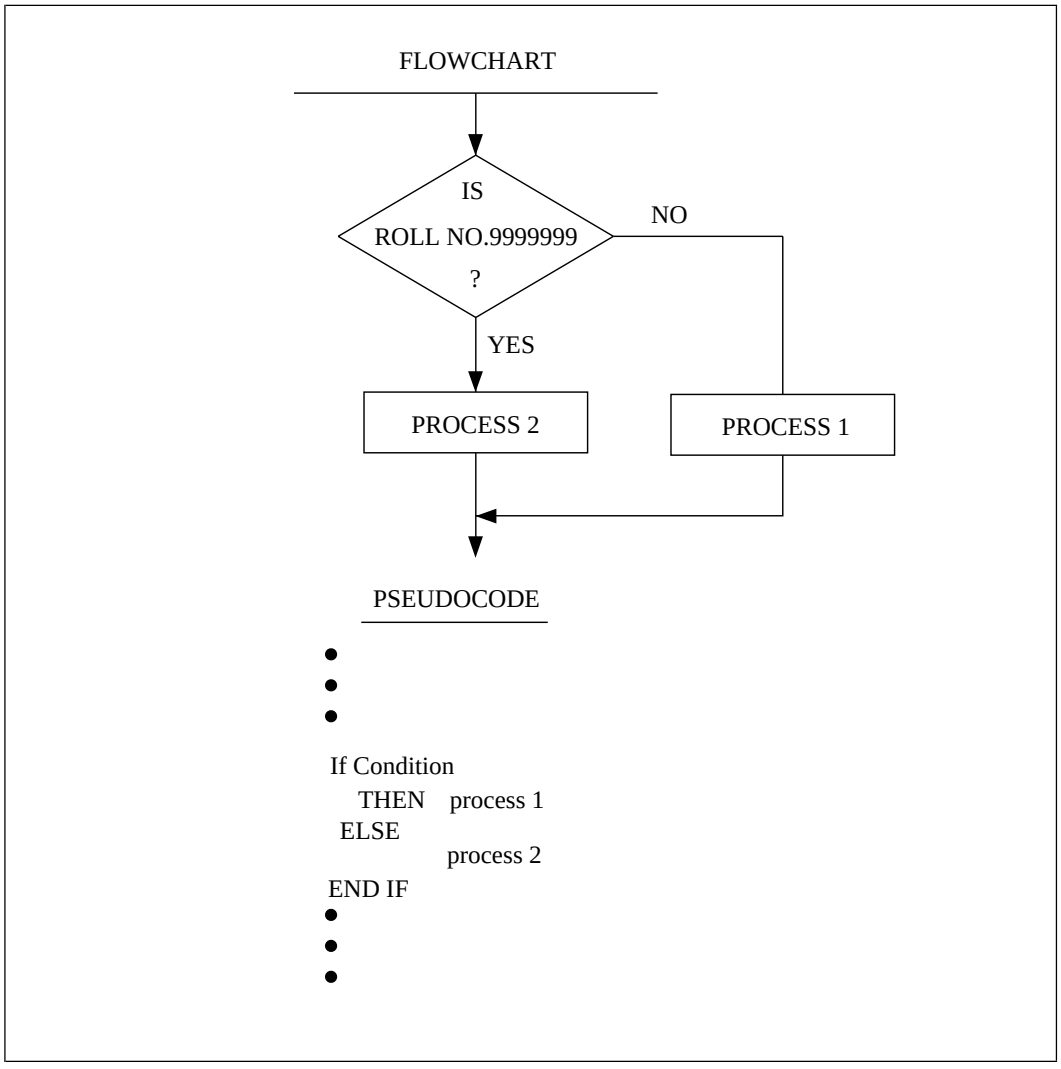

Figure 1.9 Pseudocode for If-Then-Else structure

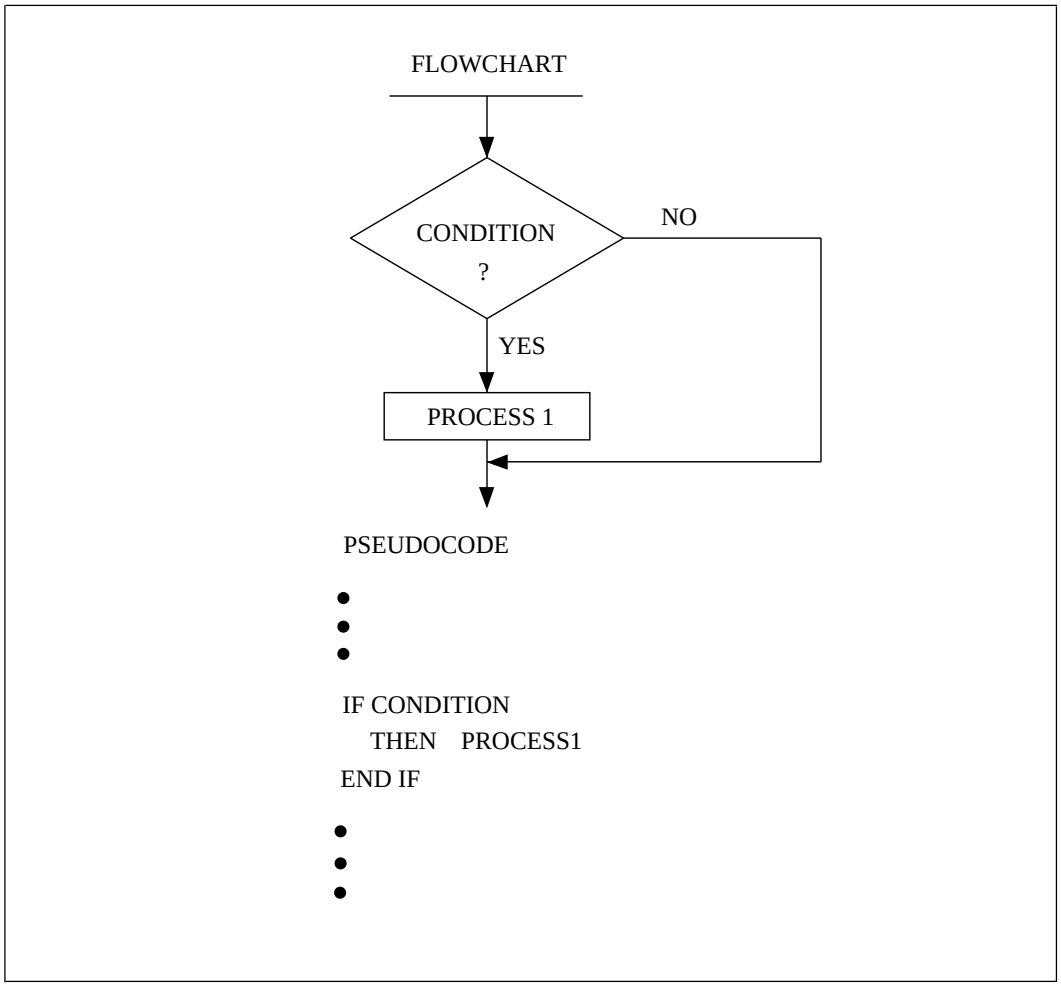

Figure 1.10 Pseudocode for If-Then selection structure

1.2.3 Iteration Logic

Iteration logic is used when one or more instructions may be executed several times depending on some condition. It uses two structures called the DO...WHILE and the REPEAT...UNTIL. They are illustrated by flowcharts in Figure 1.11 and Figure 1.12 respectively. Their corresponding pseudocodes are also given in these figures. Both DO...WHILE and REPEAT...UNTIL are used for looping.

1.2.4 Differences between Do..While and Repeat.. Until loops

The differences between the two loops are that in the

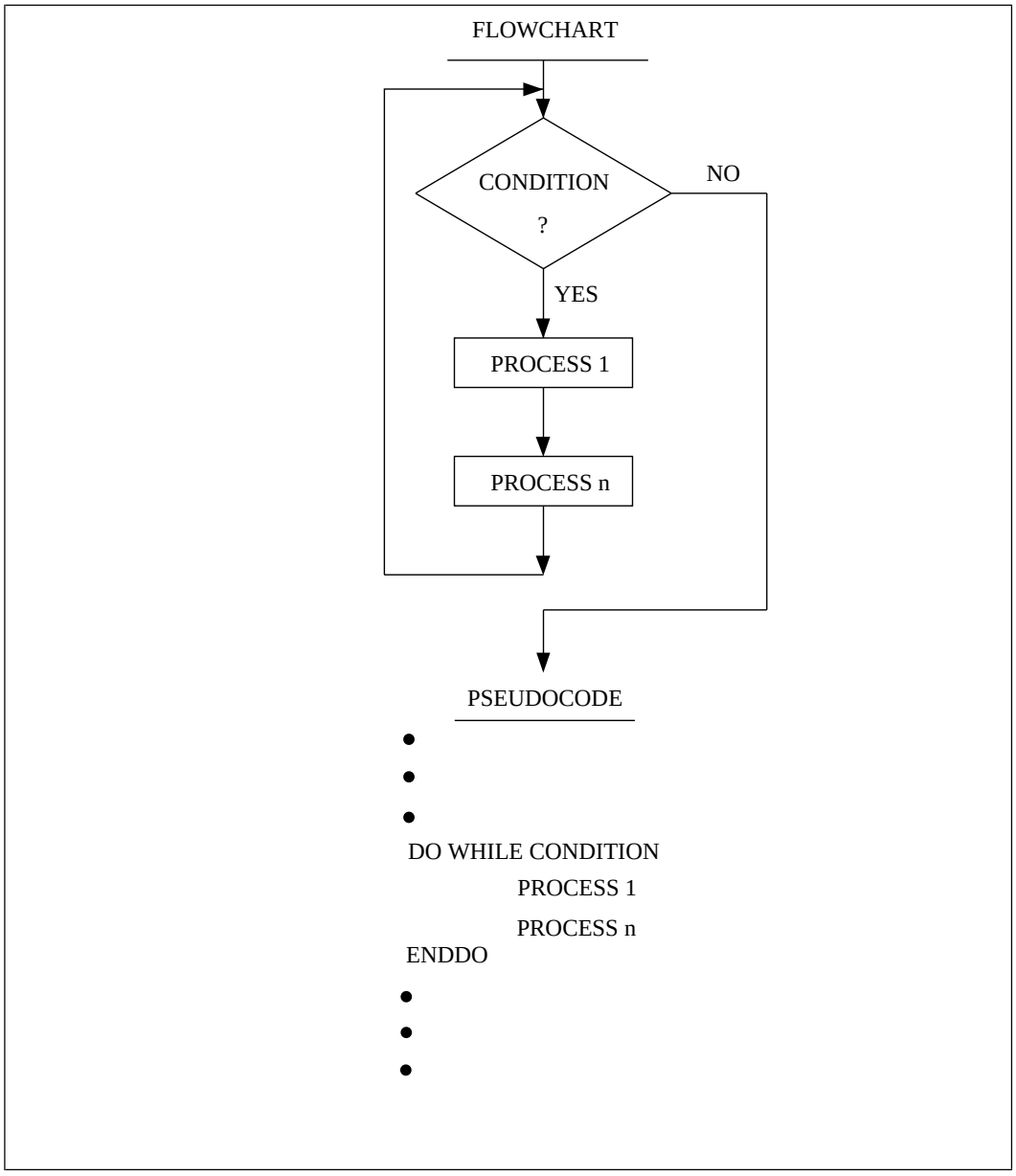

Figure 1.11 Pseudo-code for Do.. While structure

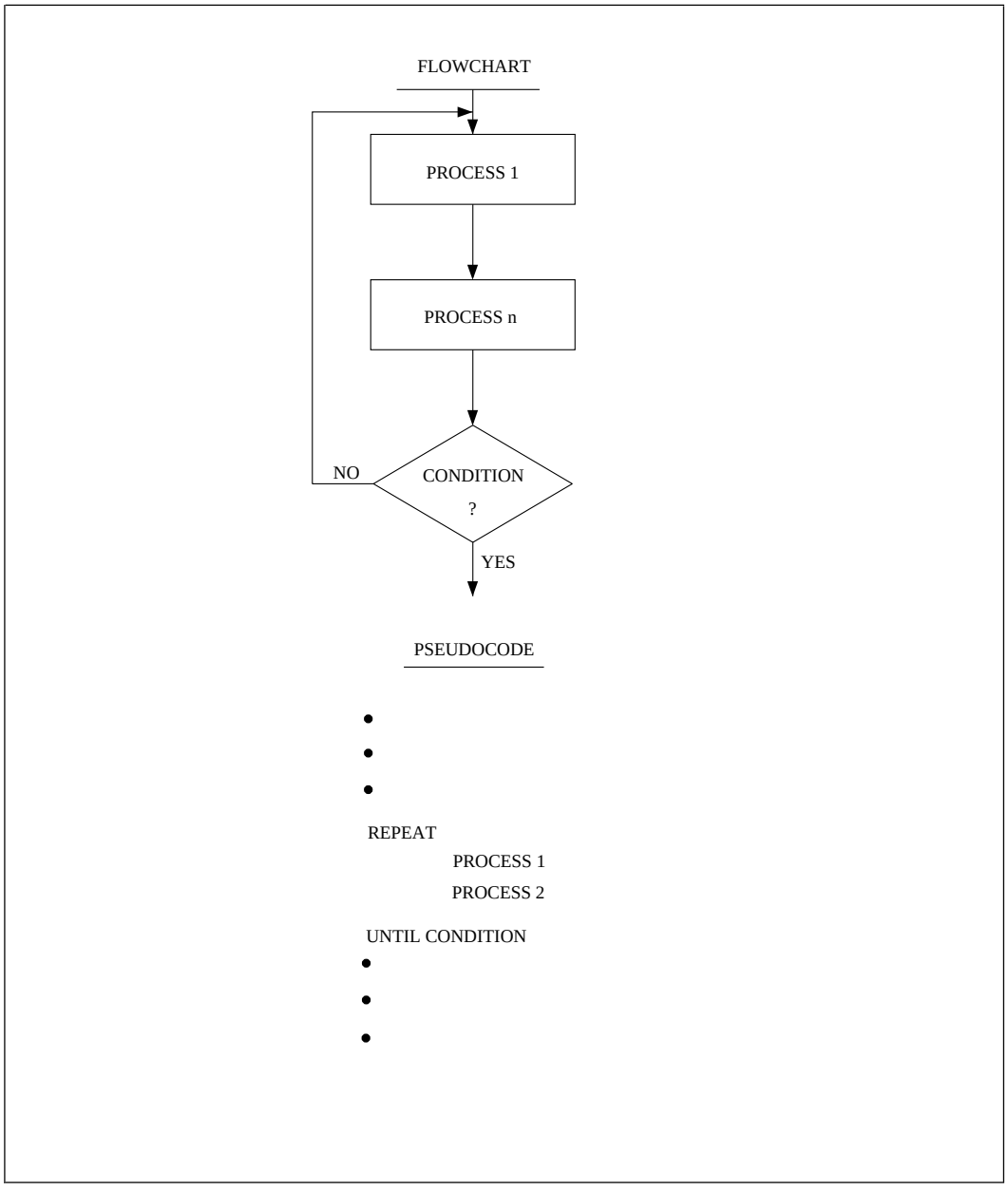

Figure 1.12 Pseudo-code for Repeat..Until structure

DO...WHILE, the looping will continue as long as the condition is true. The looping stops when the condition is not true. On the other hand, in case of REPEAT...UNTIL, the looping continues

until the condition becomes true. That is, the execution of the statements within the loop is repeated as long as the condition is not true. In both the DO...WHILE and REPEAT...UNTIL, the loop must contain a statement that will change the condition that controls the loop. Also the condition is tested at the top of the loop in the DO...WHILE loop structure and at the bottom of the loop in the REPEAT...UNTIL structure. The "ENDDO" marks the end of a DO...WHILE structure and UNTIL followed by some condition marks the end of the REPEAT...UNTIL structure.

Example 1.4

The following pseudocode describes the policy of a company to award the bonus to an employee.

```
Input
Employee number, pay, position code & years.
IF
    position code = 1
THEN
    set bonus to 1 week's pay
ELSE
    IF position code = 2
    THEN
      IF 2 weeks pay > 700
      THEN
        set bonus to 700
      ELSE
        set bonus to 2 week's pay
      END IF
    ELSE
        set Bonus to 1.5 week's pay
    END IF
END IF
IF year greater then 10
THEN
    Add 100 to bonus
ELSE
    IF years less than 2
    THEN
        cut bonus to half
```

```
    ELSE
        bonus stays the same
    END IF
Print employee number & bonus
```

Example 1.5

This example illustrates the policy of a bank or a financial institution for giving a loan to an individual.

```
Input mortgage amount
IF amount < 25,000
THEN
    down payment = 3% of amount
ELSE
     payment1 = 3% of 25,000
     payment2 = 5% of (amount - 25,000)
     down payment = payment1 + payment2
END IF
print down payment
```

Example 1.6

The following program in pseudocode form illustrates the policy of a company to give the commission to a sales person.

```
Input sales
IF sales < 500
THEN
    Commission = 2% of sales.
ELSE
    IF sales < 5000
    THEN
    Commission = 5% of sales
    ELSE
    Commission = 10% of sales
END IF
Print Commission
```

1.2.5 Advantages of Pseudocodes

Pseudocode has three main advantages:

(a) Converting a pseudocode to a programming language is much more easier as compared to converting a flowchart or a decision table.
(b) As compared to a flowchart, it is easier to modify the pseudocode of a program logic when program modifications are necessary.
(c) Writing of pseudocode involves much less time and effort than drawing an equivalent flowchart. Pseudocode is easier to write than an actual programming language because it has only a few rules to follow, allowing the programmer to concentrate on the logic of the program.

1.2.6 Limitations of Pseudocodes

(a) In case of pseudocode, a graphic representation of program logic is not available.
(b) There are no standard rules to follow in using pseudocode. Different programmers use their own style of writing pseudocode and hence communication problems occur due to lack of standardization.
(c) For a beginner, it is more difficult to follow the logic or write the pseudocode, as compared to flowcharting.

1.2.6 Limitations of Pseudocodes

(a) In case of pseudocode, a graphic representation of program logic is not available.
(b) There are no standard rules to follow in using pseudocode. Different programmers use their own style of writing pseudocode and hence communication problems occur due to lack of standardization.
(c) For a beginner, it is more difficult to follow the logic or write the pseudocode, as compared to flowcharting.

1.3 Programming Techniques

You can solve different types of business or scientific problems on a computer. All you have to do is to analyse the problem, write a step by step solution and translate it into a language understandable by a computer.

1.3.1 Problem Definition

Let us take an example of a problem faced in day- to- day life. Suppose you want to reach your school computer laboratory at 8 AM. You would lay out a plan to get ready by 7 AM, then take a bus/rikshaw and reach at the gate of your school. Then climb up the stairs to arrive in the laboratory. For all these movements, you will note the time taken for each part. If due to some reason, you are unable to get ready by 7 AM and you already know that it takes one hour to reach from house to school laboratory by bus/rikshaw, then you will take a faster means of transportation. You may take an autorikshaw or a taxi. Thus, a very simple problem of reaching the computer laboratory of your school by 8 AM will need several steps for solution. Each step is to be accurately defined/marked so that no guess work is necessary. You can thus represent the solution of this problem in three steps shown in Figure 1.11.

In Figure 1.13, steps 1, 2 and 3 appear to be very simple. In actual practice when you have to give instructions to a student, who is going for the first time, it may not be so easy. For example, you have to define the word "READY" precisely so that he knows exactly what he has to do by 7AM to get READY. Similarly in step 2, you may have to clearly specify the bus route number, the bus stop to board the bus, the place to get down from the bus etc. You may also like to tell that a bus is not to be boarded, if it is overcrowded. The word "overcrowded" needs to be exactly defined. Finally, step 3 needs further elaboration about the room number and floor number; where the computer laboratory is located and how to reach there.

You can explain all this to a person who does not know any thing about the computer laboratory, provided you know it correctly. In the same way, you can solve a problem on a computer, if you know exactly what it is and how to solve it manually.
We shall solve a few problems using flowchart.

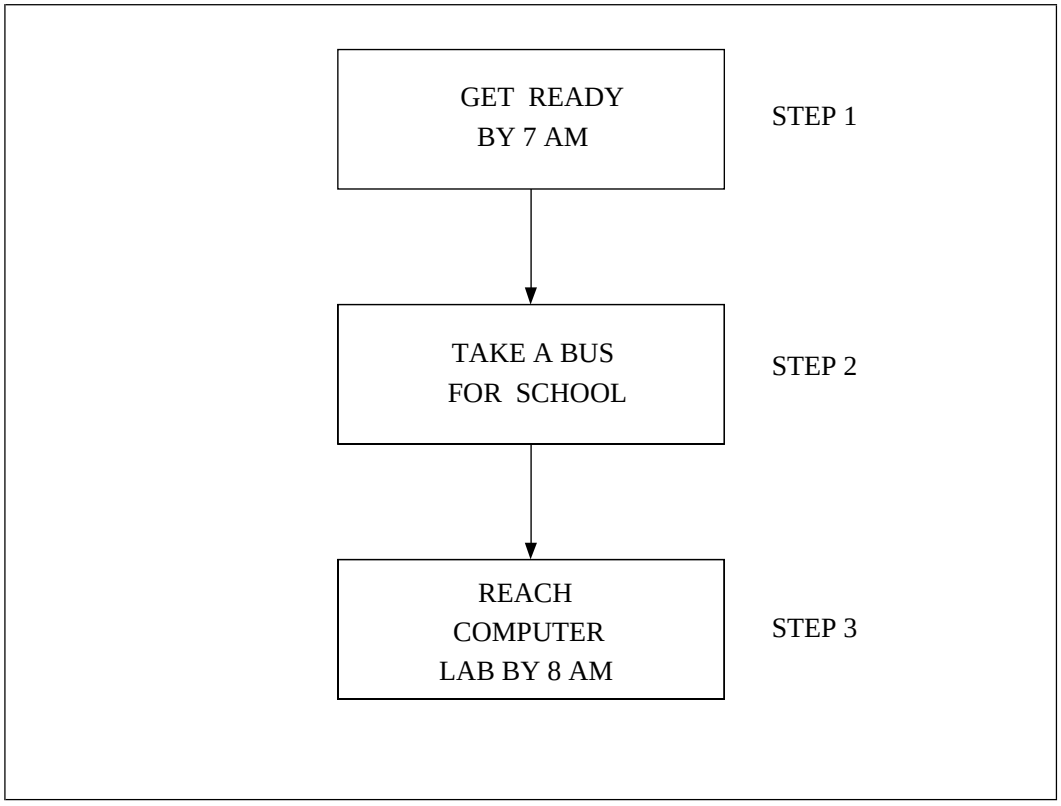

Figure 1.13 Step by step solution to reach school computer laboratory (LAB)

Example 1.7

Consider the flowchart shown in Figure 1.14. Give the value of BONUS under the following conditions:

(a) X = 20, Y = 10

(b) X = 10, Y = 20

Solution

(a) The value of X and Y are 20 and 10 respectively. These values are inputted in step 2. The value of BONUS is taken as 500 in step 1. Since the values of X = 20 and Y = 10, so in step 4, X is more than Y.

Therefore the *Yes* route is to be followed. Hence the new value of BONUS is the old value of BONUS + 100 i.e. 600. Hence the value of BONUS written in step 5 is 600.

(b) Here, the value of X=10 & Y=20. Therefore in step 4, the diamond decision shows that X is less than Y. Therefore, the *No* route is followed. The value of BONUS remains 500 only. Hence the result is 500.

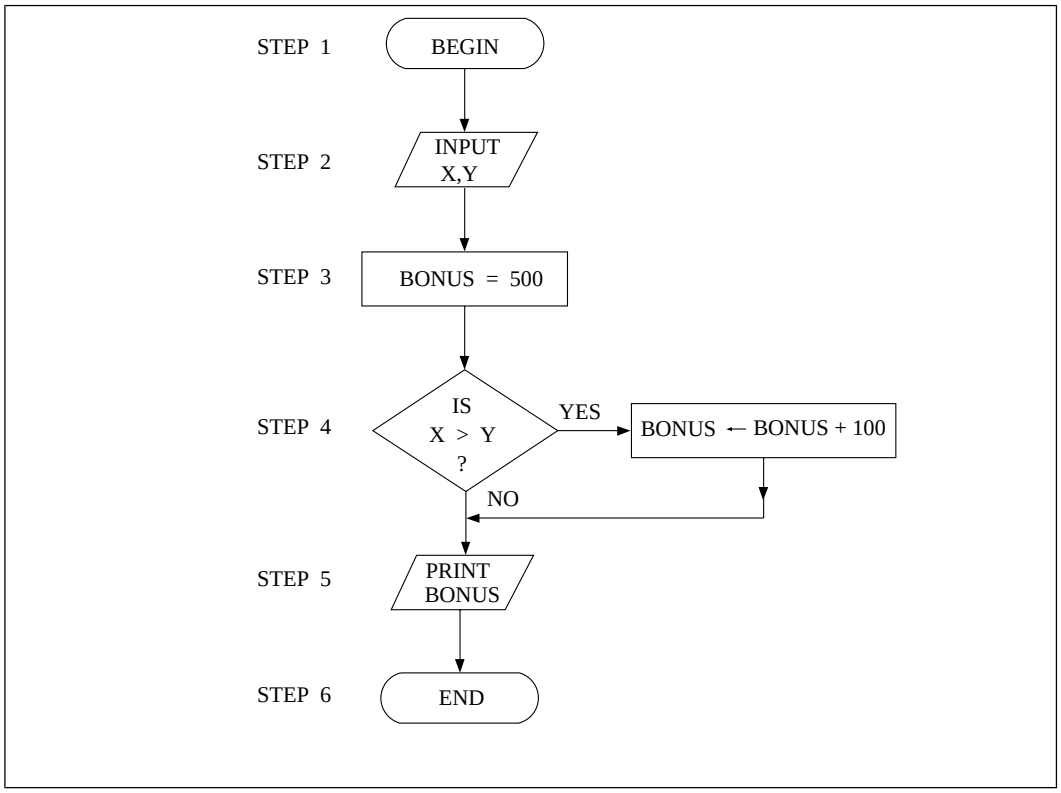

Figure 1.14 Bonus calculation

Example 1.8

Here are six types of values for the variables A, B, and C.

(a) A=9, B=5, C=1 (b) A=7, B=3, C=9

(c) A=2, B=9, C=7 (d) A=4, B=8, C=2

(e) A=6, B=3, C=8 (f) A=2, B=3, C=6

1. Using each type of data, with the flowchart shown in Figure 1.15, state the output in each case.

2. What does the procedure do?
1. What happens, if all the values of A, B and C are the same?

Solution

1.(a) The values of A, B and C are accepted as 9, 5 and 1 respectively. In step 3, the value of A is compared with B. Since 9 is greater than 5, therefore the YES path is followed. Once again, in step 4, the value of A is compared with C. Here 9 is greater than 1. So the YES path is followed. In step 5, value of 'A' is printed which is 9. Hence the answer is 9.

(b) The values of A, B and C are accepted as 7, 3 and 9 respectively. In step 3 the value of A is compared with B. Since 7 is greater than 3, therefore, the YES path is followed. In step 4, the value of A is compared with C. Since 7 is less than 9, therefore the path corresponding to NO is followed. Hence the value of C which is 9 will be printed.

(c) The values of A, B and C are accepted as 2, 9, and 7 respectively. As understood from steps (a) and (b) above, we find that in step 5 the value of B will be printed as 9.

(d) The values of A, B and C are accepted as 4, 8 and 2 respectively. In step 5, the value of B will be printed as 8.

(e) The values of A, B and C are accepted as 6, 3 and 8 respectively. The value of C which is 8 will be printed in step 5.

(f) The values of A, B and C are accepted as 2, 3 and 6 respectively. The value of C which is 6 is printed in step 5.

2. It is evident from the flowchart that the values of A, B and C are compared with each other and the one that contains the highest numeral value is printed. If A has the highest value, then A is printed, otherwise, if B has the highest value, then B is printed or finally the value of C is printed if it has the highest value.

1. If A, B and C have equal values, then the value of C which is the same as that of B or A is printed.

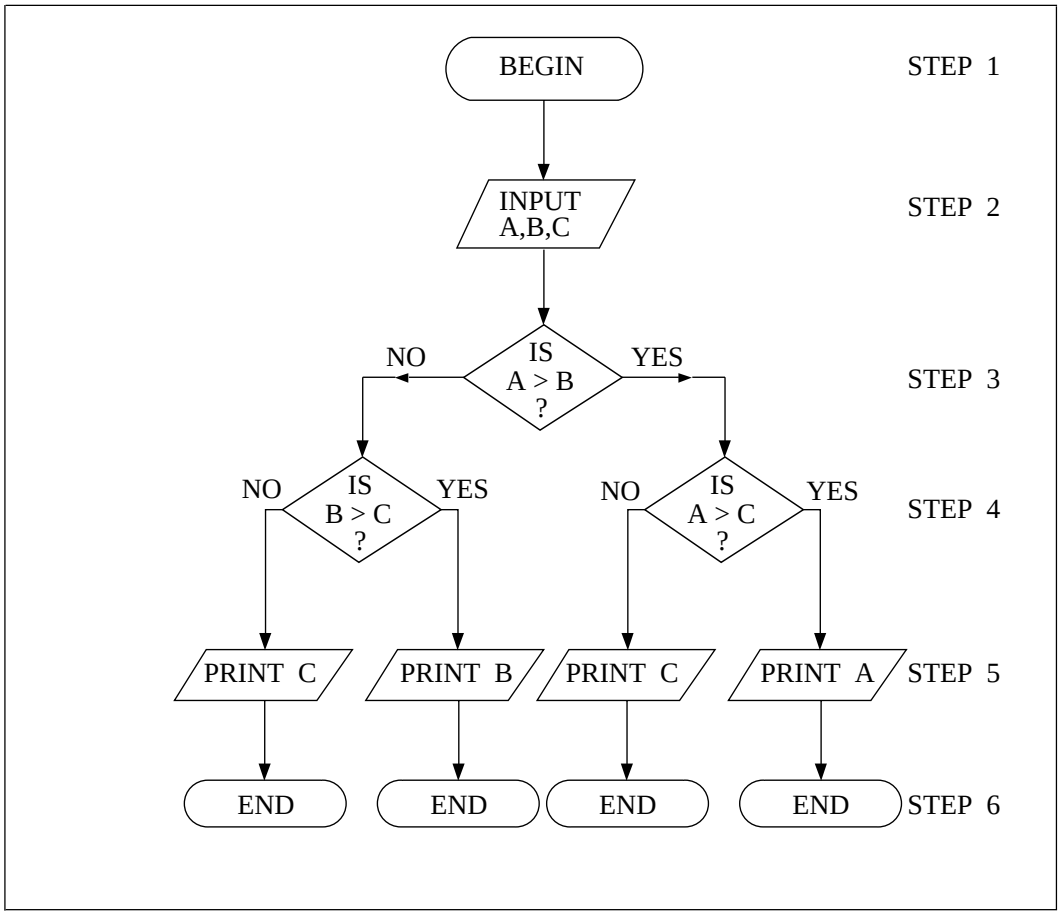

Figure 1.15 Calculation of output for different values of A, B and C

Example 1.9

The flowchart in Figure 1.16 uses connector symbols labelled as 1, 2 and 3. Find the output if the input is:

(a) A=10, B=15,
(b) A=20, B=20,
(c) A=4, B=1,
(d) A=10, B=5.

Solution

(a) In step 2 of Figure 1.16, the values of A and B are taken to be 10 and 15 respectively. In step 3, the value of A is not less than 10, so the path corresponding to NO is followed.

In step 4, the value of B is not less than 20, hence a path corresponding to NO is followed. In step 5, the value of (A+B) is assigned to C, i.e. C becomes (10+15) or equal to 25. Finally, in step 6, the value of C which is 25 is printed.

(b) In step 2, the values of A and B are taken as 20 and 20 respectively. Steps 3 to 5 are similar to that followed in part (a) above. The value of C is now equal to (A+B) which is 40. Hence 40 is printed.

(c) In step 2, the values of A and B are taken as 4 and 1 respectively. In step 3, the value of A is less than 10. Hence the YES path is followed. This is connected to the adjacent figure via the connector 1. Hence the value of A is now the old value of A + 3, i.e. 7.

Further, the decision diamond puts the question "Is A less than 10?" The answer is YES. Hence the control is transferred to the lower figure via connector 2. The new value of B becomes the old value of B multiplied by 5 which is 1 x 3 = 1. The diamond compares the value of B with 20. Since B is less than 20, so the control is transferred back at 2 in this block. The new value of B becomes the old value of B multiplied by 3.

$$= 3 \times 3$$
$$= 9$$

This new value of B i.e. 9 is again compared in the decision diamond. Since 9 is less than 20, the control goes to connecter 2. B is again multiplied and the current value of B is $3 \times 9 = 27$. As 27 is not less than 20, so the NO path is followed.

The value of C = current value of B + 3
= 27 + 3
= 30

The value 30 of C is now transferred to step 6 i.e. PRINT C. Hence the value 30 is printed.

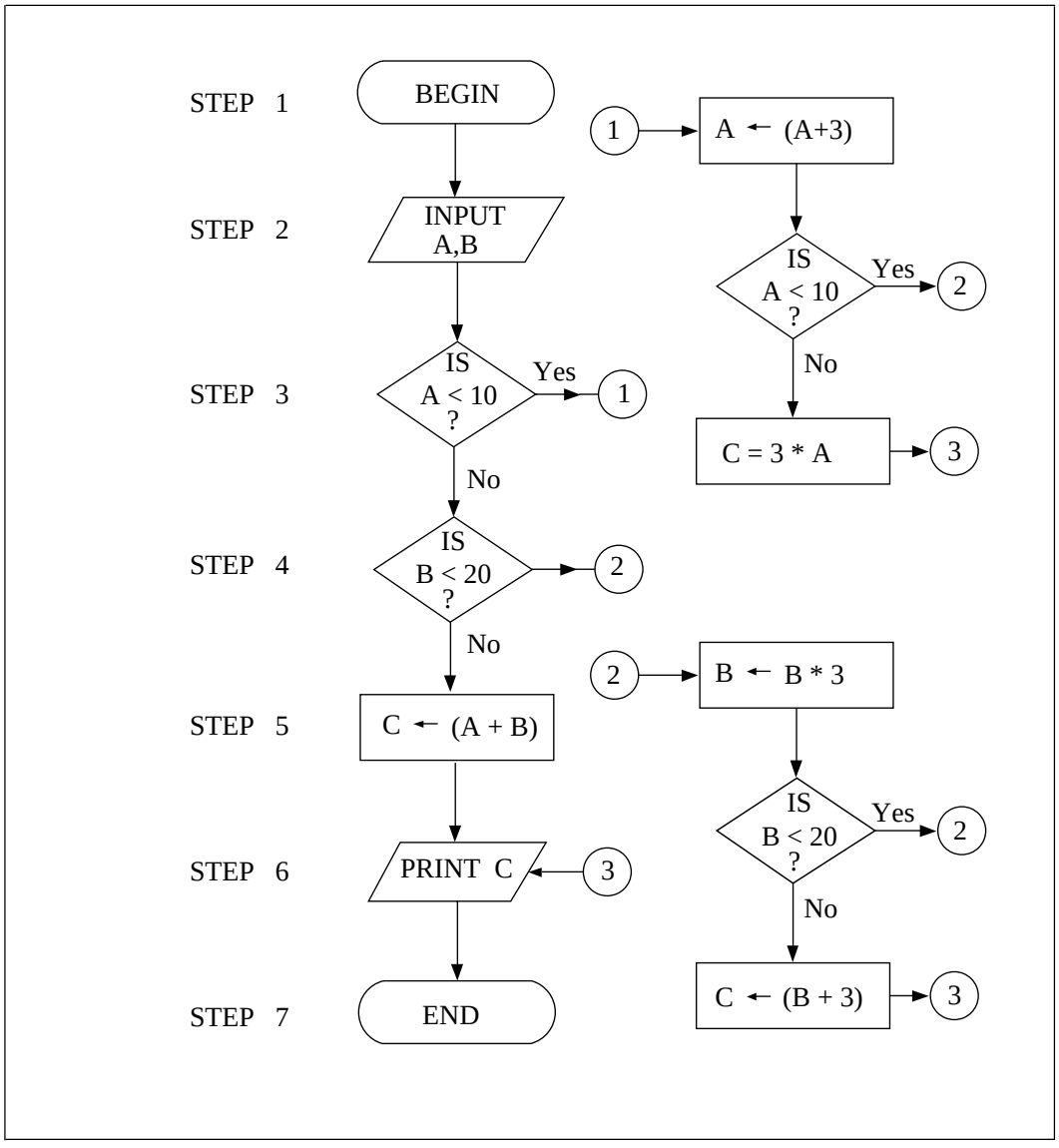

Figure 1.16 Flowchart for example 1.9

(d) Here A=10 and B=5
In step 3, the value of A is not less than 10, so the NO path is followed.

In step 4, the value of B is 5 which is less than 20. Hence connector 2 takes the control to right hand lower side flowchart. The current value of B becomes the old value of B multiplied by 3 which is now 15. This value of B i.e. 15 is the current value of B. In the decision diamond, this value of B is compared with 20. Since 15 is less than 20, so the YES path is followed. The control is transferred back via connector 2. The new value of B is calculated which will be 15 × 3 i.e. 45. This current value of B is again compared with 20. As it is not less than 20, so the NO path is followed.

The value of C = current value of B + 3
= 45 + 3
= 48

Control is transferred to step 6 via connector 3 and the value of C is printed. Thus the value of C = 48.

1.3.2 Loop

Many jobs that are required to be done with the help of a computer are *repetitive* in nature. For example, calculation of salary of different workers in a factory is given by the (No. of hours worked)× (wage rate). This calculation will be performed by an accountant for each worker every month. Such types of repetitive calculations can easily be done using a program that has a loop built into the solution of the problem .

What is a Loop ?

A loop is defined as a block of processing steps repeated a certain number of times. An *endless* loop repeats infinitely and is always the result of an error.

Figure 1.17 illustrates a flowchart showing the concept of looping. It shows a flowchart for printing values 1, 2, 3..., 20.

In step 5 of Figure 1.17, the current value of A is compared with 21. If the current value of A is less than 21, steps 3 and 4 are repeated. As soon as the current value of A is not less than 21, the path corresponding to "NO" is followed and the repetition process stops.

Terms Used in Looping:

Initialisation

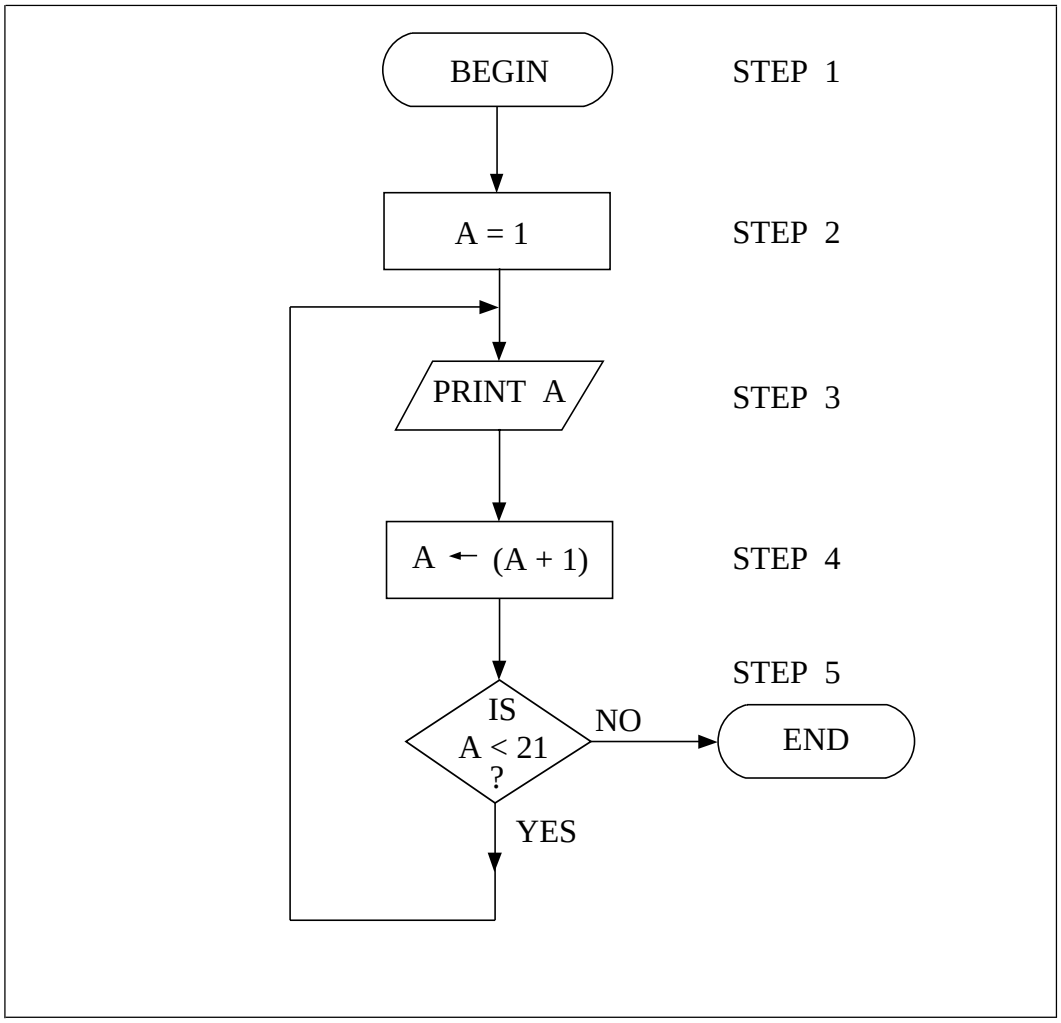

Figure 1.17 Concept of looping

It is the preparation required before entering a loop. In Figure 1.17, step 2 initialises the value of A as 1.

Incrementation It is the numerical value added to the variable each time one goes round the loop. Step 4 in Figure 1.17 shows the increment of A by 1.

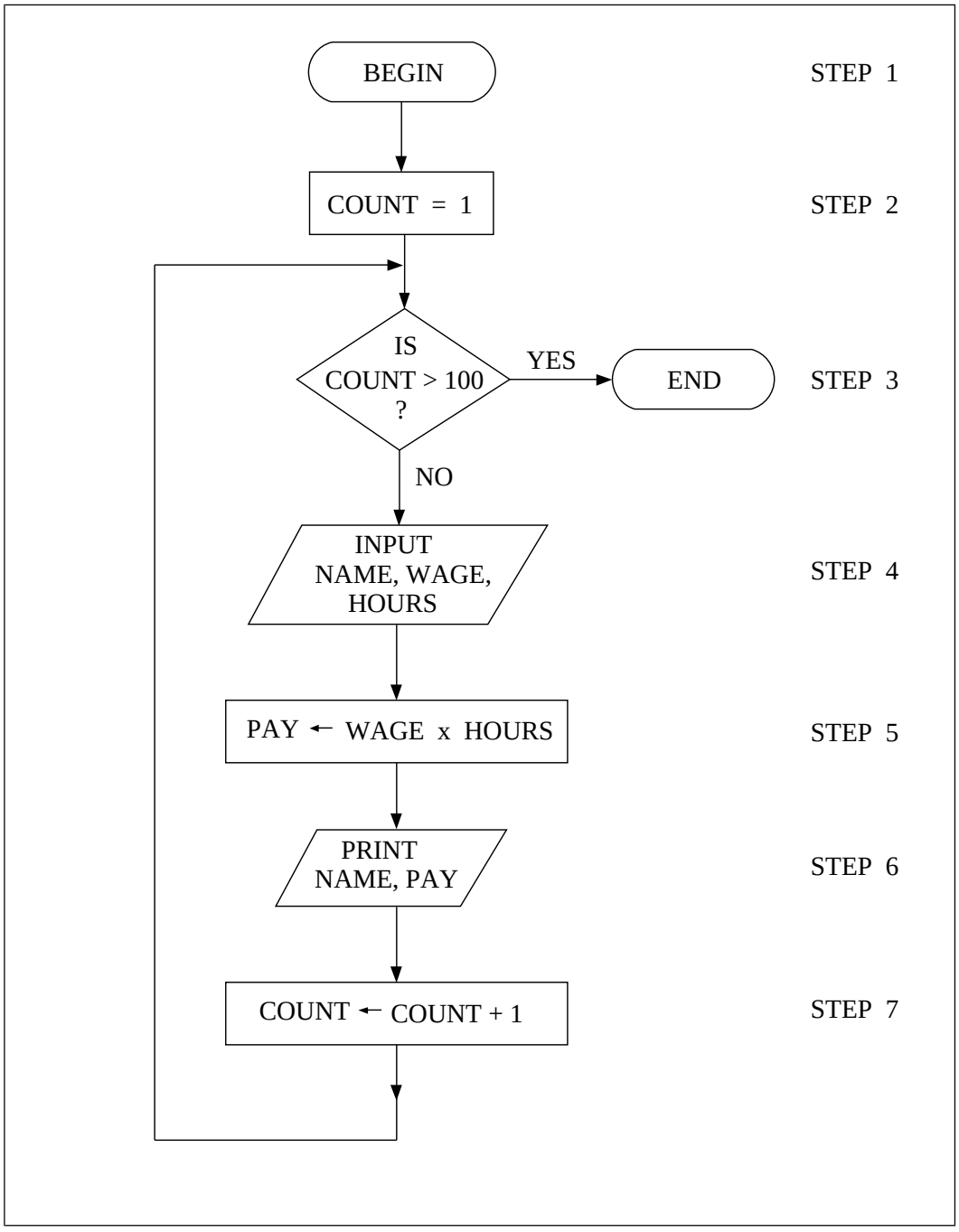

Figure 1.18 Flowchart for calculating the salary of 100 workers

The Loop Variable It is an active variable in a loop. In Figure 1.17, A is an active variable.

Loop Exit Test There must be some method of leaving the loop after it has revolved the requisite number of times. In Figure 1.17, step 5 is the decision diamond, where the value of A is compared with 21. As soon as the condition is satisfied, control comes out of the loop and the process stops.

Example 1.10

Draw a flowchart for calculating the salary of 100 workers in a factory.

Solution

Figure 1.18 represents the flowchart. In Figure 1.18, step 2 is for initialisation of the value of COUNT where COUNT is an active variable. Step 7 is the incrementation. Step 3 is for the EXIT test. Steps 4 to 6 are the repetitive steps in the loop to input the NAME, WAGE and HOURS, and then calculate the value of PAY in step 5 and print the name and pay in step 6.

In step 7, the value of COUNT is increased by 1 and the current value of COUNT is compared with 100 in step 1. If it is more than 100, the process is halted.

Exercise 1.1

Q 1. How many records will be read by the flowcharts, shown in Figures 1.19 and 1.20.

Q 2. A certain file contains 15 records. Which of the flowcharts shown in Figures 1.21 and 1.22 read 15 and only 15 records?

Q 3. The formula to compute simple interest I on a loan of rupees P at an interest rate R for T years is given as

$$I = \frac{PRT}{100}$$

Draw a flowchart to input the values of P, R and T and to calculate the value of interest I. Also print the values of P, R, T and I.

Q 4. The flowchart shown in Figure 1.23 is drawn to print the sum of integers 1 to 100. What changes would you make in this flowchart, if the SUM for the first 100 odd integers is required? Redraw the flowchart.

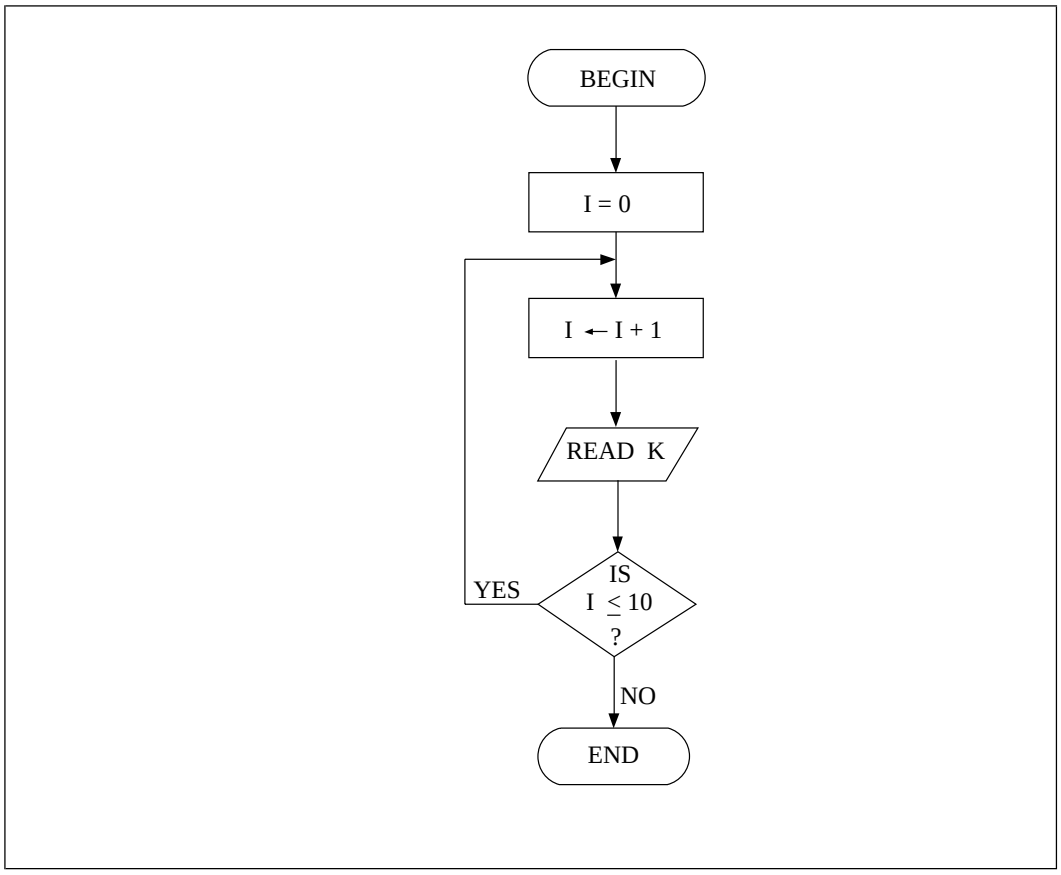

Figure 1.19 Reading of records

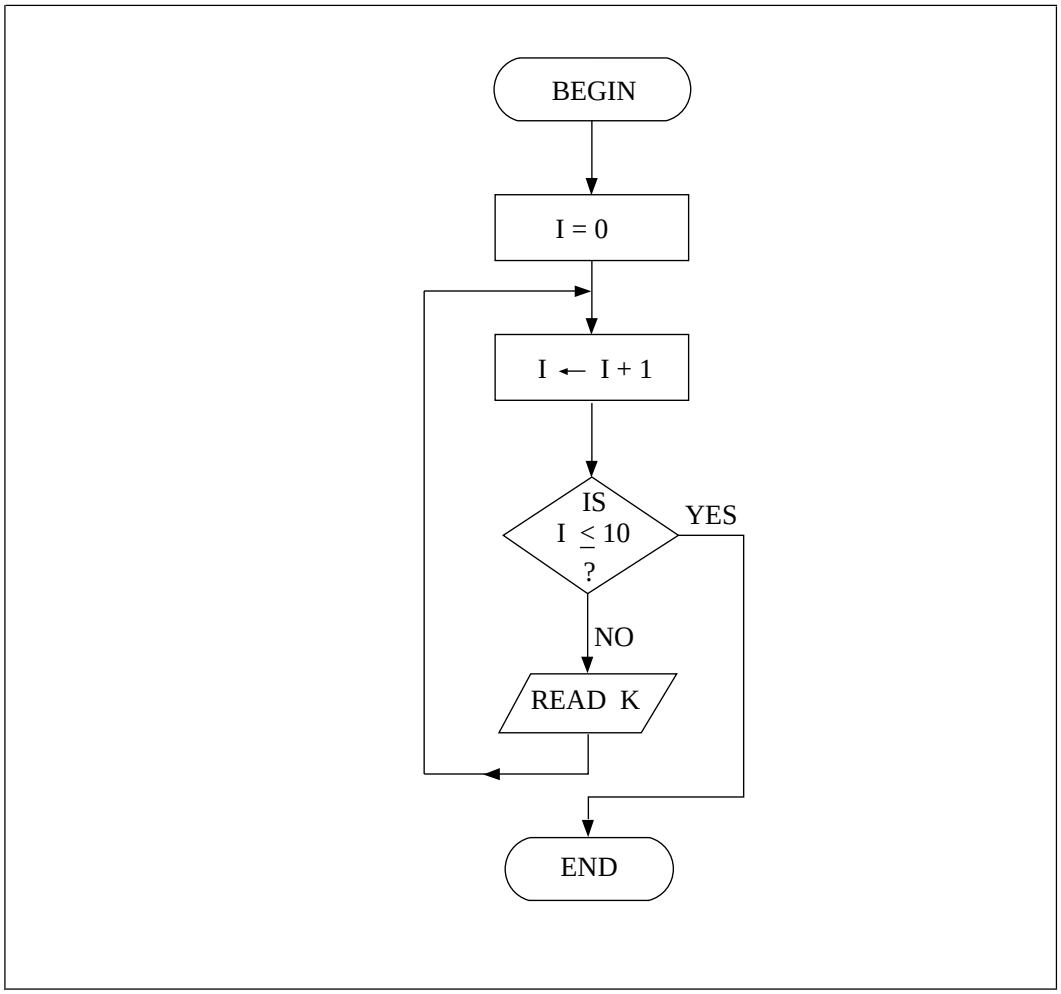

Figure 1.20 Reading of records

Q 5. Consider an equation $Y=X^2+2X+3$. Draw a flowchart which gives a method for calculating the values of Y for different values of X varying from -4 to +4 in steps of 1 and prints each value of X and the corresponding value of Y.

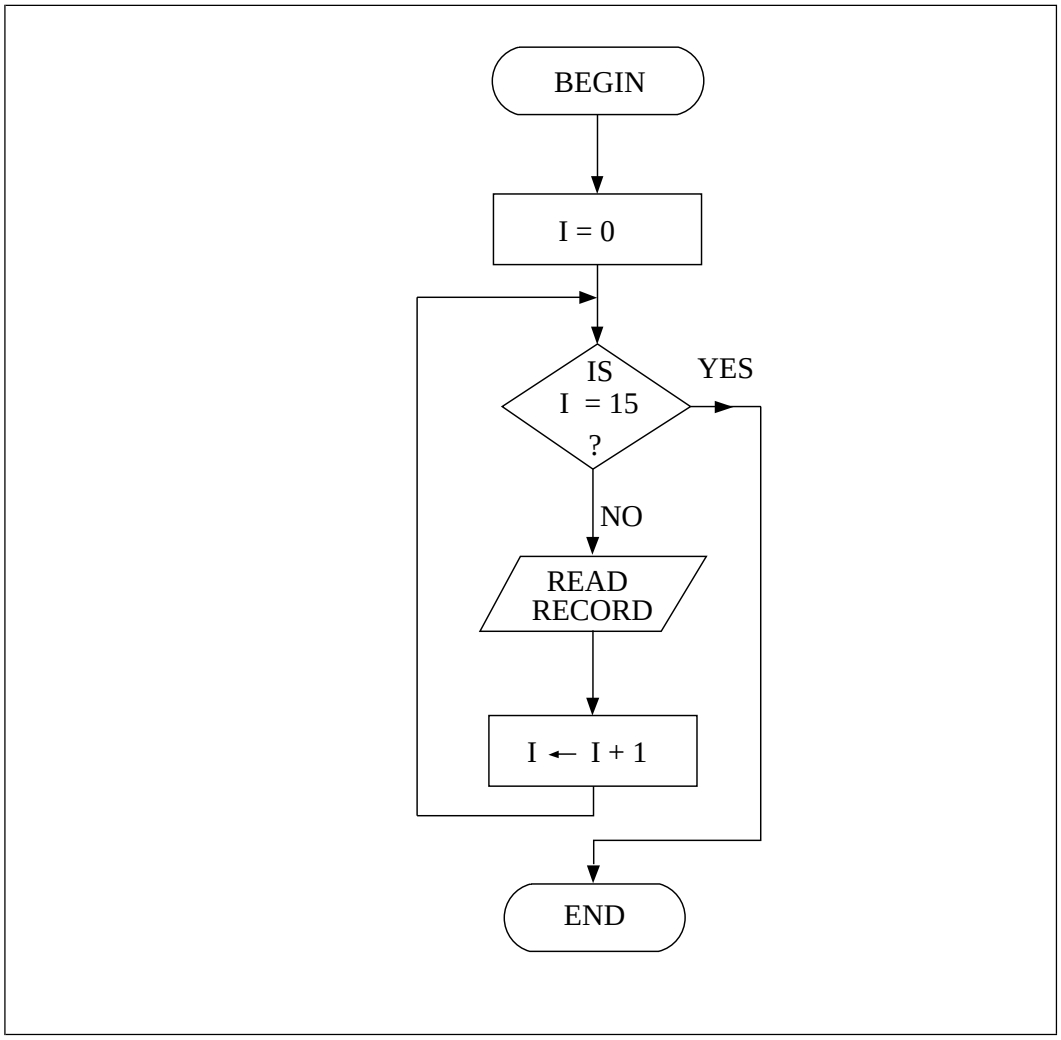

Figure 1.21 Counting of records

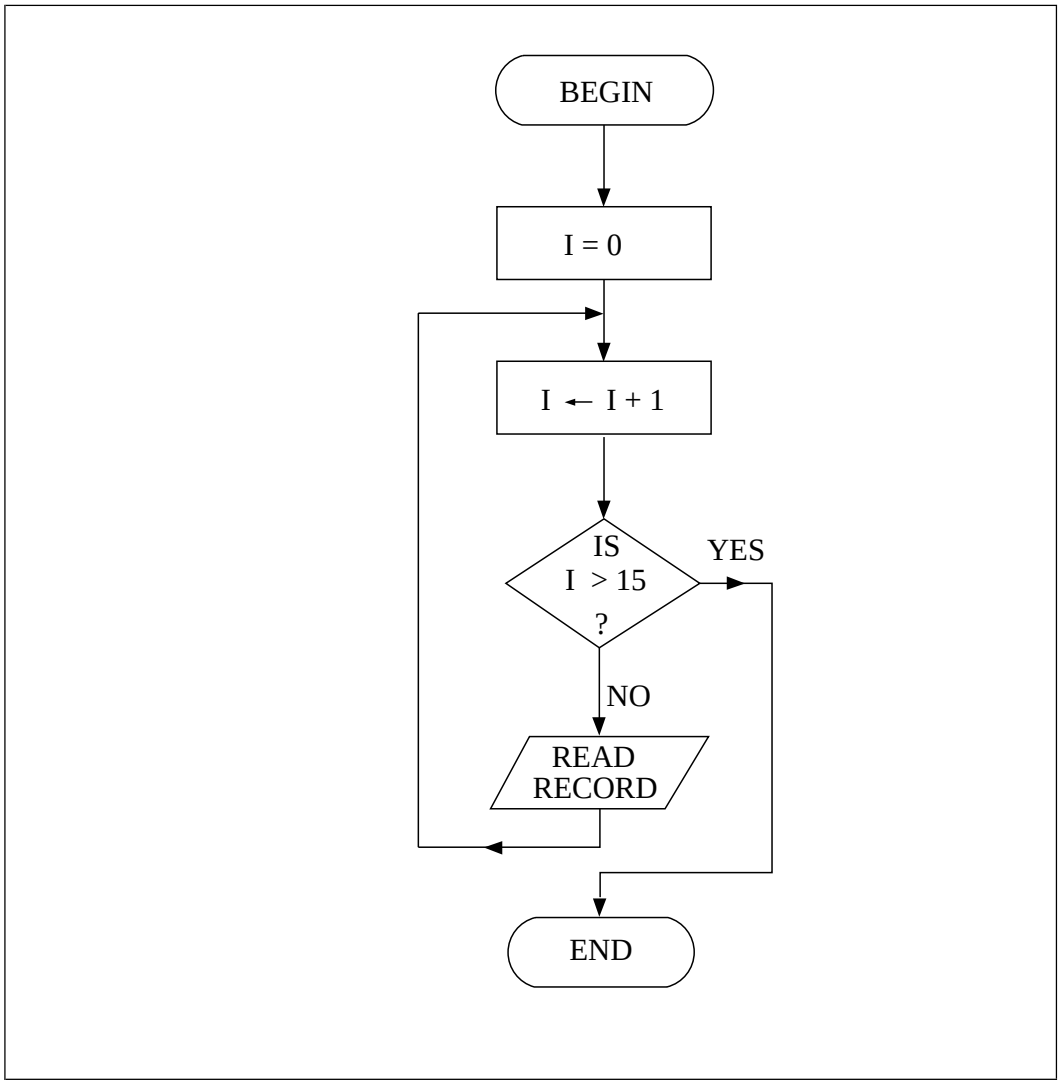

Figure 1.22 Counting of records

1.3.3 Counting

Counting is an essential technique used in the problem solving process. It is mainly for repeating a procedure for a certain number of times or to count the occurrences of specific events or to generate a sequence of numbers for computational use. Since a

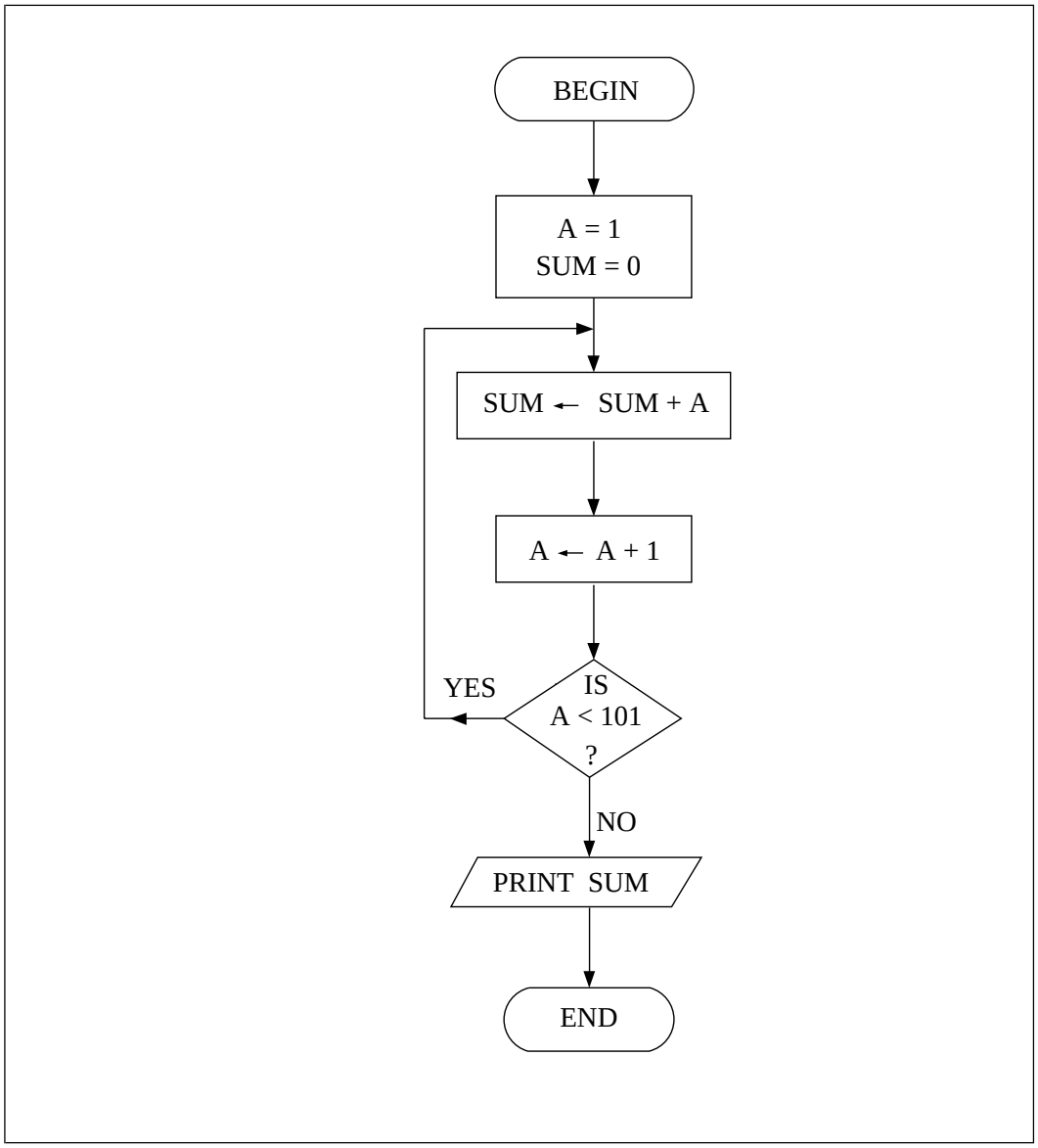

Figure 1.23 Calculating sum of integers 1 to 100

computer cannot count by itself, the user has to send the necessary instruction to do so. The counting process is illustrated in the flowchart shown in Figure 1.24

Here I is a counter which is initialised to a value zero in step 2. In step 3, a NAME is read and stored in the memory of the

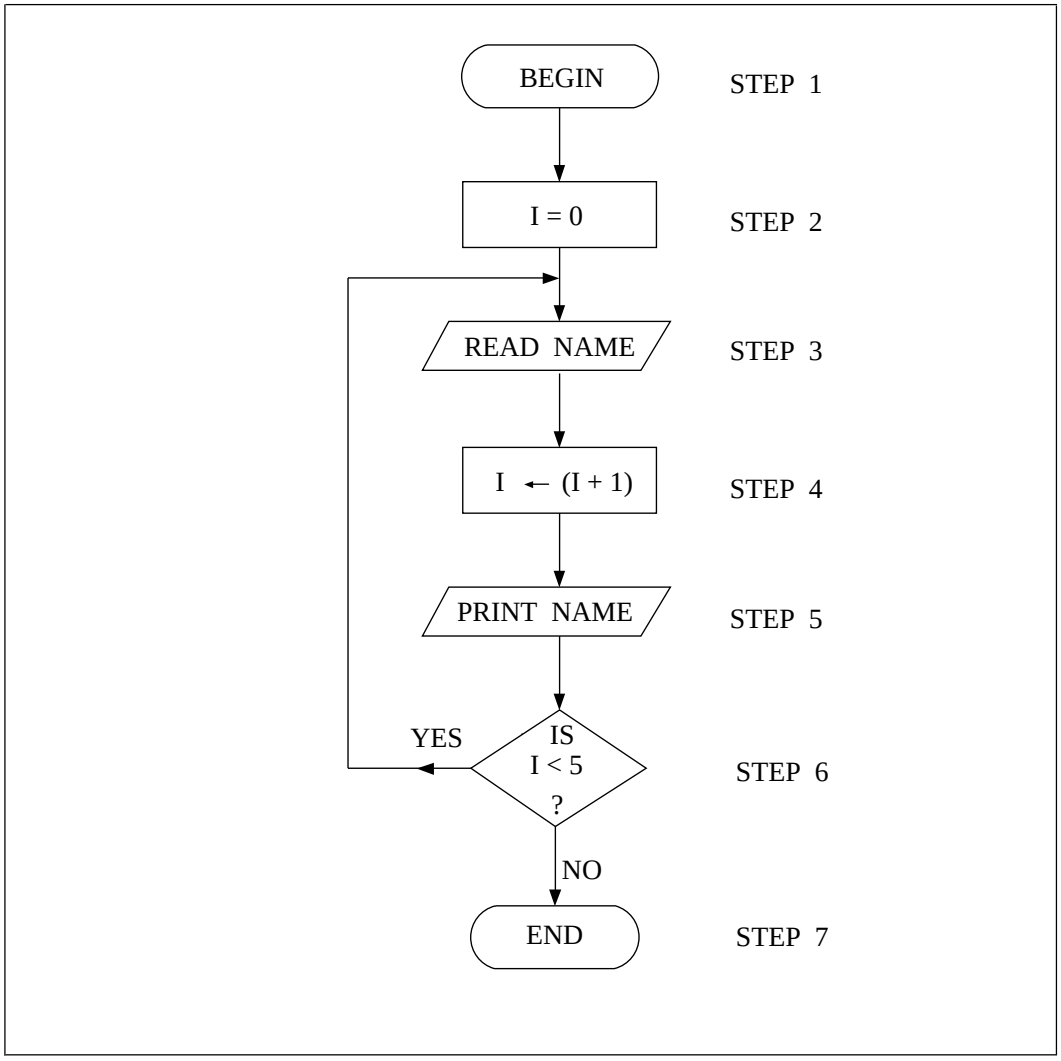

Figure 1.24 Counting process

computer. The value of the counter is incremented by 1 in step 4. In step 5, the NAME is printed from the memory of the computer. In step 6, a check is made on the value of I. If the current value of I is less than 5, the cycle is repeated from steps 3 to 5. If the value of I is equal to or more than 5, the process of reading and printing NAME stops. Using this type of flowchart we can read and print the NAME five times.

Example 1.11

Which numbers will be printed by the flowcharts shown in Figures 1.25 and 1.26 respectively?

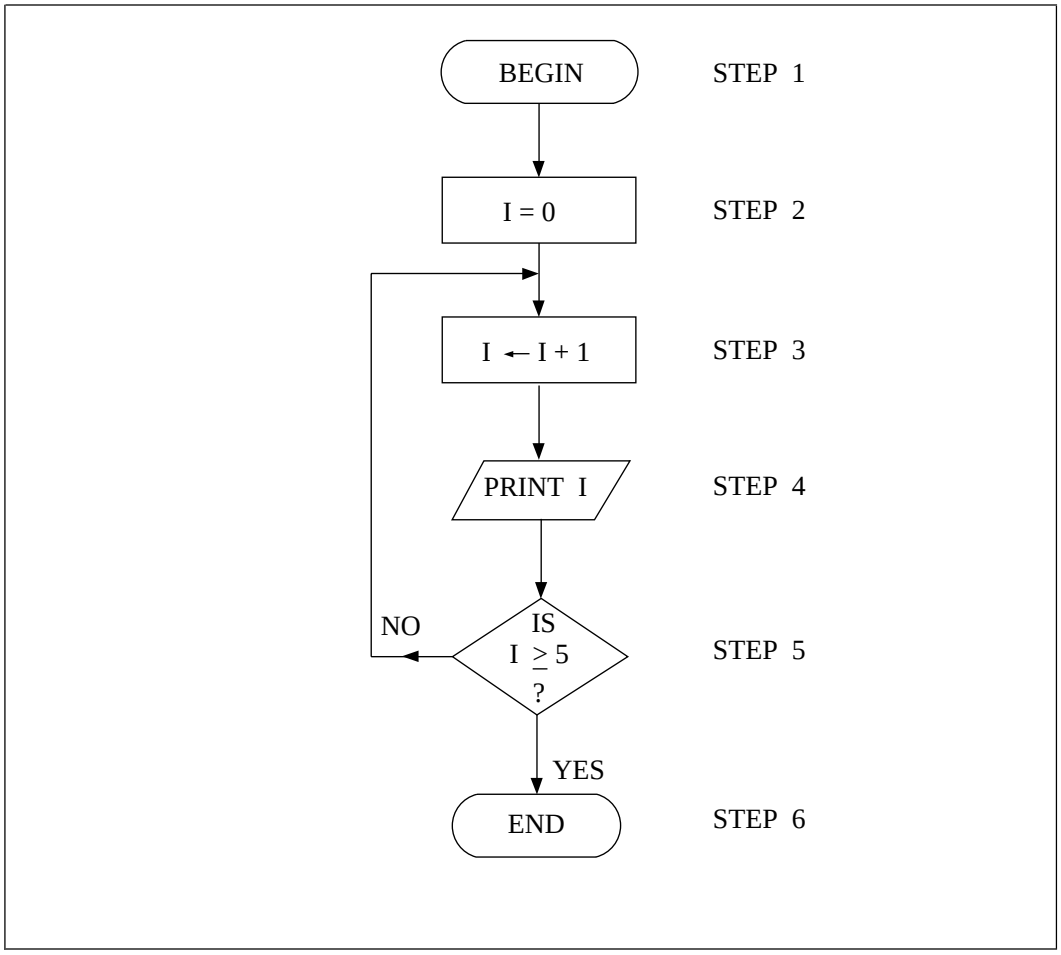

Figure 1.25 Flowchart for printing numbers

Solution

(a) In Figure 1.25, the value of I is initialised to zero in step 2. In step 3, the value is incremented by 1, i.e. it is 1. In step 4

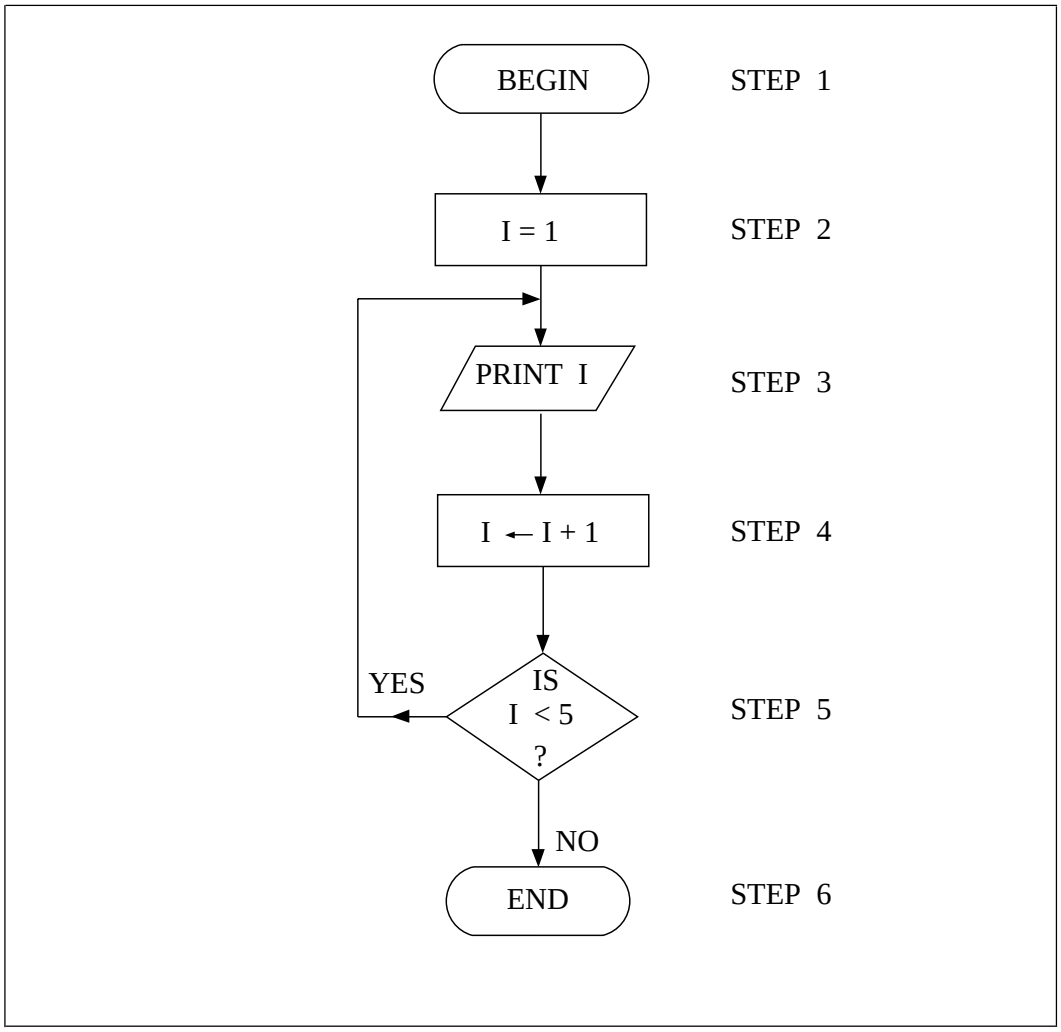

Figure 1.26 Flowchart for printing numbers

I is printed. In step 5, the current value of I is compared with 5. Since the current value of I is 1 and 1 is not greater than or equal to 5, the control is transferred to step 3. Steps 3 and 4 are repeated again till the current value of I is 5. Thus, the values of I that are printed would be 1, 2, 3, 4, 5.

(b) In Figure 1.26, the value of I is initialised to 1 in step 2. The number 1 is printed in step 3. In step 4, the value of I is increased by 1. So the current value of I is 2. In step 5, 2 is compared with 5. Since 2 is less than 5, so the control is

transferred to step 3 and the number 2 is printed. This cycle repeats and numbers 3 and 4 are printed. When the current value of I reaches 5, the control is transferred from step 5 to 6. Thus the numbers printed are 1, 2, 3 and 4.

Exercise 1.2

Q 1. Which numbers will be printed using the flowcharts shown in Figures 1.27 and 1.28 respectively.

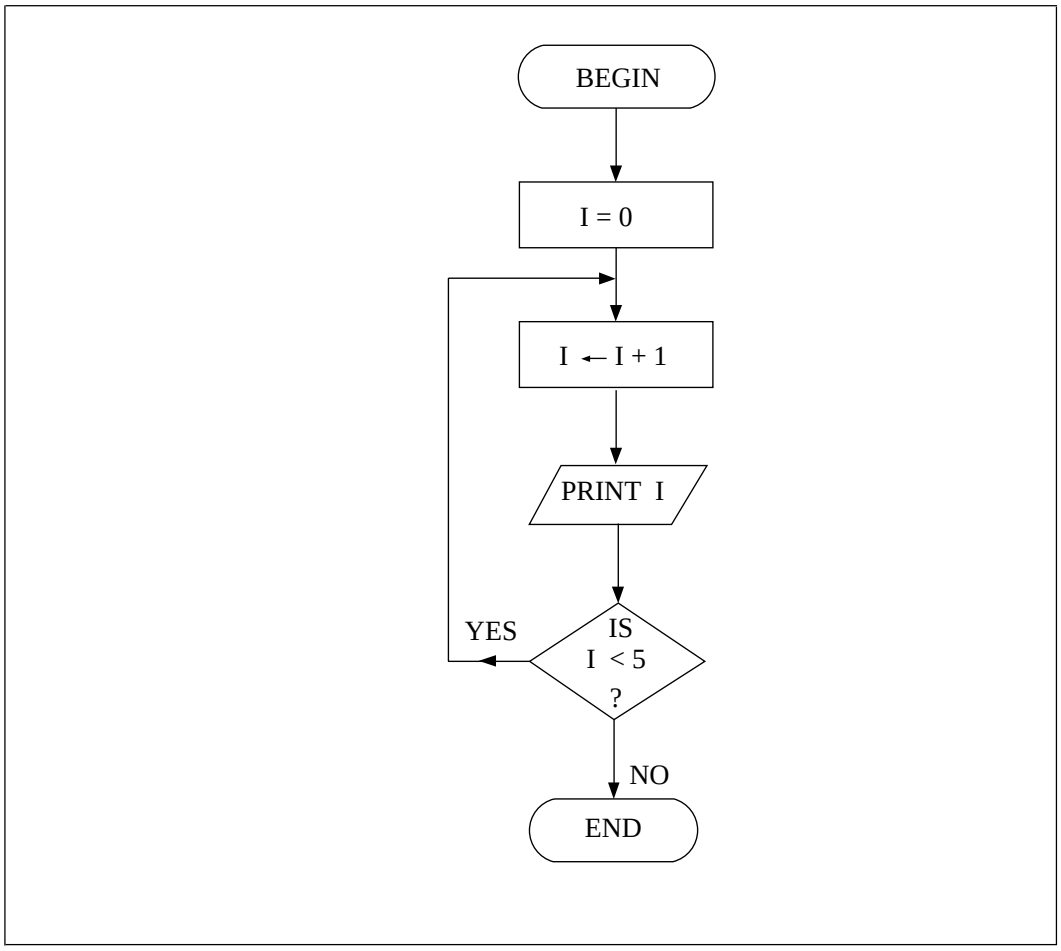

Figure 1.27 Printing of numbers

Q 2. Write a flowchart to print the following sequence of numbers.

(a) 20, 19, 18, 17,...., 1.

(b) 2, 4, 6, 8,....., 32.

Q.3 In the flowchart shown in Fig 1.29, the following data is inputted.

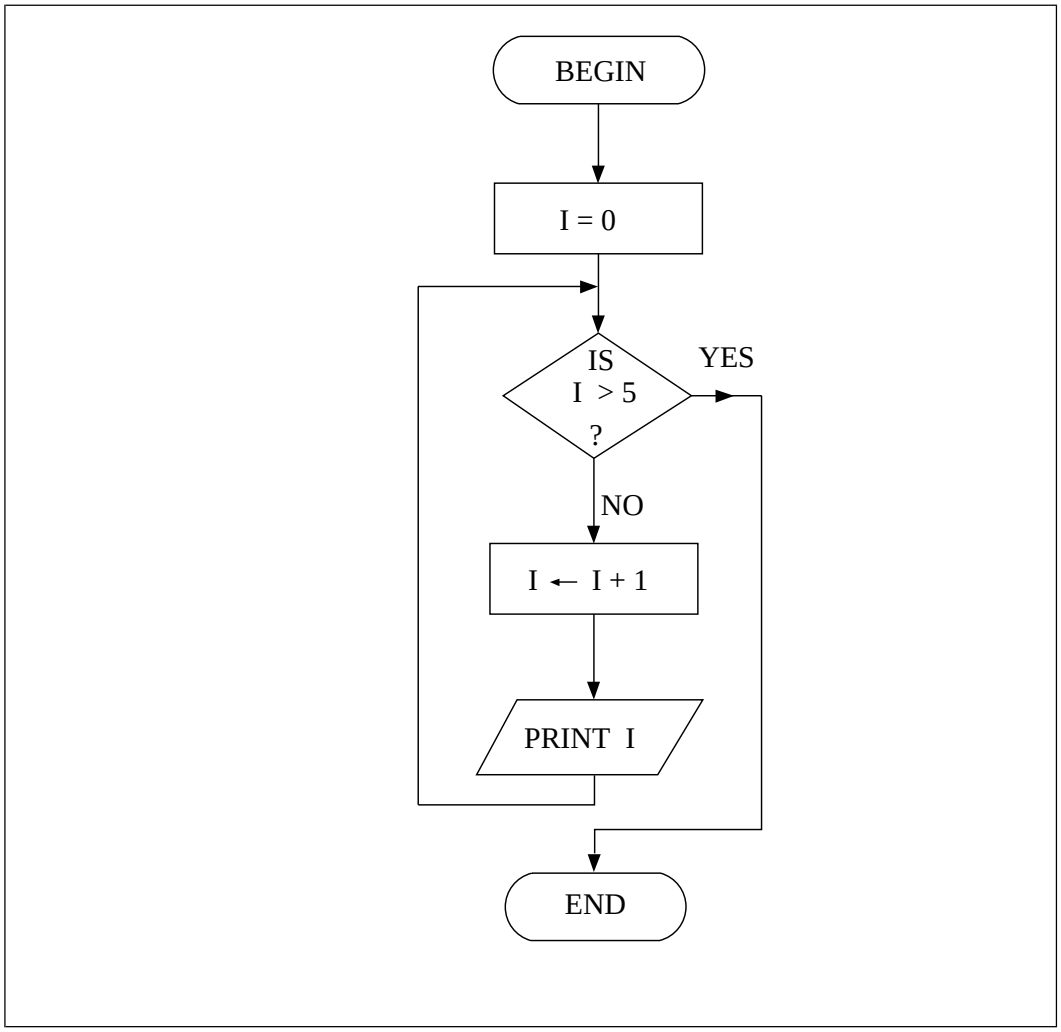

Figure 1.28 Printing of numbers

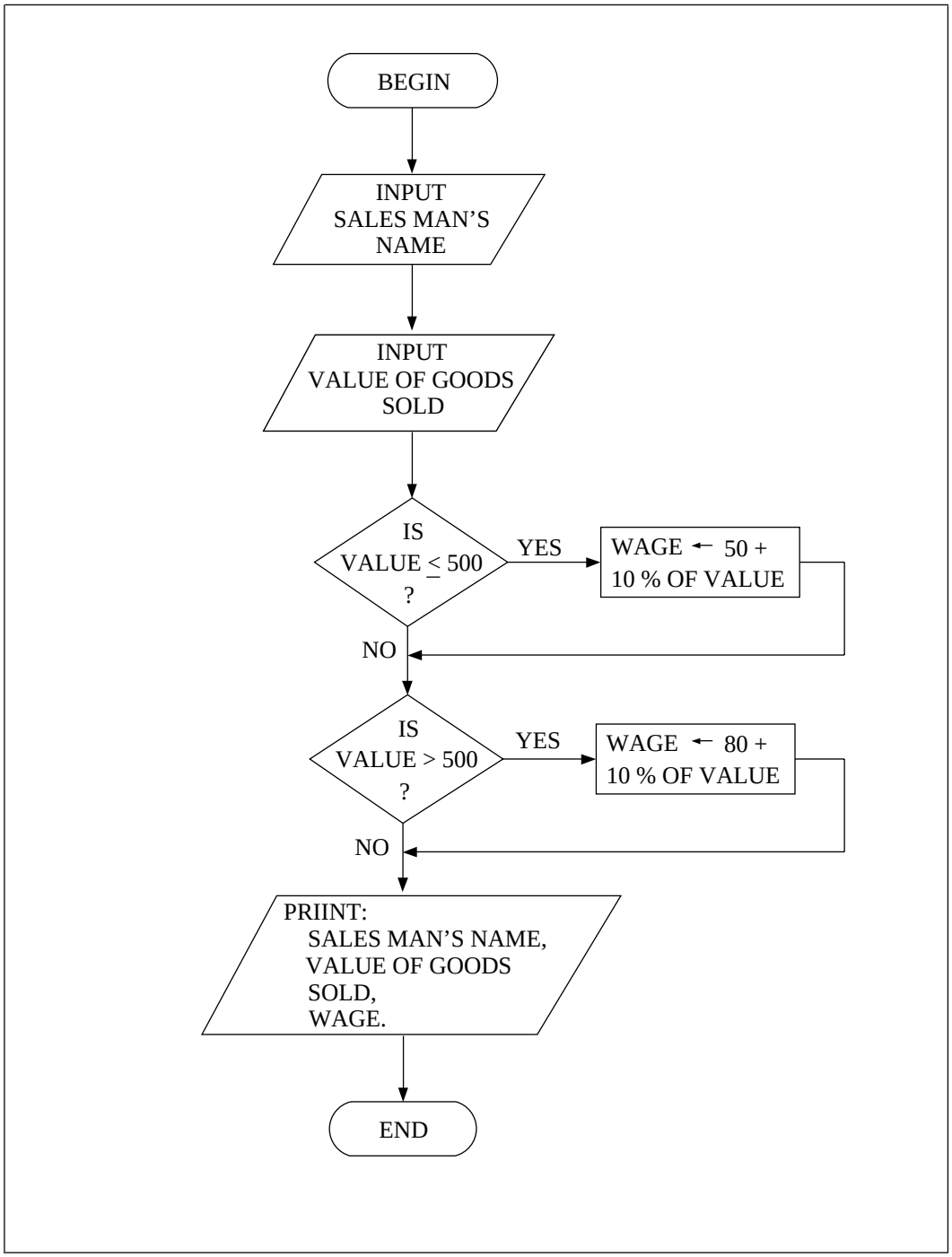

Figure 1.29 Flowchart for Q. 3 in Exercise 1.2

Input Data

	Salesman's Name	*Value of goods sold*
(a)	Mohan	1500
(b)	Shyam	2100
(c)	Krishna	510

Calculate the value of wage in each case and give the result of the print statement.

1.3.4 Counting for Controlling a Loop

Sometimes, it is essential to repeat a process for a specific number of times only. In such a case, there are two standard techniques used. These technique are as follows:

Technique 1

In this technique, we use the following six steps.

1. Initialise a counter to 0, i.e. COUNT = 0. Input value of N where N stands for the number of times a loop is to be repeated.
2. Increase the counter by 1. or COUNT = COUNT + 1
3. Test the value of counter and compare the current value with N. If the current value of the COUNT is greater than N, then branch off to step 6, otherwise continue.
4. Carryout the sets of instructions (procedure).
5. Go back to step 2.
6. End the loop and continue further programming. These six steps are shown in Fig 1.30.

Technique 2

1. In this method, the counter is first initialised to zero and the value of N is inputted.
2. Carry out the sets of instructions of the program.
3. Increase the counter by 1.
4. Test the counter. If COUNT < N then Go to step 2.
5. ELSE END

The above mentioned six steps are represented pictorially by the flowchart shown in Figure 1.31.

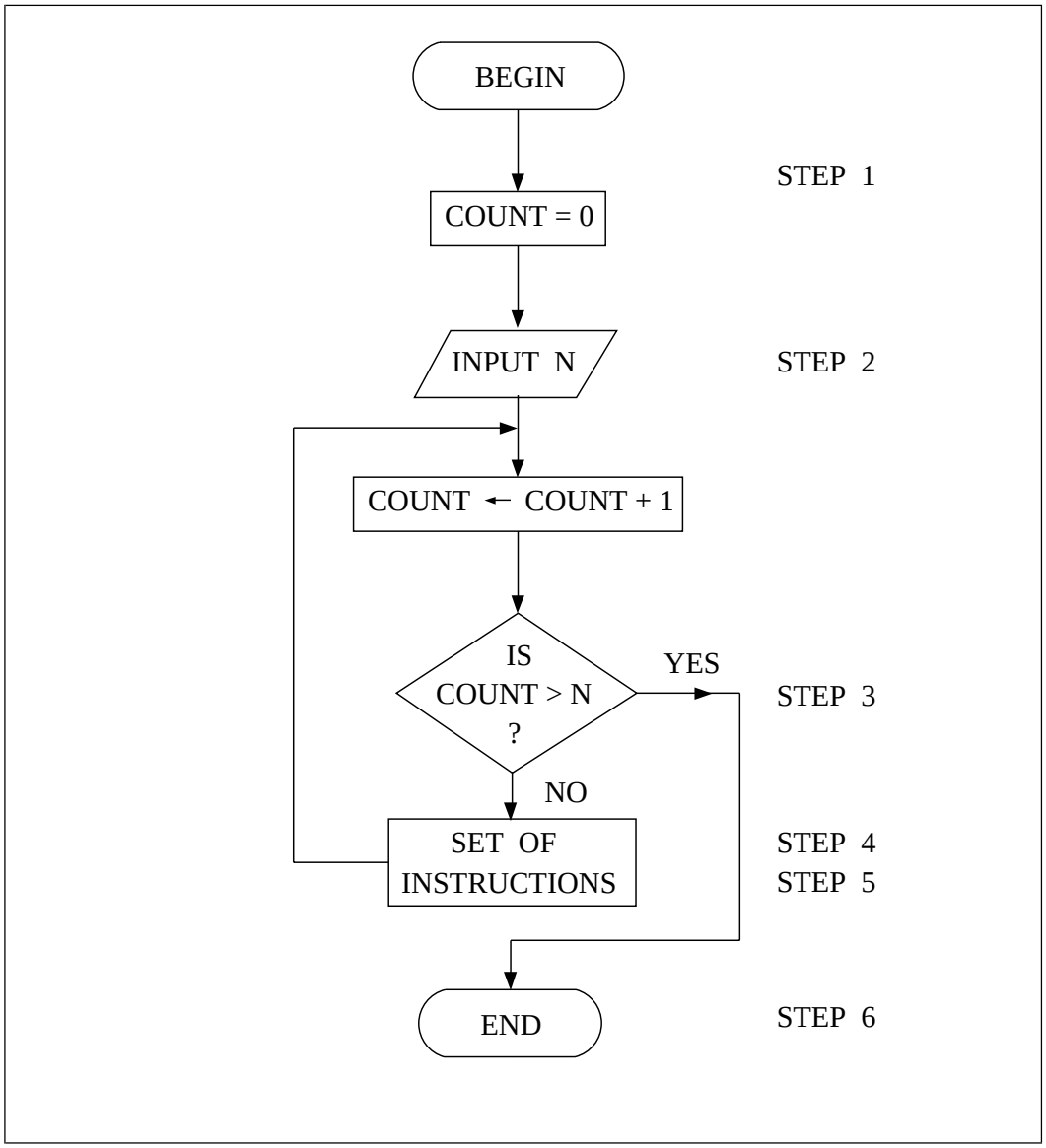

Figure 1.30 Flowchart for controlling a loop (Technique 1)

Sometimes, it may be necessary to repeatedly use any of the above two techniques in order to solve a specific problem. This is called nested loop. It means that we have the first loop and within this loop, there is another loop. Solving a problem using the nested loop technique is very helpful for a multidimensional array.

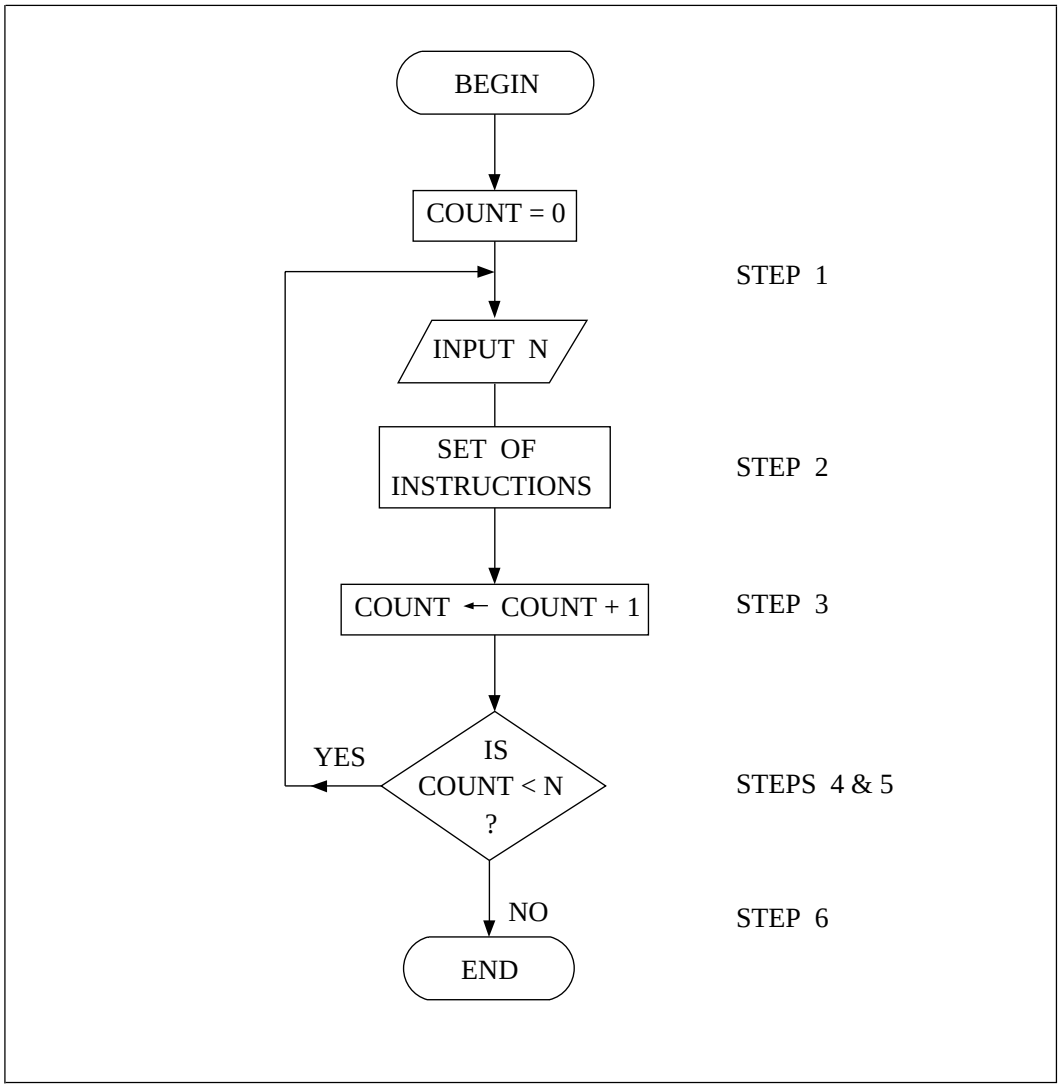

Figure 1.31 Flowchart for controlling a loop (Technique 2)

1.4 Procedure for Problem Solving

Though every problem is an entity in itself, there are a few basic steps that should be understood and followed for effectively solving a problem using computer techniques. On following these steps, your problem solving capacity will improve. There are basically six steps in solving a problem. These are:

1. First, spend sometime in understanding the problem thoroughly. In this, you are not required to use a computer. Instead try to answer and solve the problem manually.
2. Now construct a list of variables that are needed to solve the problem.
3. Once you have completed step 2, you have to decide the layout for the output format.
4. Next, select the programming technique which is best suited to solve the problem and carryout the coding.
5. Test your program. Choose some test data so that each part of the program is checked for correctness.
6. Finally use the data validation program to guard against processing of wrong data.

The above six steps are further explained in the following paragraphs.

1.4.1 Step 1. Understanding the Problem

Read each *statement* in the problem carefully, so that you can answer the first question "What is expected by solving the problem?" Do not start drawing a flowchart straight away. Instead, read each statement of the problem slowly and carefully, understanding the *keywords*. Take a pencil and paper and try to solve the problem manually for some test data. Let us understand this point by solving the problem, given in Example 1.12.

Example 1.12

Write a flowchart to accept a value M and find the sum of first M even integers.

Solution

The solution of this problem requires you to draw a flowchart so that if you input 6 as the value of M, the flowchart should get you the sum of first 6 even integers. In the first step, you should be able to answer the following two questions.

"What are the first 6 even integers?"

These are 2, 4, 6, 8, 10, and 12.

"What is their sum?"

The sum is 42. Hence the flowchart is to be so framed that the sum of first 6 even integers comes out to be 42.

1.4.2 Step 2. Construction of the List of Variables

In this step, you should think in advance the number of variables and the names of the variables before drawing a flowchart. The names chosen for variables should be an aid to memory. For example, in the case of the problem stated in step 1 above, the variables may be, I, SUM and COUNT as given below.

1. Generate even integers 2, 4, 6,...(I)
2. Total the sum of even integers 2+4+6+... (SUM)
3. Count the number of even integers, i.e. COUNT 1, 2, 3,... (COUNT)

Thus, it is clear that we need to use the above three variables and one more variable "M" whose value will be inputted from the keyboard. Hence the four variables are:

M	Which is to be inputted from the keyboard.
I	Which is to generate even integers.
COUNT	A counter to keep a track of the number of even integers that have been summed.
SUM	An accumulator that contains the current total of the even integers.

1.4.3 Step 3. Output Design

Sometimes, the 'output' format is specified in the problem, but most of the times, it is not so. If the output format is not specified, we must keep in mind that the output report should be *easily* understandable by a reader. The headings should not cause any ambiguity in the mind of the reader.

In the problem of Example 1.12 and stated in step 1 above, the output format could be as follows:

No of integers	Total value
6	42

You should keep one point in mind. The programs and problem solutions are for other people (teachers, supervisors, contrac-

tors etc). They will give you credit only if they can understand the results and analyse them. Hence, the output format should be attractive, easy to read and self-explanatory.

1.4.4 Step 4. Program Development

You should now draw a flowchart for the procedure that you have developed in steps 1 to 3 above. Standard symbols should be used for drawing a flowchart. If a problem is complex, you should divide it into different parts. Then draw a flowchart for each part separately and join them together using connectors.

A flowchart for the problem in Example 1.12 and stated in step 1 can be drawn as shown in Figure 1.32.

When a flowchart is drawn correctly, you can convert it into a programme using any of the high level languages like BASIC, COBOL, FORTRAN or PASCAL.

1.4.5 Step 5. Testing the Program

You should give a *dry* run to a program that translates the flowchart of step 4. This means giving some known values to the variables and checking the result. Test values are so selected that each arm of the flowchart is tested and consequently the program is confirmed to be free from logic errors.

1.4.6 Step 6. Validating the Program

It is quite likely that the user of your program may enter values, which are not expected by the program. Such values should be rejected by the procedure drawn by you. This is known as validation of data program. For example, in the problem in Example 1.12 and stated in step 1 above, we may give the limit to the value of "M" and "M" should be a numerical value. Such types of checks can be included in our validation program.

1.5 Algorithm

The sequence of instructions for solving a particular problem is known as algorithm. Constructing an algorithm to solve a given problem requires a high degree of ingenuity. But once an

algorithm is laid out, it can be used by a person who does not even know its purpose.

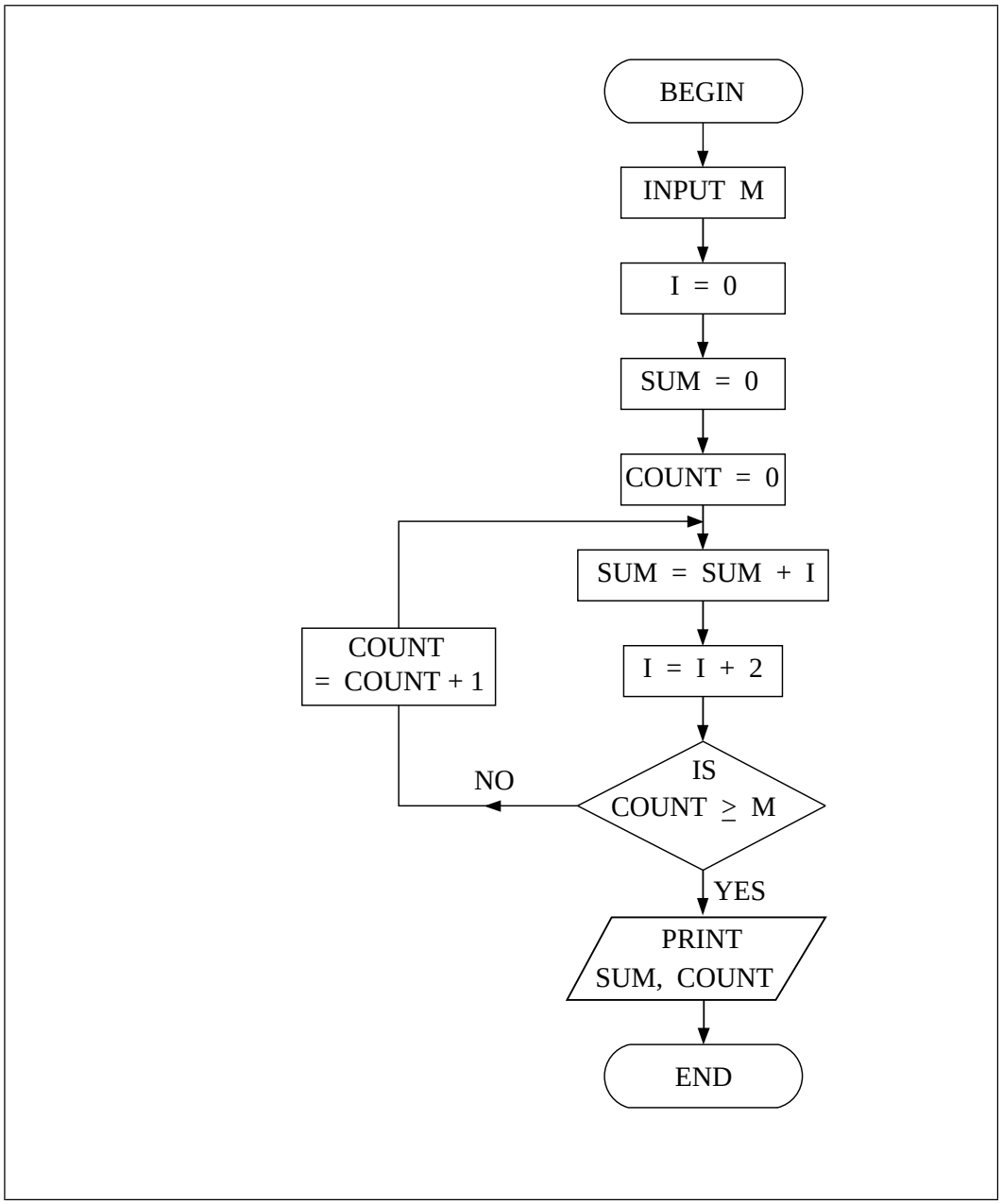

Figure 1.32 Flowchart to Example 1.13

We shall write an algorithm to solve a quadratic equation and compute its real roots.

Example 1.13

Compute real roots of a quadratic equation

$$Ax^2 + Bx + C = 0$$

Solution

Step 1

Read the values of the coefficients of X^2, X and constant quantity, i.e., the value of A, B and C and store them in the memory.

Step 2

If A = 0 and B = 0 then print "No root exists" and stop. Else continue.

Step 3

If A = 0 and B is not equal to 0, then root X1 = –C/B. Write the value of root and also write "Linear Equation" and stop. Else continue.

Step 4

Compute (B^2–4AC) and set D^2 = B^2–4AC

Step 5

If D = 0, then compute Root X1 = Root X2 =(–B/2A), write values of Root X1, Root X2 and stop. Else continue.

Step 6

If D < 0, then write "Roots are not real" and stop. Else continue.

Step 7

If D > 0, then calculate
Root X1 = (–B + (B^2 – 4AC)/(2A))
Root X2 = (–B – (B^2 – 4AC)/(2A))

1.5.1 Characteristics of the Instructions in an Algorithm

Write values of Root X1, Root X2 and stop.

The sequence of instructions must possess the following characteristics for qualifying as an algorithm :

1. Each and every instruction should be *precise* and *unambiguous*.
2. Each instruction should be such that it can be performed in a finite time.
3. One or more instructions should not be repeated indefinitely. This ensures that the algorithm will ultimately terminate.
4. After performing the instructions, that is, after the algorithm terminates, the desired results must be obtained.

Example 1.14

Let us consider another case where there are 50 students in a class who have appeared in their final examination. Their marksheets have been given. We are required to write an algorithm to calculate and print the total number of students who have passed in first division.

Solution

Step 1

Initialize TOTAL, FIRST_DIVISION and TOTAL MARKSHEETS_CHECKED to zero.

Step 2

Take the marksheet of the first student.

Step 3

Check the division column of the marksheet to see if it is FIRST_DIVISION : if no, go to step 5.

Step 4

Add 1 to TOTAL FIRST_DIVISION.

Step 5

Add 1 to TOTAL MARKSHEETS_CHECKED.

Step 6

Is TOTAL MARKSHEETS_CHECKED = 50? : if no, go to step 2.

Step 7:

Print TOTAL FIRST_DIVISION.

Step 8:

Stop.

1.6 Programming

If we compare structured programming to building a house, then structured programming is to build the house after the plan for the house is drawn. We would like to build a house using standard bricks, window-frames, doors etc., so that the house so constructed is appealing and comfortable. Thus, structured programming also contains three standard control structure.
These are:

The above three structures are simple to use as they can be recognised easily. They have one entry and one exit point and they are free from any programming language used for coding a solution of a problem. This is discussed in detail in section 1.8.

1.6.1 Modular (Top-Down) Program Design

You must have realised that an effective approach to follow in the programming analysis stage of program development is to break down a large problem into a series of smaller and more understandable tasks. Thus, the programmer may first develop a main-control program that is used to outline the major segments, or modules, that are in turn needed to solve a problem. The main-control program specifies the order in which each subordinate module in the program will be processed. The programming

analysis stage continues until every module has been reduced to the point that the programmer is confident that he or she has a solution method that will solve the task.

When this modular (top-down) program design practice is used, an instruction in the main-control program branches control to a subordinate program routine (subroutine) or module. When the specific processing operation performed by the module is completed, another branch instruction may transfer program control to another module or return it to the main-control program. Thus, the modules or subroutines are really programs within a program. Each module typically has only one entry point and only one exit point. Some of the advantages of using this construction option are:

(a) Complex programs may be divided into simpler and more manageable elements.

(b) Simultaneous coding of modules by several programmers is possible.

(c) A library of modules may be created, and these modules may be used in other programs as needed.

(d) The location of program errors may be more easily traced to a particular module, and thus debugging and maintenance may be simplified.

(e) Effective use can be made of tested subroutines prepared by software suppliers and furnished to their customers.

1.6.2 Pseudo Code Revisited

Pseudo code is an abbreviated form of expression that makes use of both the English language and certain programming language control words such as IF - THEN - ELSE & END - IF.

The user describes in plain English language, the sequence of steps necessary to solve a particular problem. Sentences are generally written one per line. Indentation is used in the IF statement to outline which actions are to be taken if a condition is true and which are to be taken if the condition is not true.

Differences between PseudoCode and Flowchart

Pseudocode differs from the flowchart in the following ways:

(a) Pseudocode is self-explanatory and does not require a separate documentation. This is so because it is written in plain English language.

(b) Pseudocode has a structure similar to that of a programme written in BASIC or PASCAL language. But flowcharts have a tendency to extend flow lines in all directions and thus their paths of instructions do not parallel the final BASIC code.

The following are some examples of flowcharts and their equivalent pseudocode.

Example 1.15

Working with Masala Dosa Mincer

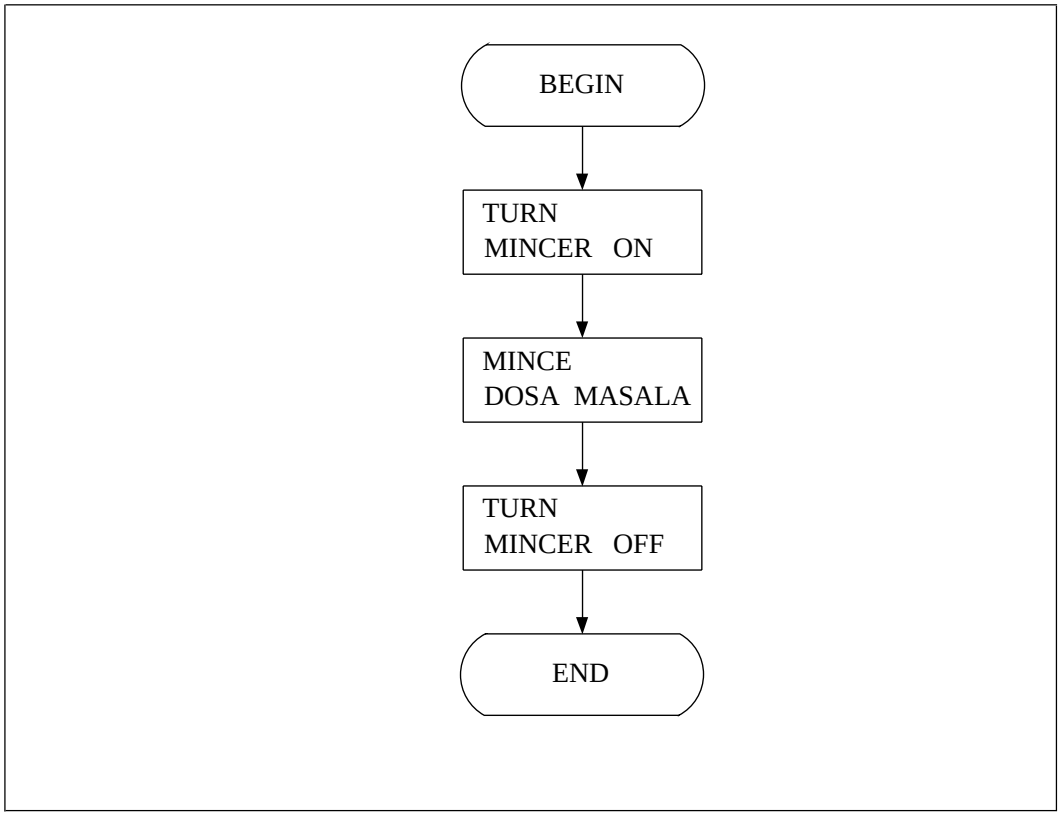

Figure 1.33 Flowchart for Example 1.15

Pseudo code

BEGIN

Turn Mincer on

Mince dosa masala

Turn Mincer off.

END

This is a case of simple sequence. The flowchart is given in Figure 1.33

Example 1.16

In this example a flowchart for Drink selection is shown in Figure 1.34

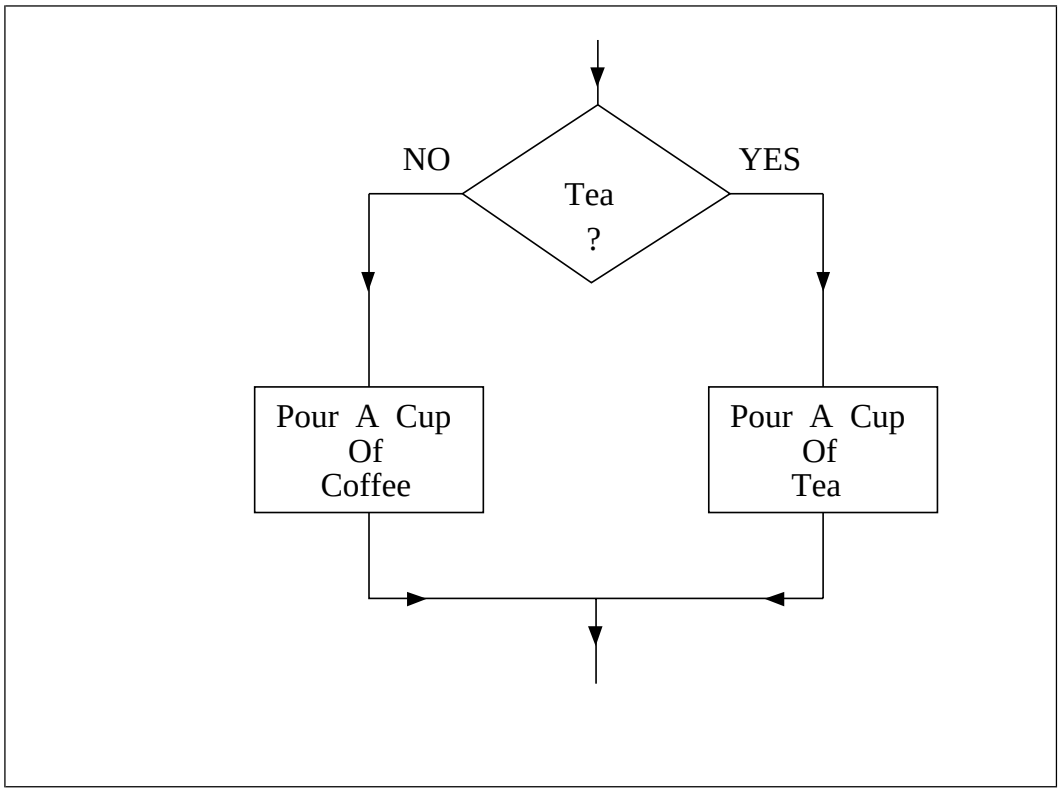

Figure 1.34 Process of selection for example 1.16

Pseudo Code

IF Tea
Then Pour a cup of Tea
Else Pour a cup of Coffee

Example 1.17

In this example, the flowchart and its equivalent pseudo code for the process of repetition is shown in Figure 1.35.

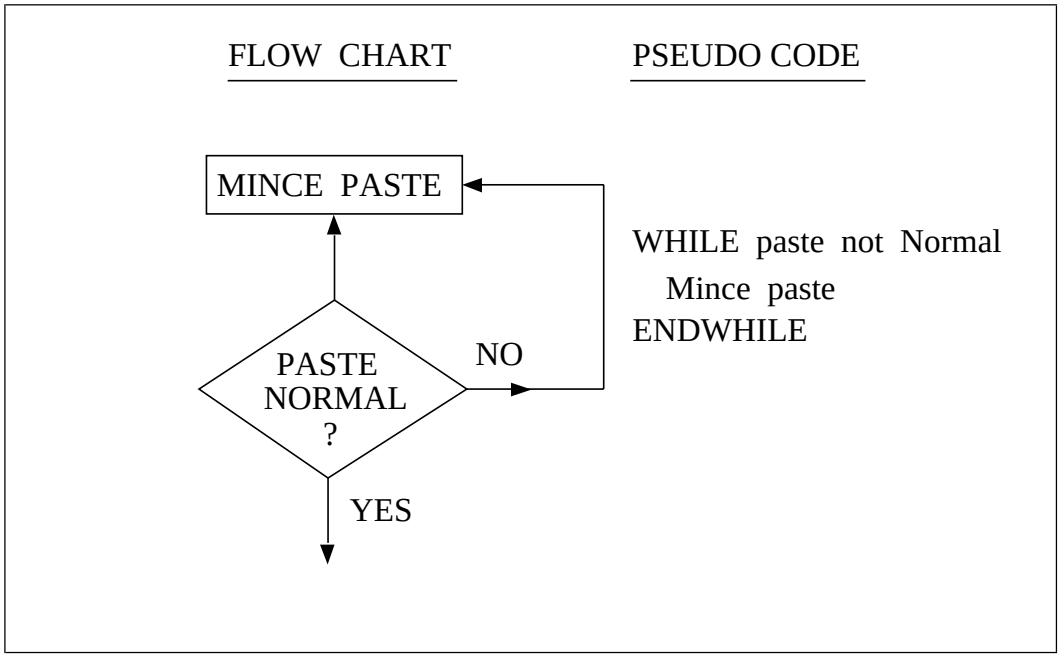

Figure 1.35 Process of repetition and its pseudocode using While - End While loop

Example 1.18

Figure 1.36 illustrates the multichoice selection or the CASE structure. In CASE structure selection can be made out of many choices using the word CASE. CASE structures are used in evaluating situations that can have a number of different results. CASE in this sense refers to a refinement of a basic IF-THEN-ELSE type of conditional structure (IF A is true, then

do B), but a CASE structure functions more like a series of nested IFS (IF A, then do this; if B then do that...). For example if you have three types of drinks, then the flowchart shown in Figure 1.36 gives a method of selecting any one of them. This can also be written in pseudocode form using CASE statement.

An equivalent Pseudocode of the flowchart shown in Figure 1.3.6 is given below:

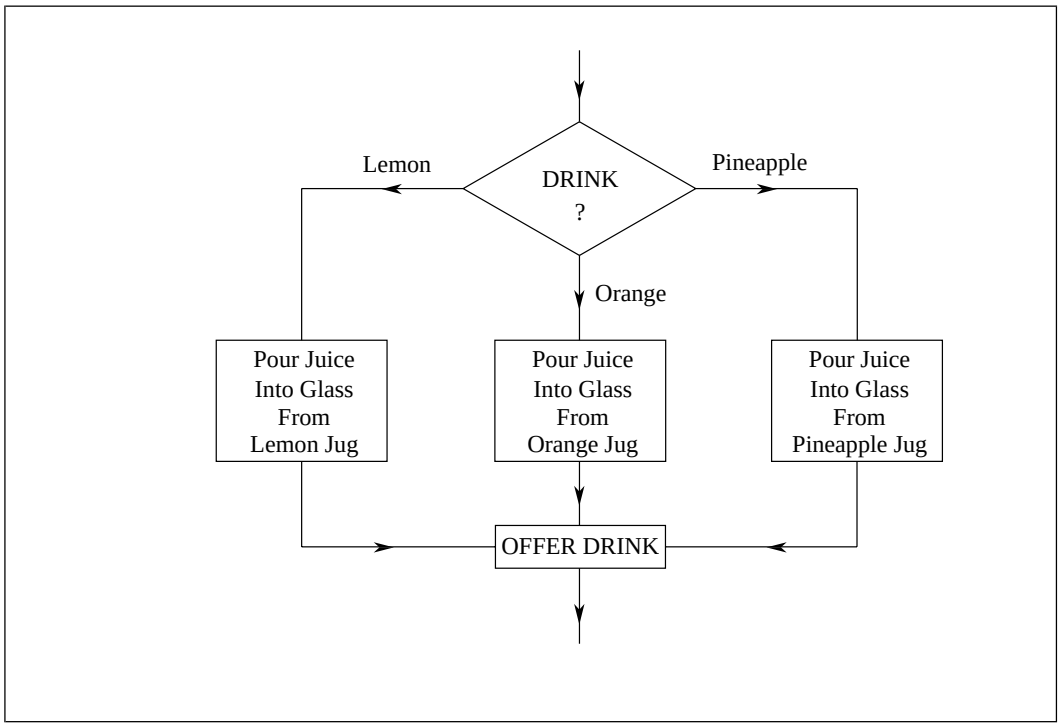

Figure 1.36 Flowchart for case or multichoice selection

CASE drink
Lemon
  Pour juice into glass from Lemon jug.
Orange
  Pour juice into glass from Orange jug.
Pineapple
  Pour juice into glass from Pineapple jug.
END CASE

REPEAT - UNTIL LOOP Structure

Example 1.19

In this example REPEAT - UNTIL loop structure is used. This loop structure is an alternate to While - End While Loop.

Suppose you want to insert a nail in the wall by hitting its head with a hammer. Then the flowchart representation and its equivalent pseudo code will be as follows:

Pseudo code for Figure 1.37
REPEAT
Hit head of nail
UNTIL nail fully in

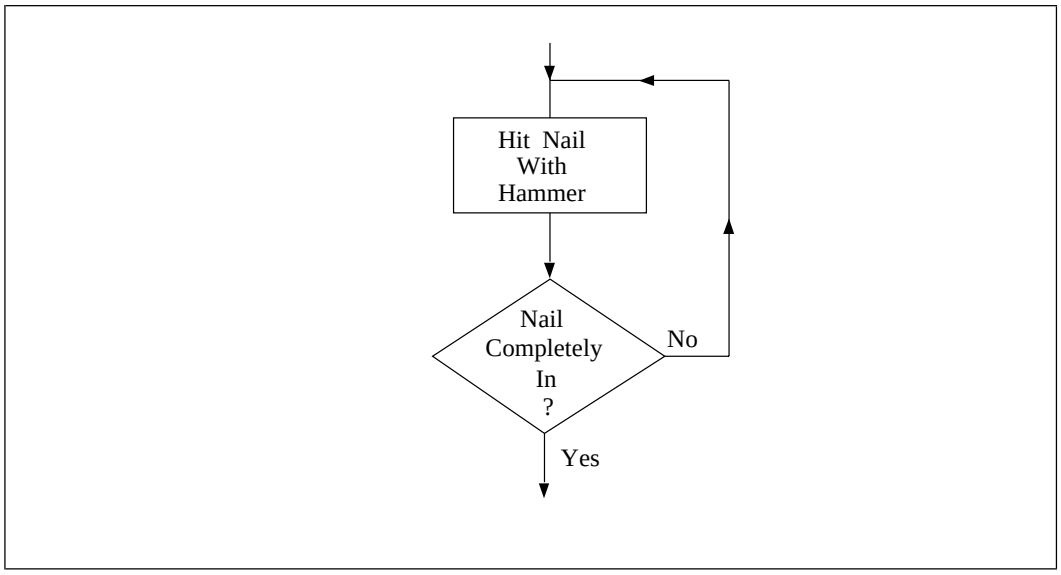

Figure 1.37 Flowchart for repeat-until loop of inserting a nail in the wall

An equivalent of the above flowchart using WHILE - END WHILE structure will be as follows:

WHILE Nail is not fully in
Hit nail head with hammer
END WHILE
Flowchart for this pseudocode is shown in Figure 1.38.

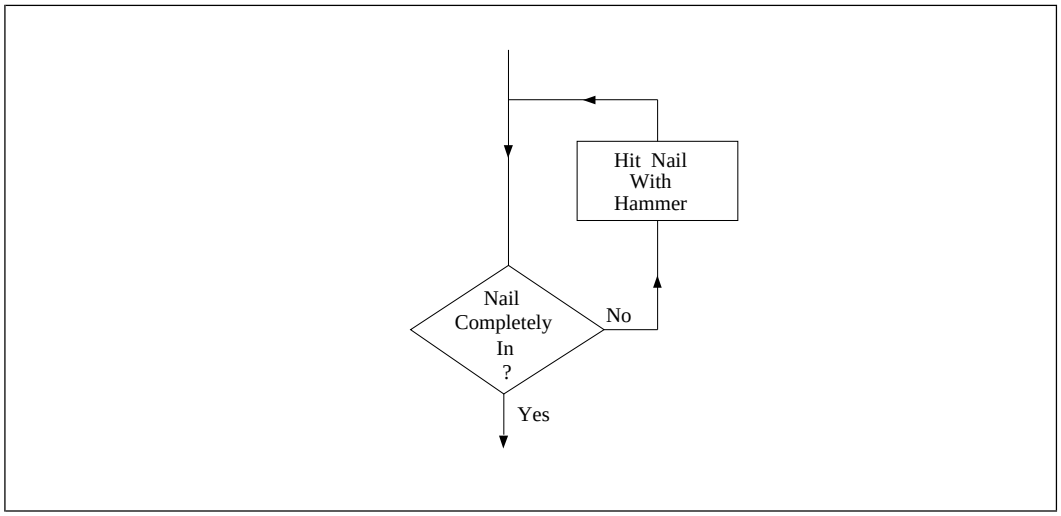

Figure 1.38 Flowchart for While-End While loop of inserting a nail in the wall

Exercise 1.3

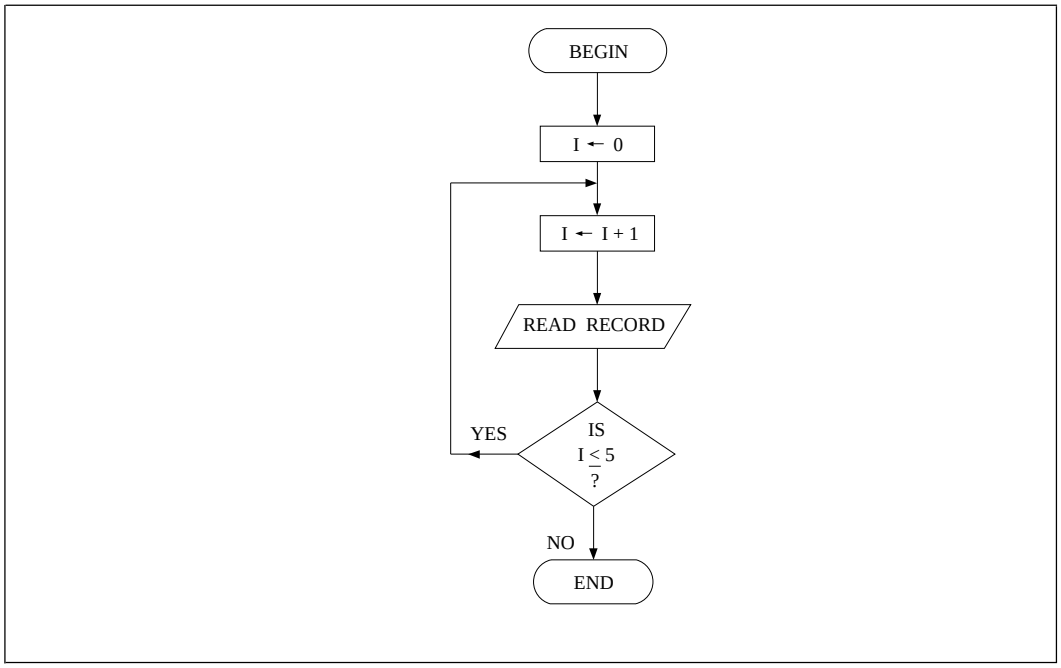

Figure 1.39 Flowchart for Q. N. 1

Q.1 A certain file contains 5 data records. Which of the

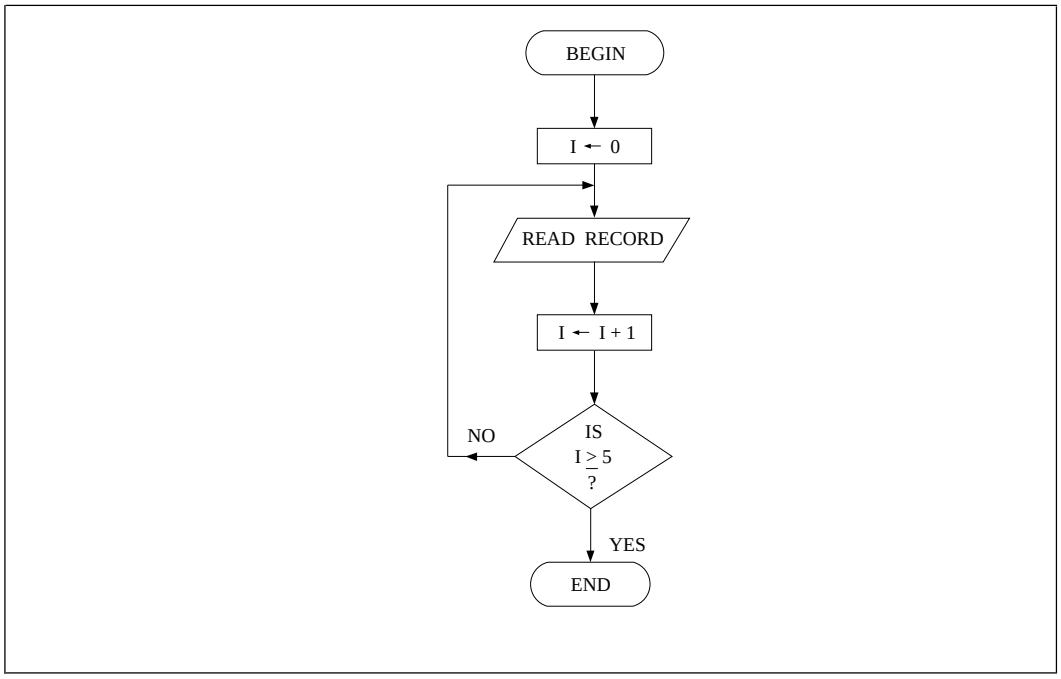

Figure 1.40 Flowchart for Q. N. 1

flowcharts shown in Figures 1.39 and 1.40 will read 5 and only 5 data records?

Q 2. How many records will be read by each of the following flowcharts? (see Figure 1.41 and 1.42)

Q 3. Consider the flowchart shown in Fig.1.41.

Find out the output if the input values of X,Y and Z are as follows.

(a) X=7, Y=9, Z=11

(b) X=11, Y=9, Z=7

(c) X=9, Y=11, Z=7

Q 4. Draw a flowchart that interprets the following statements.

(a) If SALES is 10,000 or more then
BONUS= 3% of SALES

(b) If SALES is 10,000 or more then
BONUS=3% of SALES ELSE
BONUS=2% of SALES.

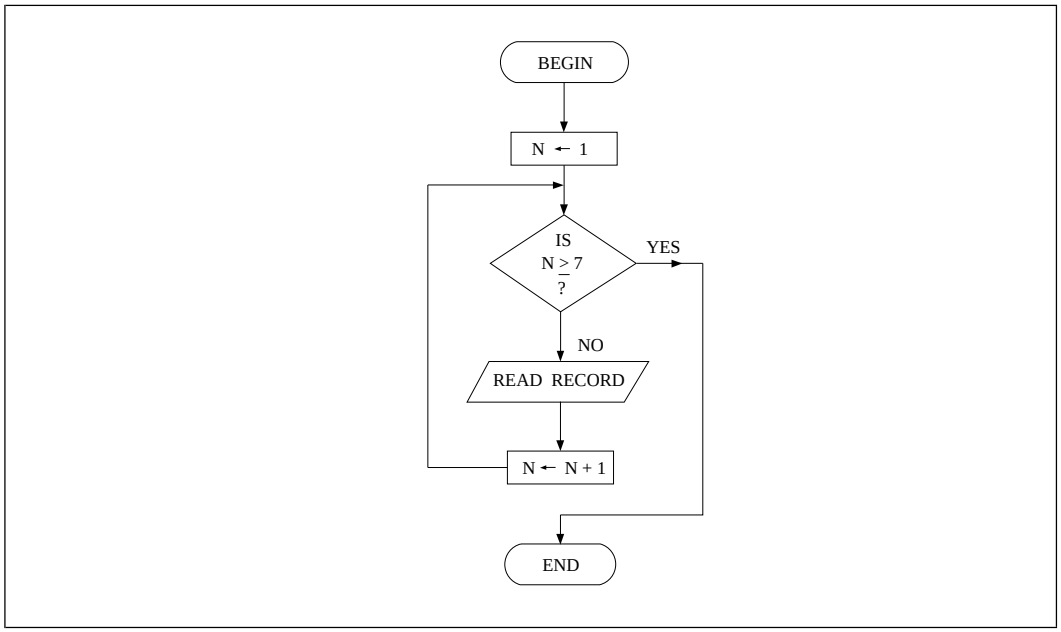

Figure 1.41 Flowcharts for Q N. 2

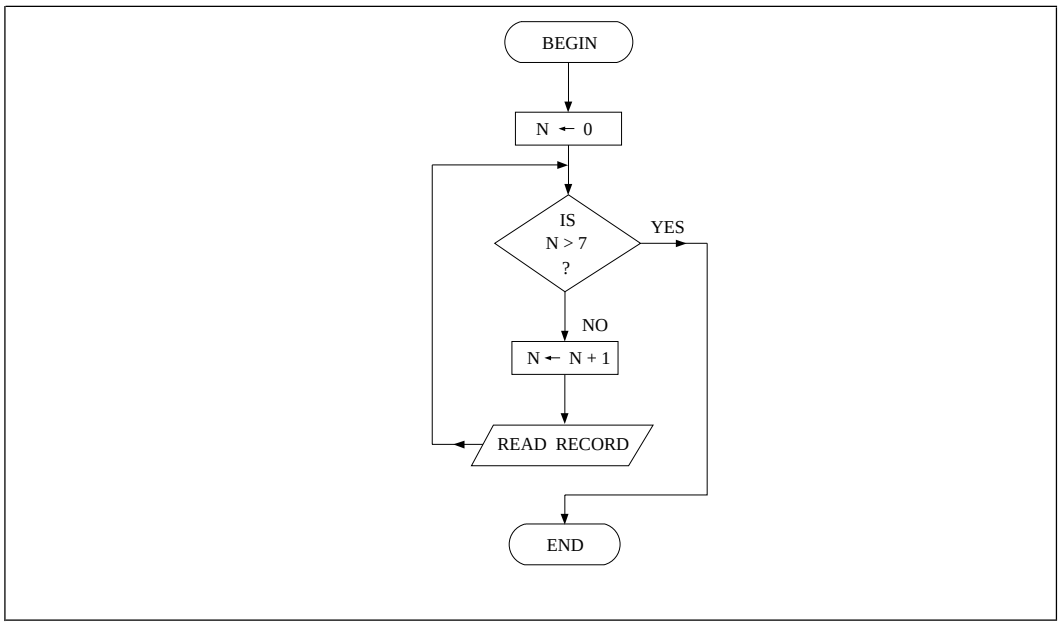

Figure 1.42 Flowchart for Q. N. 2

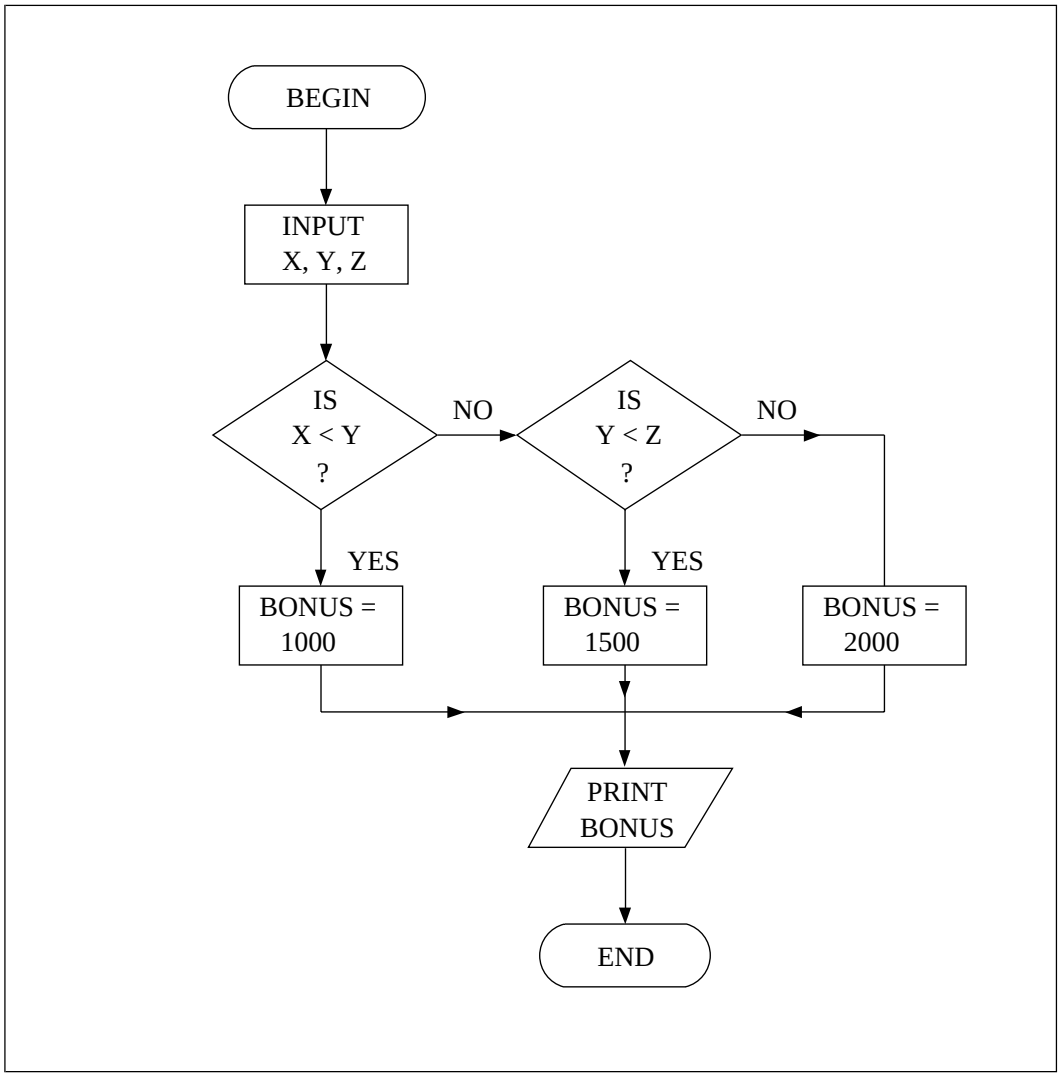

Figure 1.43 Flowchart for Q.N.3

Q 5. What is the value of SUM for the flowchart shown in Figures 1.44 and 1.45.

Q 6. What is an algorithm? Write algorithms for each of the following:

(a) Formatting a disk in your computer.

(b) Finding a book in the library.

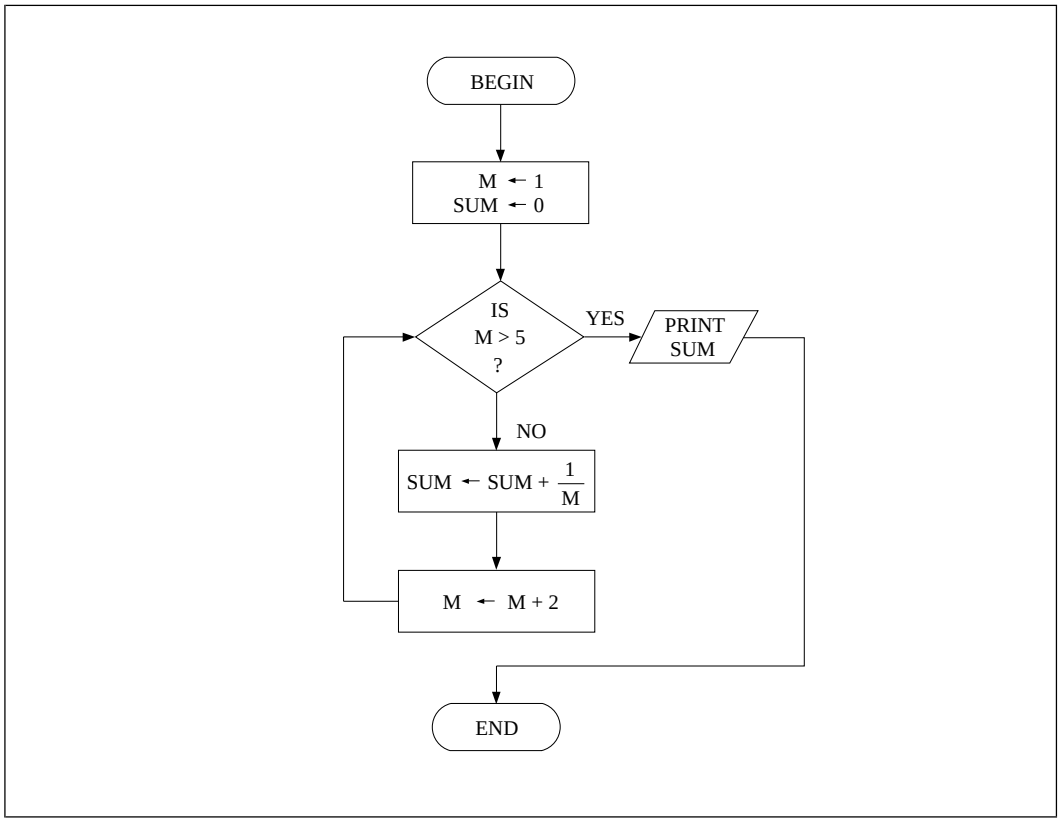

Figure 1.44 Flowchart for Q No. 5

Q 7. The following steps are written to calculate the standard deviation for any given numbers. Use this algorithm to calculate the standard deviation for the numbers 8, 10, 12, 16, 19:

1. Find the sum of the list of numbers.
2. Divide the sum by the total number of numbers to obtain the mean.
3. Subtract the mean from each number in the list and square this result.
4. Calculate the sum of the squares found in step 3.
5. Divide the sum of step 4 by the total number of numbers in the original list.
6. Find the square root of the result found in step 5.

This number is known as the standard deviation of the original list.

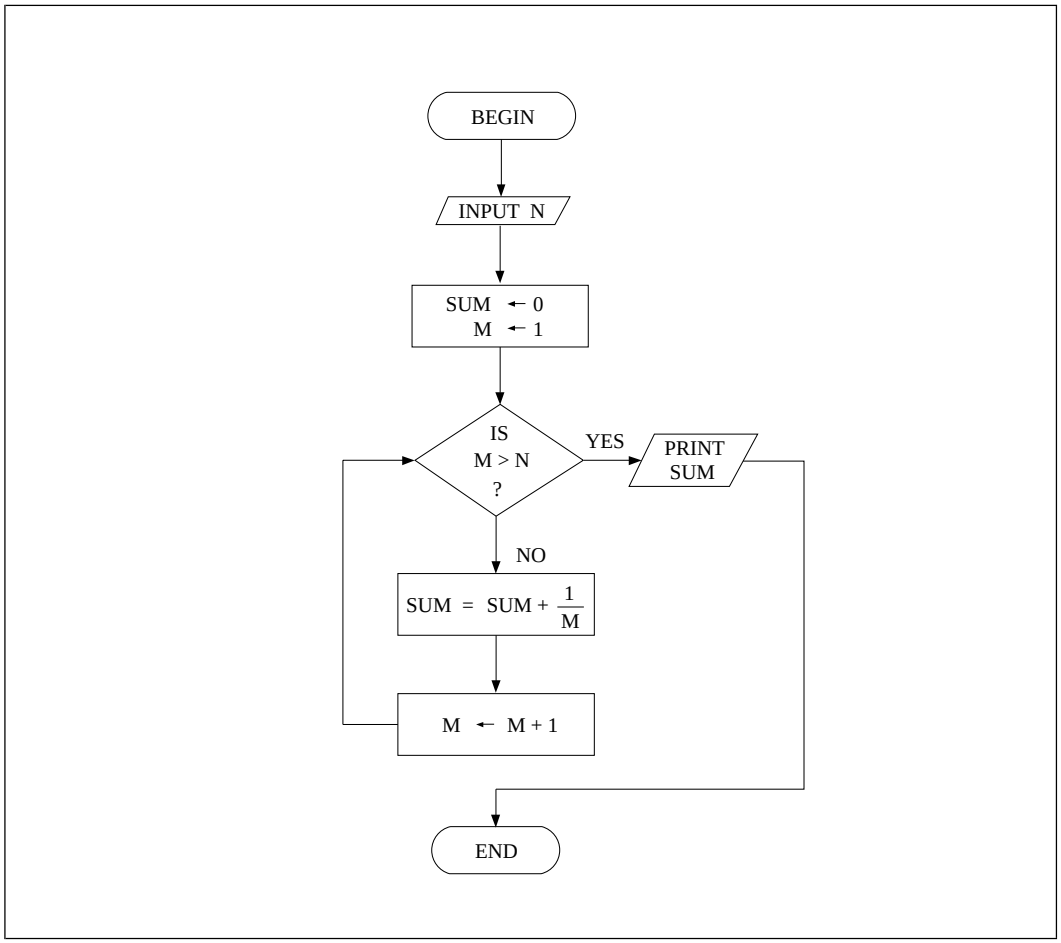

Figure 1.45 Flowchart for Q No. 5

Q 8. Write an algorithm to compute a sales person's commission based on the following table:

Amount Sales	**Commission (% of sales)**
Under Rs. 500	2%
Rs. 500 or more but under Rs. 5000	5%
Rs. 5000 and above	10%

Q 9. Draw the flowchart for the following pseudocode:

BEGIN
Input mortgage amount
IF amount < 25,000

THEN
down payment = 3% of amount
ELSE
payment 1 = 3% of 25,000
payment 2 = 5% of (amount - 25,000)
down payment = payment 1 + payment 2
END IF
print down payment

Q 10. Salesman's commission is calculated using the following Pseudocode. Use this for writing a flowchart.
Input sales
IF sales < 500
THEN
Commission = 2% of sales.
ELSE
Commission = 5% of sales.
END IF
Print Commission

Q 11. Convert the flowchart shown in Figure 1.46 into a pseudo-code for finding the largest of three numbers.

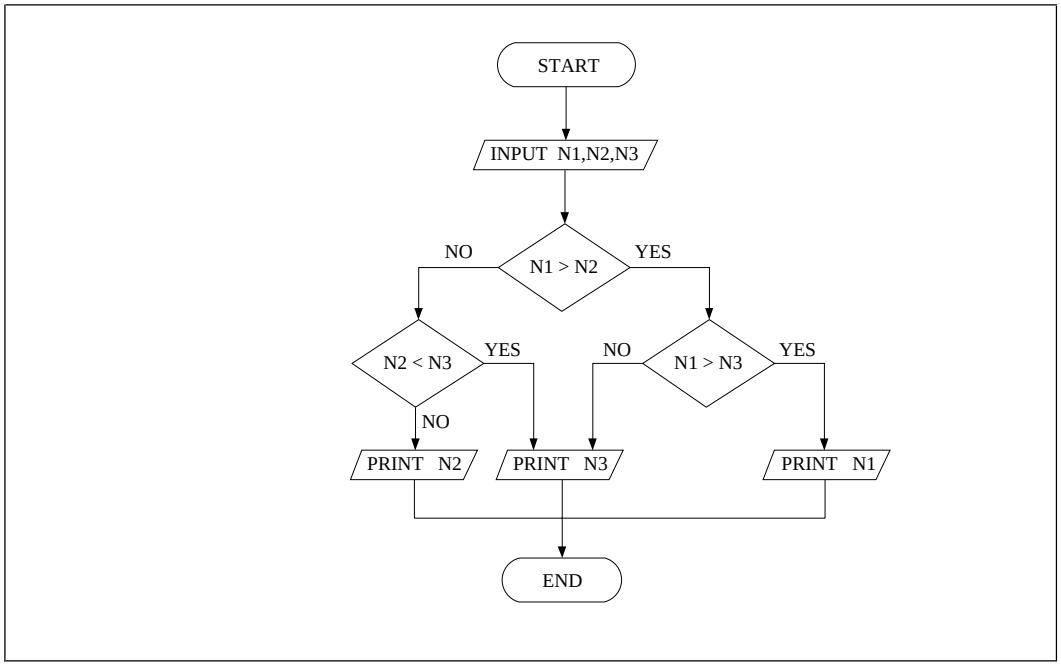

Figure 1.46 Flowchart for Q No. 11

Q 12. The pseudo code for the year and bonus is given below. Draw a flowchart corresponding to this pseudo code.

```
INPUT    employee number, pay, position code & years.
IF       position code = 1
THEN     set bonus to 1 week's pay
ELSE IF position code = 2
        THEN
           IF 2 weeks pay > 700
           THEN
             set bonus to 700
           ELSE
             set bonus to 2 week's pay
           END IF
        ELSE
             set Bonus to 1.5 week's pay.
        END IF
IF year greater than 10
THEN
 Add 100 to bonus
ELSE
 IF years less than 2
 THEN
      cut bonus to half
 ELSE
      bonus stays the same.
 END IF
END IF
Print employee number & bonus.
```

1.7 Programming Methods

1.7.1 Top-down Design

Top-down design is the technique of breaking down a problem into the major tasks to be performed. Each of these tasks is then further broken down into separate *subtasks*, and so on until each subtask is sufficiently simple to be written as a self contained module or procedure. The program then consists of a series of simple modules.

In top-down design we initially describe the problem we are working on at the highest, most general level. The description of the problem at this level will usually be concerned with what must be done – not how it must be done. The description will be in terms of complex, higher-level operations. We must take all of the operations at this level and individually break them down into simpler steps that begin to describe how to accomplish the tasks. If these simple steps can be represented as acceptable algorithmic step, we need not refine them any further. If not, we refine each of these second level operations individually into still simpler steps. This stepwise refinement continues until each of the original top-level operations has been described in terms of acceptable shortest (primitive) statements.

Advantages of Top-down approach

(a) It allows a programmer to remain "on top of" a problem and view the developing solution in context. The solution always proceeds from the highest levels down. With other techniques we may find ourselves bogged down with very low-level decisions at a very early stage. It will be difficult to make these decisions if it is not clear how they may affect the remainder of the problem.

(b) It is a very good way to delay decisions on problems whose solution may not be readily apparent. At each stage in the development, the individual operation will be refined into a number of more elementary steps. If we are not sure how to proceed with step 1 we can still work on step 2.

(c) By dividing the problem into a number of subproblems, we have made it easier to share problem development. For example, one person may solve part 2 of the problem and the other person may solve part one of the problem.

(d) Since debugging time grows so quickly, it will be to our advantage to debug a large program as a number of smaller units rather than one big chunk. The top-down development process specifies a solution in terms of a group of smaller, individual subtasks. These subtasks thus become the ideal unit of program testing and debugging.

By testing the program in small pieces, we greatly simplify the debugging process. In addition, we will have the satisfaction of knowing that everything we have coded so far is correct. When we add a new piece of code, say "p" to the

overall program "P", and an error condition occurs, we can definitely state that the error must either be in "p" itself or in the interface between "p" or "P", because "P" has been previously checked and certified.

(e) Another advantage of the top-down development process is that it becomes an ideal structure for managing the implementation of a computer program using teams of programmers. A senior programmer can be responsible for the design of a high-level task and the decomposition into subtasks. Each of those subtasks can then be "farmed out" to a more junior programmer who work under the direction of the senior staff. Since almost all software projects are done by teams of two or more programmers, this top-down characteristic is very important.

In summary, top-down programming is a program design technique that analyses a high-level problem in terms of more elementary subtasks. Through the technique of stepwise refinement we then expand and define each of these separate subtasks until the problem is solved. Each subtask is tested and verified before it is expanded further. The advantages of this technique are:

(a) Increased intellectual manageability and comprehension.
(b) Abstraction of unnecessary lower-level detail.
(c) Delayed decisions on algorithms and data structures until they are actually needed.
(d) Reduced debugging time.

1.7.2 Bottom-up Design and Implementation

When faced with a large and complex problem, it may be difficult to see how the whole thing can be done. It may be easier to attack parts of the problem individually, taking the easier aspects first and thereby gaining the insight and experience to tackle the more difficult tasks, and finally to try and bolt them all together to form the complete solution. This is called a bottom-up approach. It suffers from the *disadvantage* that the parts of the program may not fit together very easily. There may be a lack of consistency between modules, and considerable re-programming may have to be done.

1.7.3 Modular Design and Programming

In industry and commerce, the problems that are to be solved with the help of computers need thousands or even more number of lines of code. The importance of splitting up the problem into a series of self-contained modules then becomes obvious. A module should not exceed about 100 or so lines and should preferably be short enough to fit on a single page.

Advantages of modular design

(a) Some modules will be standard procedures used again and again in different programs or parts of the same program.
(b) Since module is small, it is simpler to understand it as a unit of code. It is therefore easier to test and debug, especially if its purpose is clearly defined and documented.
(c) Program maintenance becomes easier because the affected modules can be quickly identified and changed.
(d) In a very large project, several programmers may be working on a single program. Using a modular approach, each programmer can be given a specific set of modules to work on. This enables the whole program to be finished sooner.
(e) More experienced programmers can be given the more complex modules to write, and the junior programmers can work on the simpler modules.
(f) Modules can be tested independently, thereby shortening the time taken to get the whole program working.
(g) If a programmer leaves part way through a project, it is easier for someone else to take over a set of self contained modules.
(h) A large project becomes easier to monitor and control.

1.8 Structured Programming

In the 1960s in the USA a number of surveys confirmed what most data processing managers had believed for a long time – that there is a substantial variation in programmer abilities and that too much time is spent on debugging programs and on maintenance activities. The surveys generated much controversy, but the effect they had was dramatic. Suddenly everyone became concerned with the programmer's productivity and the

way in which he actually programmed. In 1965, Professor Dijkstra of Eindhoven University in Holland presented a paper at the IFIP Congress in New York suggesting that the GOTO statement should be eliminated from programming languages altogether, since a program's quality was inversely proportional to the number of GOTO statements in it. In the following year, Bohm, and Jacopini showed that any program with single entry and exit points could be expressed in terms of three basic constructs:

(a) Sequence or carrying out a process

(b) Iteration or looping

(c) Selection or decision taking

This was the beginning of structured programming. This dramatically improved the quality of programming and of programmer productivity. The above three constructs have been explained in sections 1.2 and 1.6.2.

There is no doubt that structured programming has been successful, but it does not solve all our problems. Poorly constructed system designs can still negate the benefits provided by structured programming. It was not surprising then that similar principles were applied to the tasks of analysis and design and a full range of structured methods came into being.

1.8.1 Why Structured Programming ?

In computer programming, the spaghetti code is the method of coding that confuses the program flow because of the excessive use of GOTO or jump statements. Hence one of the major improvements has been the shift from spaghetti code to TOP-Down modular design and structured programming methods. The reason for the evolution to structured programming is the need for well organized programs that are ultimately easier to :

(a) Design

(b) Read and understand

(c) Modify

(d) Test and Debug

(e) Combine with other programs

Writing programs that are clear for any programmer to read and understand is a very important consideration, particularly if there is a change in programming personnel.

1.8.2 Characteristics of Structured Programs

In general terms, structured programming is the development of computer programs that are well organized. Here is a list of characteristics of structured programs.

(a) The programs are based on top-down modular design. In other words, the problem at hand is analysed or broken down into major components, each of which is again broken down if necessary. Therefore, the process involves working from the most general down to the most specific. The design of the modules is reflected in hierarchy charts such as the one shown in Figure 1.47.

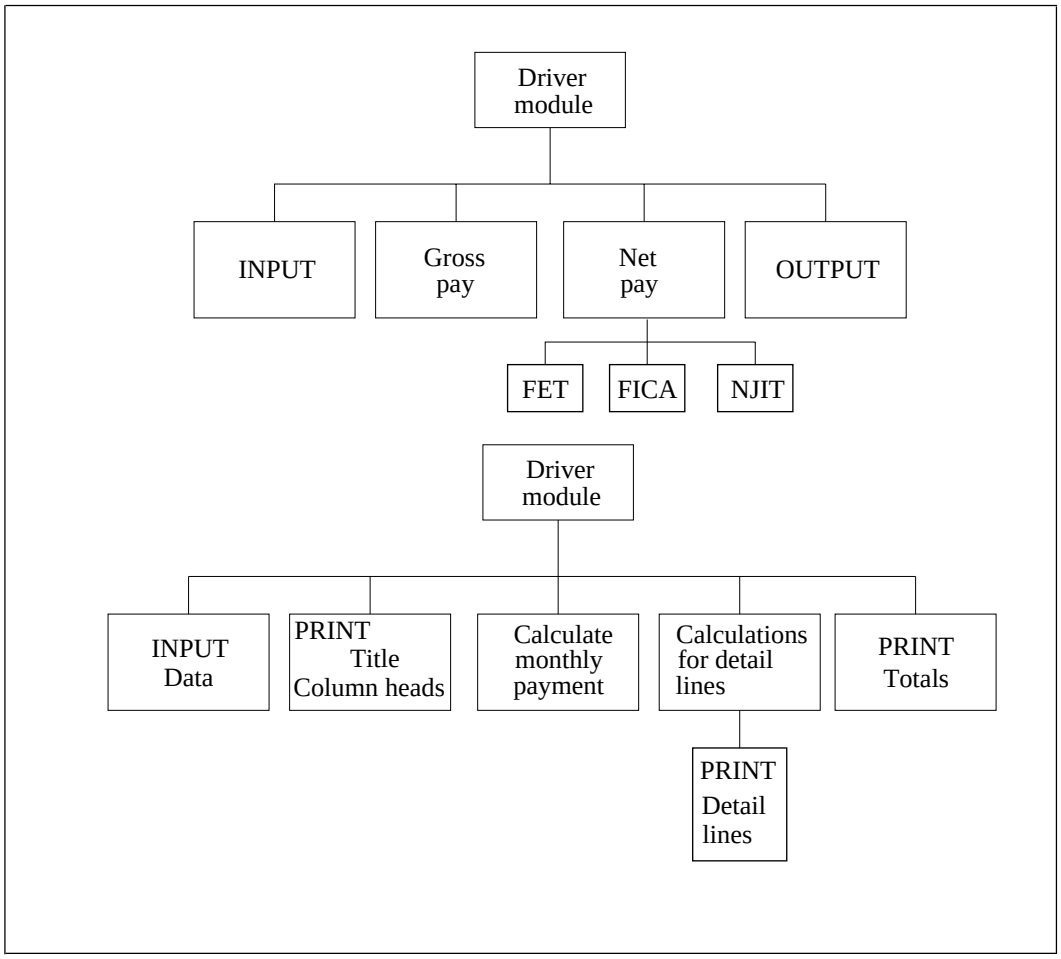

Figure 1.47 Top-Down modular Design

(b) Each module has one entry point and one exit point. The GOTO statement is never used to jump from one module in the program to another. (Pure structured programs do not use GOTO). The following program in BASIC is an example of structured programming:

```
340 REM   * * * * * * *   INPUT DATA   * * * * * * * * * * * *
350   PROMPTS WILL SHOW ON THE SCREEN – DATA WILL BE ENTERED
          AT KEYBOARD
360   CLS     : ' CLEAR SCREEN
370   OK$ = "N"
380   WHILE OK$ = "N"
390     INPUT "ENTER THE EMPLOYEE NAME"; EMPNAME$
400     INPUT "ENTER THE HOURS WORKED"; HOURS
410     INPUT "ENTER THE RATE PER HOUR"; RATE
420     INPUT "ENTER THE NUMBER OF EXEMPTIONS"; EXMP
430     INPUT "ENTER THE SALARY TO DATE"; SALDATE
440     PRINT: PRINT
450     INPUT "ENTRIES OK (Y OR N)"; OK$
460     WHILE OK$ <> ''Y'' AND OK$ <> ''N''
470       INPUT "PLEASE ENTER A 'Y' OR AN 'N' " ; OK$
480     WEND
490   WEND
500 RETURN
```

(c) A rule of thumb is that the modules should not be more than one half page long. If they are longer than this, they should preferably be split into two or more submodules.

(d) Two-way decision statements are based on IF..THEN, IF..THEN..ELSE, and nested IF statements.

(e) Loops are not custom designed with the use of the GO TO statement, but are based on the consistent use of WHILE..WEND and FOR..NEXT. In WHILE..WEND, the loop is based on the truth of a condition, and in FOR..NEXT, on a counting process, and the number of repetitions that can easily be predicted. (See Figure 1.48)

1.8.3 Other Characteristics of "GOOD" Programs

Readability

The ease with which some one can read a program is based on the

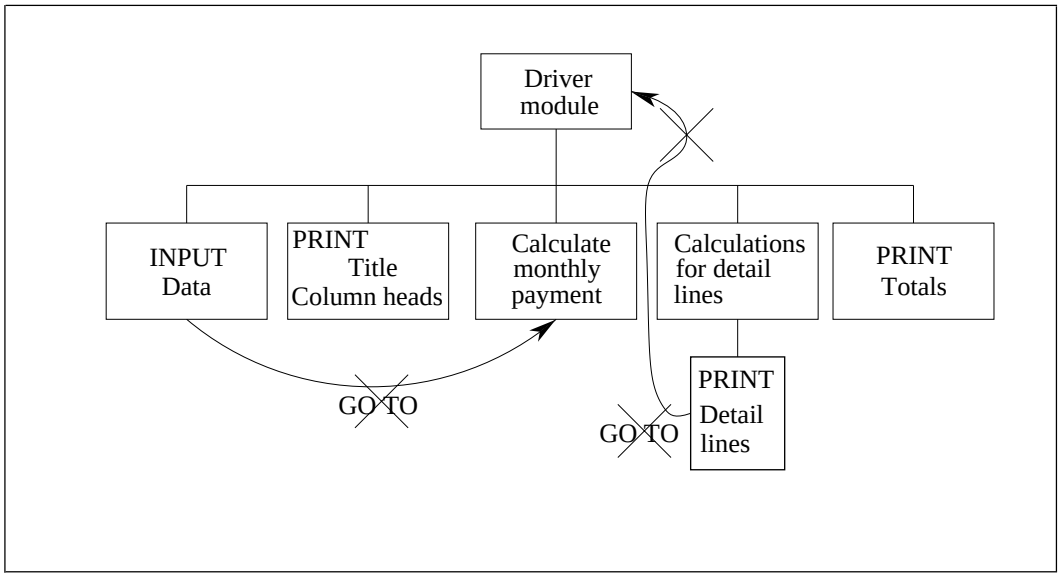

Figure 1.48 GO TO statements are removed

manner in which it has been typed. This includes printing gaps between the modules.

Documentation

The documentation is greatly improved if the variable choice is descriptive. This serves as a mnemonic device to aid in remembering the meaning of the variables.

Efficiency

The speed of execution is something to consider, but it is not a primary concern when using a particular programming language. In terms of the manipulation of numbers, integers are the fastest, followed by single precision, with double precision being the slowest.

1.8.4 Importance of Structured Programming

Structured programming is important for the following reasons:

(a) It is much easier for students to debug structured programs and for instructors to grade them.

(b) Students who learn top-down modular design retain what they have learned over a much longer period of time.
(c) Students who go on from structured BASIC to PASCAL and/or structured COBOL are better prepared for these languages.

1.9 Modularity

One way to improve the structure of a program is to break down the original problem to be solved into independent tasks (*called *modules* in pseudocode and *subroutines* in BASIC) and then execute these modules in a predefined sequence. The resulting design consists of a network of modules.

Breaking down a problem into smaller pieces (modules) allows you to focus more easily on a particular piece of the problem without worrying about the overall problem. The tasks become easier to solve and are more manageable since each performs a very specific function. Typically, you can code each module independently of the others and test or debug each module separately. Once all modules are working properly, you can link them together by writing the coordinating code (generally called the root segment or the main code). The coordinating code activates the various modules in a predetermined sequence. Consider Figure 1.49, which illustrates these principles. Note that the resulting program consists of five modules: the coordinating module and the four modules A, B, C, and D. It is, of course, conceivable that a particular module itself might be broken down into other modules.

By breaking down a problem into independent modules, a programmer in charge of a very complex problem can easily assign various members of a team the responsibility for developing one or more modules. Such modules can be run and tested independently of one another. Making one change in one module is a very local intervention that does not require an understanding of all other modules. In contrast, making one change in a nonmodular program held together by myriads of GOTO statements not only requires an understanding of the whole program but can also produce masses of error messages and a lot of unwanted output.

Another advantage of a modularized design is that you will find yourself using the GOTO statement less frequently. This in

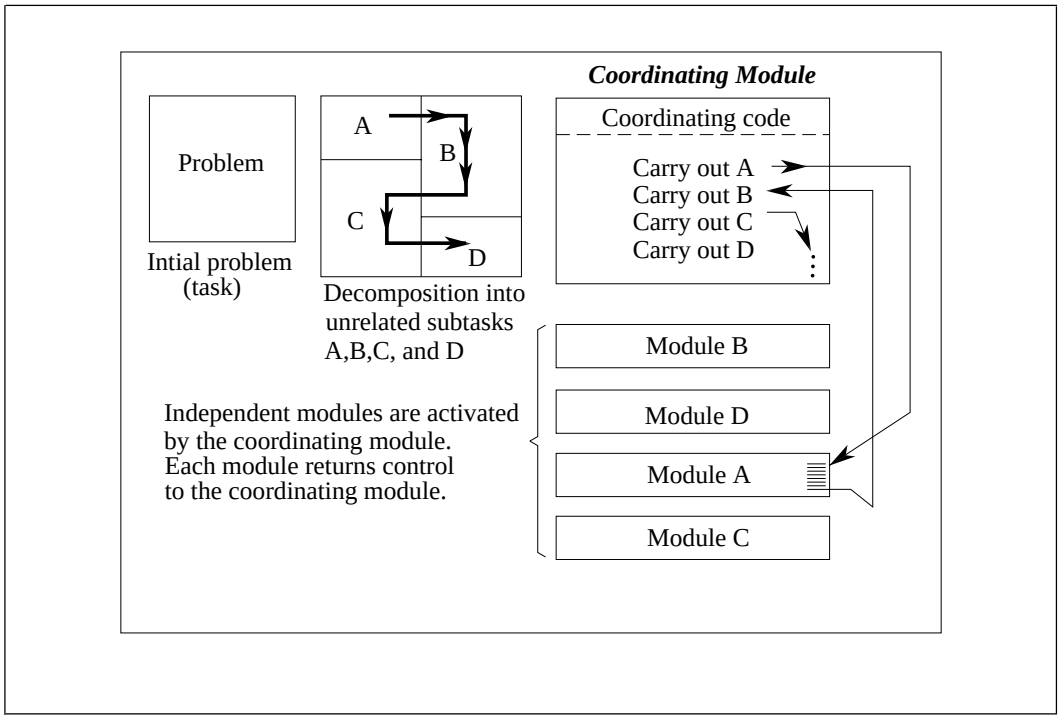

Figure 1.49 Program Decomposition

itself is very important, since the undisciplined use of the GOTO statement can often give rise to the "spaghetti code syndrome", which makes it very difficult for the program reader to remember where in the program he/she is coming from and where he/she is going!

In a modular environment, each module can be respecified as a sequence of smaller modules describing what is to be done at increasing levels of detail. The technique of expanding a program plan into several levels of detailed subplans and presenting the program structure as a hierarchy of tasks is sometimes referred to as *top-down* design. A hierarchy chart, sometimes called a structured diagram, is a useful tool for illustrating module relationships and hierarchies. Figure 1.50 shows a hierarchy chart for the problem shown in Figure 1.49

In summary, modularization helps the programmer write better structured programs that are generally more compact and thus easier to work with and more readable. A subroutine can also reduce the code required for writing tasks that are to be performed repeatedly at different places in a program.

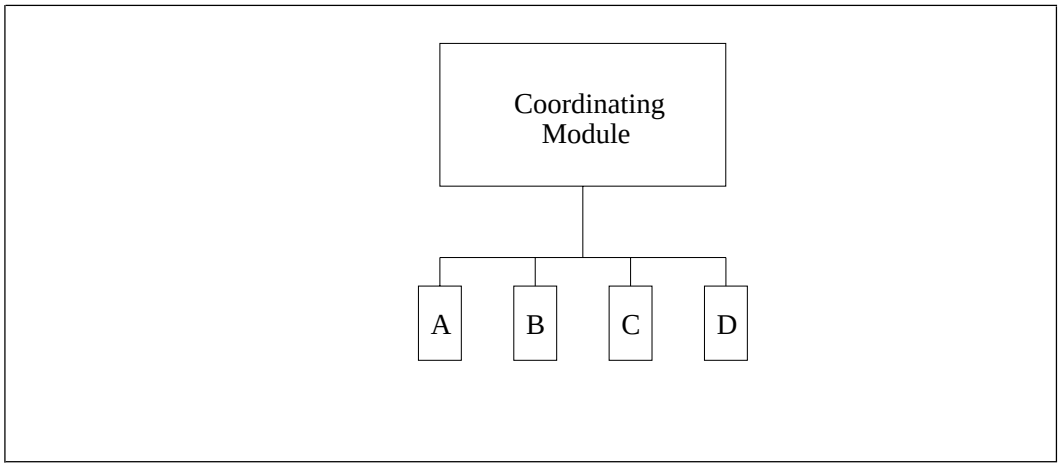

Figure 1.50 A Hierarchy Chart

1.10 Programming Tools

These are the programs specially designed for the benefit of programmers so that they can formulate the instructions to solve a problem properly and easily. One of the important tools is editor. This is explained in the following paragraphs.

1.10.1 Editors

An editor is a piece of software which enables the programmer to enter and edit a program. The different kinds of editors are as follows:

(a) Screen editor
(b) Line editor.
(c) Linkage editor
(d) Text editor

Screen editor

A screen editor allows the programmer to position the cursor anywhere on the screen and insert or delete text.

Line Editor

A line editor allows lines of text to be entered and edited only

by entering commands which refer to a particular line or block of text. This type of editor was used in the days before VDU screens were common, when the lines typed in were printed out on a piece of paper as well as being stored in the memory. An example of a line editor is EDLIN which is supplied with MS-DOS. This editor is suitable for typing in small amounts of text such as batch files. Typical commands include:

Line Number	*Edits the line specified*
A	Appends the line
D	Delete line
I	Insert line
L	List text
E	End editing

Linkage Editor

The linkage editor is a utility program which adapts a program that has just been assembled or compiled into a particular computer environment. It formally links cross references between separate program modules, and it links the program to various libraries containing prewritten subroutines. The output of a linkage editor is a load module, a program ready to run in the computer.

Text Editor

A text editor is a software program that creates and manages text files. Text editors are used to create and edit source language programs, data and text files. Unlike word processors, text editors do not have elaborate formatting and printing features. For example, there is usually no automatic word wrap or underline, bold face and italics printing in text editors.

1.10.2 Other Program Development Aids

Some of the other aids for a programmer may include

(a) An on-line help facility giving information about any command, standard function or procedure.
(b) Utility programs enabling the user to design input screens and have the code automatically generated.

(c) Separate compilation of modules, allowing the programmer to build up a precompiled tested code.
(d) An integrated development environment combining a text editor and a compiler with pull down menus, windows, help facility and debugger.
(e) Application generators–These are the software programs that generate application programs from a description of the problem rather than from detailed programming. Application generators are one or more levels higher than the programming language, whose source code they generate, but still require the user to input basic mathematical expressions in order to state complex processing on their business data. For example, a complicated pricing routine will require that the pricing algorithms be stated just as they would be in any programming language.

1.10.3 Types of Program Errors

Once a program has been typed in, different types of errors may show up. These include:

(a) Syntax/semantic errors
(b) Logic errors
(c) Runtime errors

Syntax/semantic errors

Syntax is a set of rules governing the structure of and relationship between symbols, words and phrases in a language statement. A syntax error occurs when a program cannot understand the command that has been entered.

Logic errors

Logic refers to a sequence of operations performed by a software or hardware. Software logic or program logic is the sequence of instructions in a program. Logic errors are the errors that have been entered in the instructions created because of the mistake made by a programmer. Suppose you wanted to do the sum of A and B and put the result in the variable C. This is accomplished by typing:

```
C := A + B
```

But while typing, the programmer has typed the following expression:

```
C := A - B
```

Such a program will run properly but will give erratic result because the value of C will not be the *sum* of A and B but it will be the *difference* of A and B which is quite different. Such errors can only be detected with the help of test data that will give the input values of A and B and the resultant value that should come out of the program as a result of the operation performed on A and B. If the result given by the computer and the result calculated manually with the help of a *calculator* are the same, then you can say that there is no logical error. Otherwise a logic error may be present in the program.

Runtime error

Runtime errors occur when a program is run on the computer and the results are not achieved due to some misinterpretation of a particular instruction. This could be some thing like dividing a number by zero which results in a very large value of quotient. It may also be any other instruction which a computer is not able to understand. To overcome this problem, there is a built-in error detector in the language interpreter or compiler which will give the message and that will reflect the reason for the run time error.

1.10.4 Program Testing

It is the job of the programer to test, as far as possible, that all parts of the program work correctly. It should be realised that complete testing is not possible except in the case of the most trivial program; one can never be completely certain that all errors have been removed, but sufficient tests can be performed to give a reasonable measure of confidence in the program.

The situation is analogous to testing children on multiplication tables; once a child has answered a certain number of multiplication table tests correctly, at some point or other the teacher assumes that all other tests will be answered correctly and testing ceases.

1.10.5 Designing a Test Plan

Good testing requires the following:

(a) A thorough knowledge and understanding of what the program is supposed to do.
(b) Plan out in advance what ought to be tested.
(c) To work out expected results for each of the test cases.
(d) Writing out the test plan.

Since we cannot test everything, each test must be carefully planned to provide more information about the program. A major benefit of preparing a comprehensive test plan with expected results is that it forces the programmer to think carefully about the program and often errors are spotted even before running the test.

1.10.6 Methods of Testing

The objectives of testing can be stated in two basic questions:

(a) Does the logic work properly? This means, answering the following:
  (i) Does the program work as intended?
  (ii) Can it be made to crash?
(b) Is the necessary logic present? This means answering the following:
  (i) Are there any functions missing?
  (ii) Does the program or module do everything specified?

There are two different ways of testing. These are:

(a) Functional testing
(b) Logical or structural testing

1.10.7 Functional Testing

Functional testing is carried out independently of the code used in the program. It involves looking at the program specification and creating a set of test data that covers all the inputs and outputs and program functions. This type of testing is also known as "*black box testing*". For example, to test the program that calculates check digits we could draw up the following test plan:

Table 1.1 Test Plan for Check Digits

Serial Number	Test for numerical data	Purpose	Expected result	Actual result
1	Enter 1234	Test validity of data	1234 Digits	3
2	Enter 8 digits	Test extreme case	are printed as it is	3
3	Enter 123W	Testing invalid data	Data not accepted	Error

From Table 1.1 we will be able to make out whether the software is testing the data correctly or not. If it is not testing properly, then we may have to correct the program.

1.10.8 Logical (Structural) Testing

Logical testing (*white box testing*) is dependent on the code logic, and derives from the program structure rather than its function. In other words, we study the program code and try to test each possible path in the program at least once. The problem with logical testing is that it will not detect the missing functions.

One method of devising a test plan is to start with a set of functional test cases, and then add additional tests to exercise each statement in the program at least once, making sure that each decision is tested for all outcomes.

1.10.9 Debugging Aids

Once the presence of logic errors has been detected with the help of test runs, there are various ways of finding where the error or errors lie. These include the following:

(a) A dry run through the program, building up a trace table while manually following the program steps.
(b) The inclusion of extra "write" statements to examine the contents of variables at various points in the program.
(c) The printouts of file contents before and after processing.
(d) An 'On line debugger' which allows a programmer to step through the program line by line and examine the values of variables at any point in the program.

1.10.10 Assembler

A program which translates an assembly language program into a machine language program is called an *assembler*. An assembler which runs on a computer for which it produces object codes (machine codes) is called a self assembler (or resident assembler). A less powerful and cheaper computer may not have enough software and hardware facilities for program development and convenient assembly. In such a situation, a faster and powerful computer can be used for program development. The programs so developed are to be run on smaller computers. For such program development a cross assembler is required. A *cross assembler* is an assembler that runs on a computer other than that for which it produces machine codes.

1.10.11 Compiler

A program which translates a high-level language program into a machine language program is called a compiler. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program execution time is more, and occupies a larger part of the memory. It has low speed and low efficiency of memory utilization. If a compiler runs on a computer for which it produces the object code, then it is known as a *self* or *resident compiler*. If a compiler runs on a computer other than that for which it produces object code, then it is called a *cross-compiler*.

1.10.12 Interpreter

An interpreter is a program which translates *one* statement of a high-level language program into machine codes and executes it. In this way it proceeds further till all the statements of the program are translated and executed. On the other hand, a compiler goes through the entire high-level language program once or twice and then translates the entire program into machine codes. A compiler is nearly 5 to 25 times faster than an interpreter. An interpreter is a smaller program as compared to the compiler. It occupies less memory space. It can be used in a smaller system which has limited memory space. The object program produced by the compiler is permanently saved for future reference. On the

other hand, the object code of the statement produced by an interpreter is not saved. If an instruction is used next time, it must be interpreted once again and translated into machine code. For example, during the repetitive processing of the steps in a loop, each instruction in the loop must be reinterpreted every time the loop is executed.

1.10.13 Data Description Language

Data description language(DDL) is a language used to define data and their relationships to other data. It is used to create files, databases and data dictionaries.

Hierarchical and network database management packages are used in large computers such as mainframes or mini computers. In these systems, large database packages are controlled by the database administrator whose job is to identify the logical relationships that exist in an organization's records. A special data description language is used to retrieve the stored information. Programming skills are needed by those who work with these software products and DDL helps them to do the job more efficiently.

1.11 What is Program Maintenance?

User requirements from a specific program do not remain static. They change frequently in response to such factors as new laws, new ideas, new products, or new computer facilities. Program maintenance covers a wide range of activities including correcting coding and design errors, updating documentation and test data, and upgrading user's support. In other words, maintenance may be viewed as enhancement i.e. adding, modifying, or developing the code to support changes in the specifications.

1.11.1 Why Program Maintenance?

Although software does not wear out like a piece of hardware, it "ages" and needs to be updated for the following reasons:

(a) Errors are found that were not detected when the system was tested initially. Even though the system was thoroughly tested, errors or bugs often appear after the system has been in use for some time.

(b) A new function may have to be added to the system. For example, the school management had initially asked for the preparation of monthly report of those students who had not paid their fee in time. But at a later date, the auditors or the principal of the school needs a report of those students who are defaulters in paying their fee for more than three months so that their names can be struck from the school. In such a case, the report format will need a change.
(c) Alteration in the original program may cause errors elsewhere in the program and detected later. This will require modifications in the program.
(d) The final reason for program maintenance is that the requirements of the user change with time. For example, programs that produce income tax returns have to be modified almost every year because of changing tax laws.

1.11.2 Problem Areas in Program Maintenance

Two problem areas in program maintenance are:
(a) high cost of software, and
(b) errors caused due to modification in the original program.

High Cost of Software

Major problem with software maintenance is that the program writing is labour intensive in nature. Human beings who write programs are likely to make errors. For example the programmer who was originally assigned the task of solving and coding the program may not be available to modify this program at a later date. Therefore, the new person who is given the job of modification will have to study and understand the logic of the original program and then only he can modify the program. This process may involve more time, money, and there are more chances of making errors. In such cases, it may be easier and economical to rewrite a new program rather than amending the old one.

Errors Caused due to Modification in the Original Program

Alteration in the original program, no matter how slight, must be manually introduced. There is no easy way of making sure that the modifications made will not cause any errors elsewhere in any of the subprograms or the main program. Using the old codes

depend mostly on the programmer's ability to judge what the code can and cannot do. Hence the program is to be thoroughly tested after modification before implementation.

1.11.3 Impact of Software Errors

The quality of a software system depends on its design, development, testing, and implementation. An important aspect of software quality is its reliability. A program is reliable if, when used in a reasonable manner, it does not produce failures that are dangerous or costly. Software errors can cause failure of the system or produce inaccurate results. Programmers should therefore strive to design error free programmes. The software errors are mainly due to design errors or specifications. Hence these are also to be rechecked before studying the impact caused by the errors.

1.11.4 The Problem of Software Modification

The basic problems of modifying software can be attributed to time schedule and the user-need satisfaction. These are as follows:

Time schedule

Modification of a program may not be feasible within the time schedule because of several reasons. The reasons may be non-availability of the right kind of manpower, improper documentation of the old program or testing data. For such cases, the earlier version of the software will have to be used till all the errors are removed in the newer version after it has been tested thoroughly.

User-need Satisfaction

People who are using the program are the final evaluators of the modification. Hence it should be found out whether these users are now satisfied with the new version of the software or not. Further how useful is the new software for them? How enthusiastic are they about the friendliness and ease of working with the new software? Answers to all these questions are to be evaluated and implemented.

1.11.5 Software Life-cycle

Every software has a life cycle, just as a living organism or a new product. Commercial programs such as payroll, accounts, stock control and other software share a common life cycle pattern. One method of doing things may work well for a period of time. This may last for several years. However, owing to expansion or changes in the nature of the business, the economic environment, the need to keep up with new technology or other factors, the program may seem to be inadequate for future use. At this point investigations are made, requirements are analyzed, and a new specifications are proposed and a new program is developed. The life cycle of the new program thus starts again.

Stages in Software Life cycle

Analysis of the Problem and laying down Specifications The problem must be first defined and analyzed and the specifications of input and the output should be prepared. Sometimes a systems analyst is asked to prepare a feasibility study and the cost benefits. Once these are accepted, the software design starts.

Design and Development The design stage involves a number of tasks such as designing the output, input, files, database if applicable, system controls and test plan. Input forms must be designed, clerical procedures laid down and all aspects of the design must be documented. In the development stage, there are two aspects. First the program development and next the equipment acquisition. The senior programmer will rewrite program specifications to describe what each program in the system will do and how it will do it. The equipment will be acquired and the program will be tested on the computer system.

Testing and Debugging The program so developed will be tested for conforming the specification laid down and the results achieved. At the same time, if there are any bugs in the system, these will be removed and programs are re-tested.

Implementation This is the stage when the new software becomes operational. It is a critical phase of the project, requiring careful timing, coordination and training of all the user departments.

Maintenance All software need to be maintained. It means, performance monitored, modifications made if required, errors corrected, documents kept up-to-date. If the system needs

major modification then the life-cycle starts again. It means, the changes are done and the software development is once again carried out.

Training the User

Since one purpose of the new software system is to change existing procedures, *training* is crucial. All individuals have to understand what is required by the new software system. After the problems of installation have been resolved and the organization has adjusted to the changes created by the new software system, the operational stage begins. That is, the system now operates on a routine basis. However, this routine does not mean that it remains unchanged. There is a constant need for maintenance and enhancements. Maintenance is required because programs inevitably have errors that must be corrected when they appear. As users work with the new software system, they will learn more about it and will develop ideas for change and enhancements. The system continues to evolve throughout its life cycle.

1.11.6 Documentation and its Importance

Documentation provide a general outline as well a few specific details of the overall program structure. The documentation of a program is a continuous process. After successfully completing the program, we must ensure that documentation is complete and is in a finished, usable form. This includes both *technical documentation* for the programmers who may be working with and modifying the completed program and *user-level documentation* for the users of the program.

A good user documentation is essential if a program is to be a useful tool. Good technical documentation is essential for maintaining a program. We may decide at a future date to add features to the program, or we may have to find and correct errors in the program, both of which can be virtually impossible if the technical documentation is inadequate.

1.11.7 Aim of Program or System Documentation

The aims of documentation are:

(a) To help in the design of the system by working to a set of standards and having available a clear description of the work done so far. The documentation needs to be kept up-to-date throughout the project.
(b) Good documentation ensures that everyone involved in the system (system designers, programmers, and users) fully understand how their aspect of the system will work. For example, what data will be inputted and how, and what information will be available from the system. This allows any misunderstandings or disagreements to surface before they become deeply entrenched in the system.
(c) To ensure that the system can be maintained after completion even though the programmers involved in the original design or programming of the system may not be available. It is essential that proper documentation is kept to enable a newcomer to make necessary corrections, alterations or enhancements.

1.11.8 Contents of a Document of a System

The following are the parts of a document of the system:

(a) An accurate and up-to-date system specification.
(b) System flowchart(s) or data flow diagrams showing the inputs to the system, files required, processes to be carried out, and output from the system.
(c) A description of the purpose of each program within the system.
(d) Organization, contents and layout of each file used.
(e) Layout and contents of all output prints and displays.
(f) Current version of each program listing.
(g) Test data and expected results.

In addition, the following items of documentation will need to be prepared.

(a) Clerical Procedure Manual
(b) Operating Instruction
(c) Data Preparation Instruction

Clerical Procedure Manual

This manual details the activities that the clerical staff will undertake in preparing data for input to the system. For exam-

ple, *batching* documents and calculating *hash* totals. It will also describe what action is to be taken when errors occur–for example, when a validation program reports errors.

Operating Instructions

This document gives the details to the computer operator of how to run the program. It may include:

(a) Details of the procedure for starting the program.
(b) Details of disks and tapes required.
(c) Special stationery to be used.
(d) The number of copies of each report, and who is to receive the output.
(e) Backup procedures to be followed.
(f) Recovery procedures in the event of hardware failure.

Data Preparation Instructions

This will contain instructions on data entry, showing if necessary how each field should be entered. For example, a date field may be entered in various formats and the correct one needs to be specified.

In a small system, written perhaps for a single user working on a micro-computer, these three manuals may be combined into one User Manual.

1.12 Data Flow Diagrams (DFD)

Data flow diagrams are used to emphasize the logical flow of data through a system. The basic symbol is a circle or bubble and is called a 'transform' since it identifies a function that transforms data. Figure 1.51 illustrates a data flow diagram for a small organization.

Data flow diagrams can be made at different levels. Symbols are also used to denote logical conditions such as AND or an exclusive OR. Figure 1.52 shows the logical AND and the exclusive OR symbols used in a data flow diagram concerned with validating input. Both the transaction input and the master file input are required for the transform VALIDATE INPUT. One of the inputs by itself is insufficient. As a result of VALIDATE INPUT, either a rejected transaction or a valid transaction will be produced. However, both cannot be produced since the output

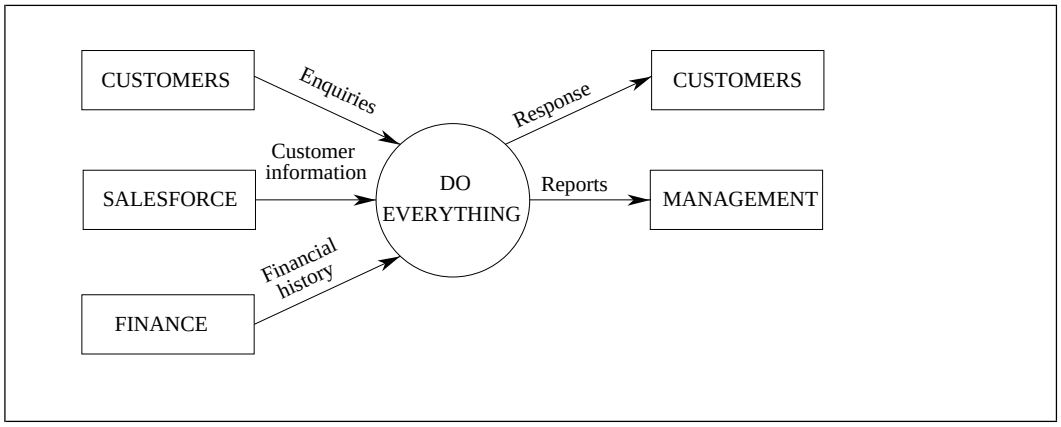

Figure 1.51 Data flow diagram for a small business organization

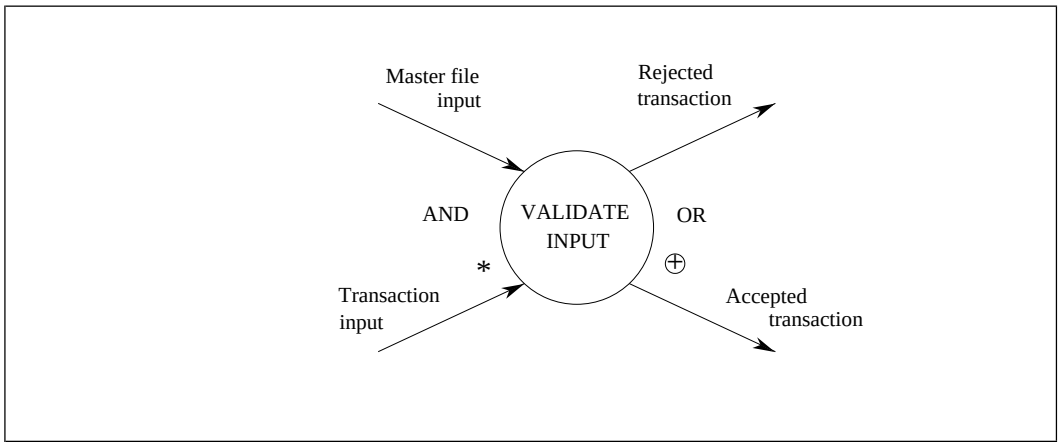

Figure 1.52 Data flow diagram for validating input data

form the transform is an exclusive OR.

The data flow diagrams shows the inputs and outputs clearly. A data flow diagram for customer enquiry system is shown in Figure 1.53. This data flow diagram has the basic elements namely SOURCE, DATA, STORE and DESTINATION.

1.13 Structure Charts

A flowchart is a blueprint or a logical diagram of the solution of a problem. As explained earlier, flowcharts use a standard set of

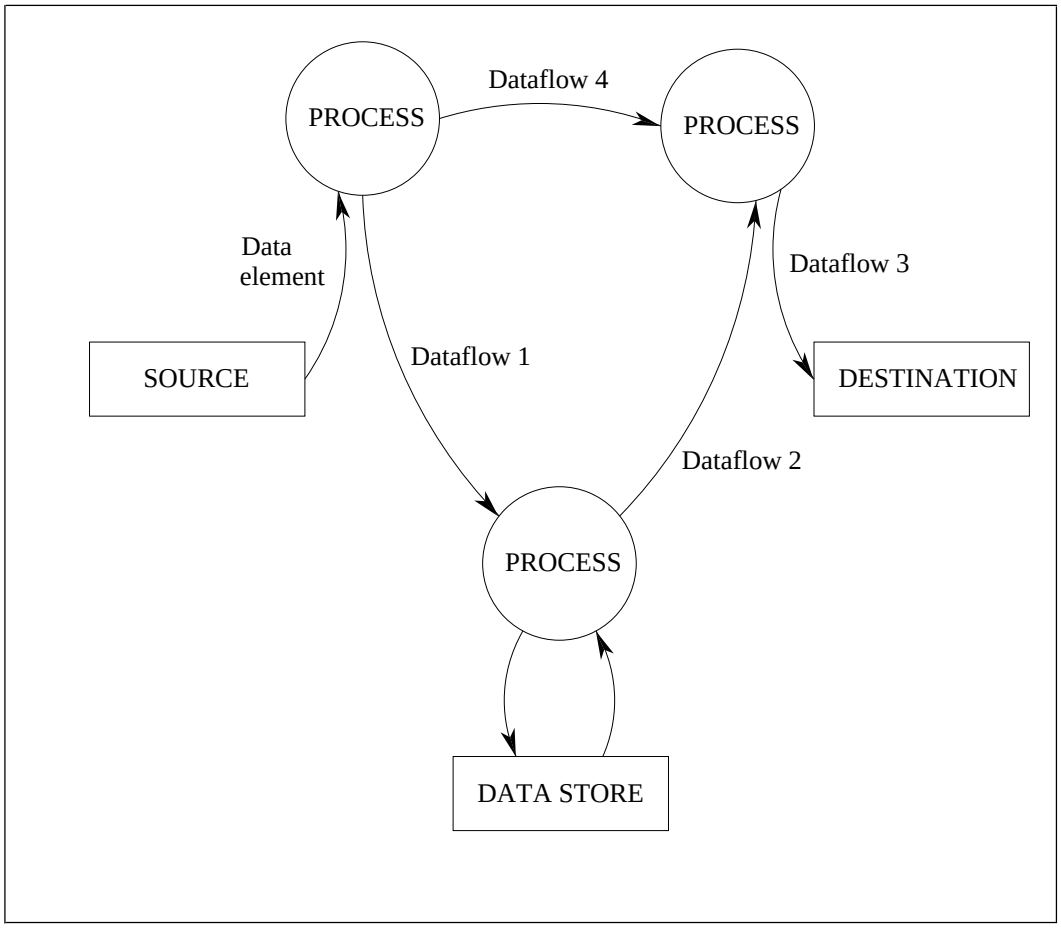

Figure 1.53 Data Flow Diagram for customer enquiry system

symbols and the actual operation to be performed is written inside the symbol. The arrow coming out of the symbol indicates which operations is to be performed next.

There are special types of flowcharts, called structure charts, that restrict the types of allowable operations and inter connections in order to produce well organized and readable diagrams. Structure charts are thus pictorial representation of the *design* of a system.

The flowcharting technique is useful primarily for macro-level or system flowcharts level where we are concerned with the most general level of operations needed to solve a large problem. Each element of a system flowchart would typically represent a fairly large and complex manual, clerical or computer procedure

for which an algorithm must be developed and implemented. These individual procedures could be developed and represented using an algorithmic language. However in a structured chart, there is a sequence of process shown along with the hierarchy of the process. These structure charts are used at the system design phase. Table 1.2 highlights the differences between data flow diagram and structure chart.

Table 1.2 Difference between data flow diagram and structure chart

Sl. No.	Data Flow Diagram	Structure Chart
(1)	In a data flow diagram only the process is shown but the sequence is not important	In a structure chart the sequence in which process is to be done is also important
(2)	In this case, the hierarchy or the ladder in which the process is to be executed is not important	In this case, hierarchy of the process is important
(3)	A data flow diagram is normally made at the system analysis phase.	A structure chart is normally made at the system design phase.

1.13.1 Elements of Structure Chart

Structure charts have three basic elements. These are :

Module

It is a rectangular shape with the process name written in it. Figure 1.54 shows a module. The box represents a module and the alphabets such as A, B, C written inside the box are the names of the processes performed.

Connectors

Connectors connect any two modules with the arrows drawn in the downward direction. Figure 1.55 shows the arrow which is a connector.

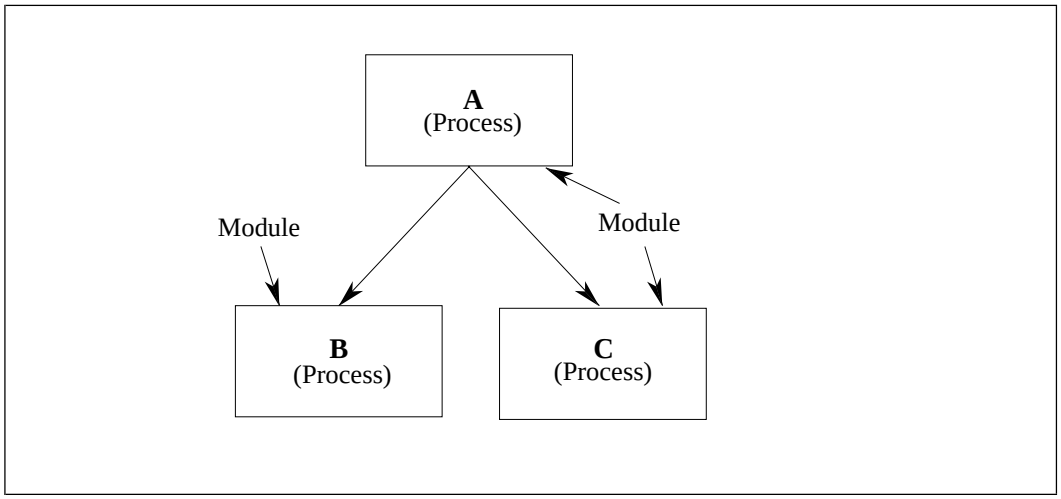

Figure 1.54 Module

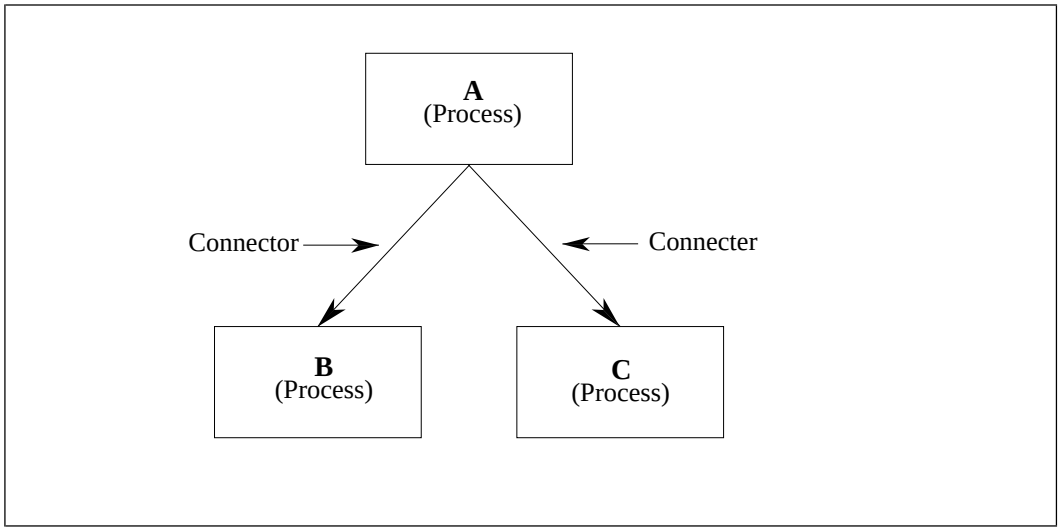

Figure 1.55 Connector

Couple

The carriers of data are called couples. These indicate the data flow between the modules. Figure 1.56 shows a couple. Couple can be of two different types.

(a) Data couple
(b) Control couple

Data Couple and Control Couple

Data couple carries data between two modules and it can have upward or downward direction. But control couple only carries information about data couple and does not carry any data. Control couple is always directed upwards because it shows the module which is to be evoked. Figure 1.56 show the two different types of couple. The arrows with a blank circle at the tail is called data couple. The arrow with a completely filled circle at the tail end is called a control couple.

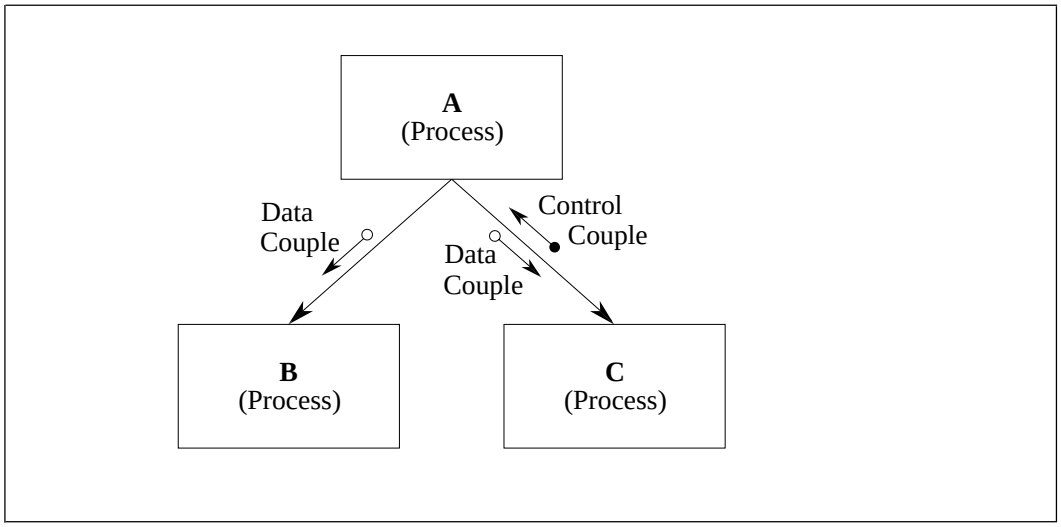

Figure 1.56 Couple (data and control)

How to Draw Structure Chart ?

In order to draw a structure chart, you will need to identify the inputs, process and the outputs. You should draw different modules for different processes in a system. This will be more clear with the help of the example given below.

Example 1.20

Draw the structure chart for calculating the amount of scholarship (a cut in the tuition fee) to be awarded to a student. The conditions are as follows:

(a) If marks are ≥ 90, give 15% cut in the fee

(b) If marks are ≥ 85, and < 90, then give 12% cut in the fee

(c) For all other cases, no cut in the fee.

Solution

The structure chart is shown in Figure 1.57.

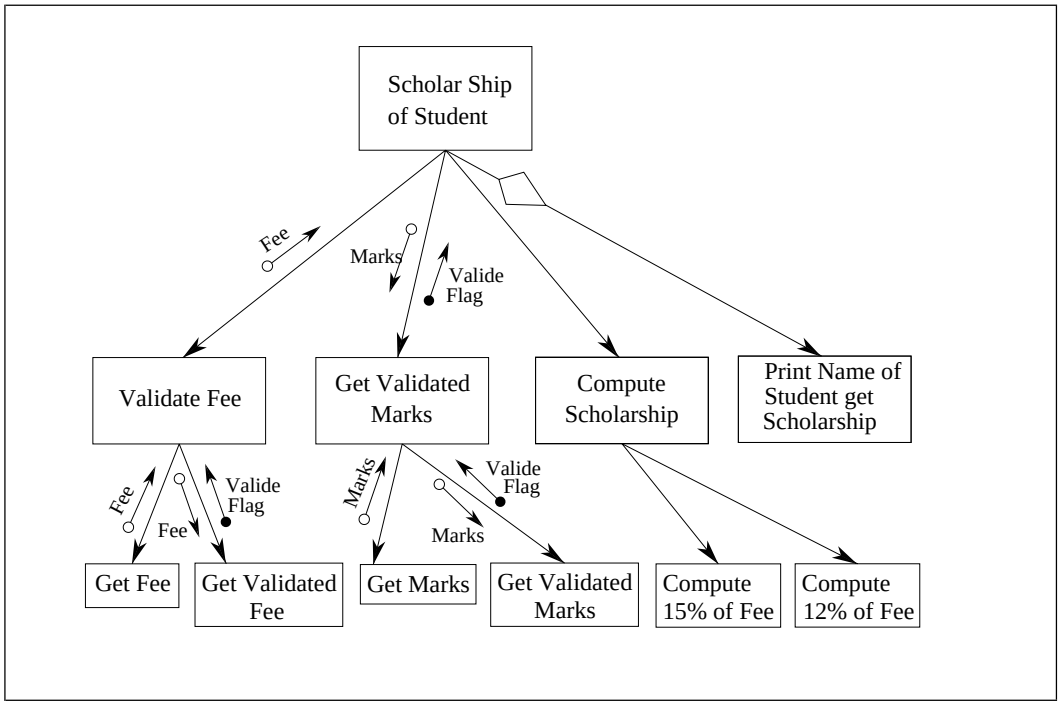

Figure 1.57 Structure chart for example 1.20

Example 1.21

Draw a structure chart for searching a record as roll no. and then print the record.

Solution

The structure chart for example 1.21 is shown in Figure 1.58. The diagram is self-explanatory.

Looping in a Structure Chart

Looping means the repetitions of a set of statements in a

program. In a structure chart, looping is shown by an angle with the arrow mark. See Figure 1.59.

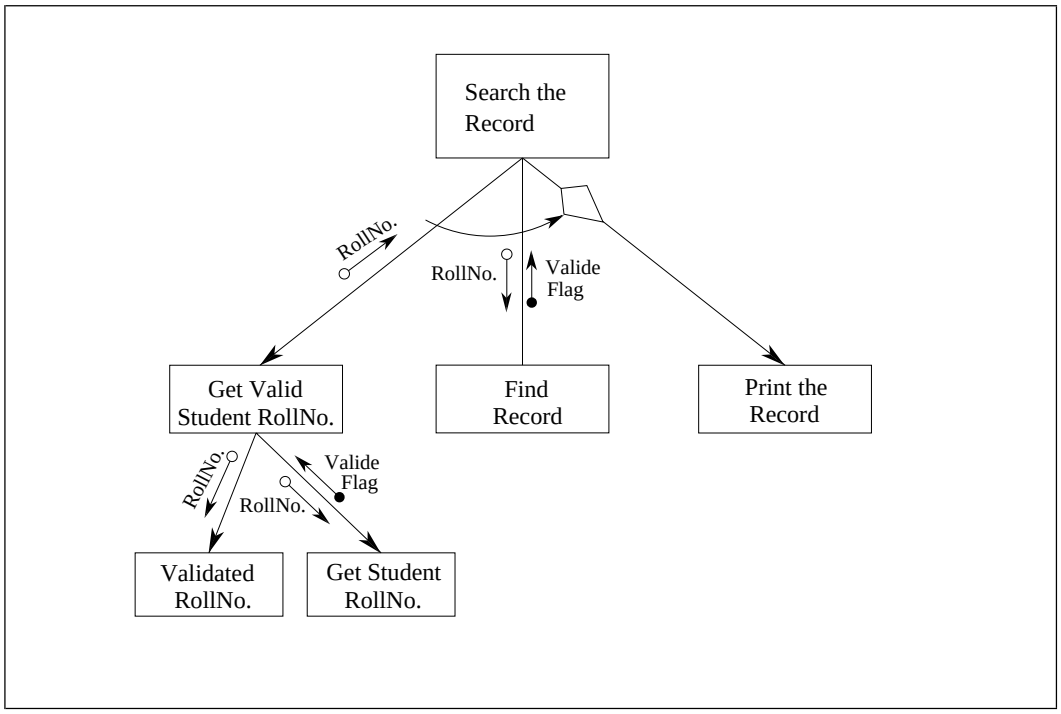

Figure 1.58 Structure chart for example 1.21

Conditional in Structure Chart

Conditional is a term which describes one action or operation which takes place only when the condition is true and the other action or operation which takes place when the condition is false. Figure 1.59 gives the representation of the condition by a rectangular polygon.

We have used the concepts of looping and conditional in examples 1.20 and 1.21 as shown in Figure 1.57 and 1.58. (See the angular arrows and the rectangular polygon).

1.14 Context Analysis Diagram

This type of diagram is made in the system analysis phase and it gives a graphic representation of the system as a whole. In this

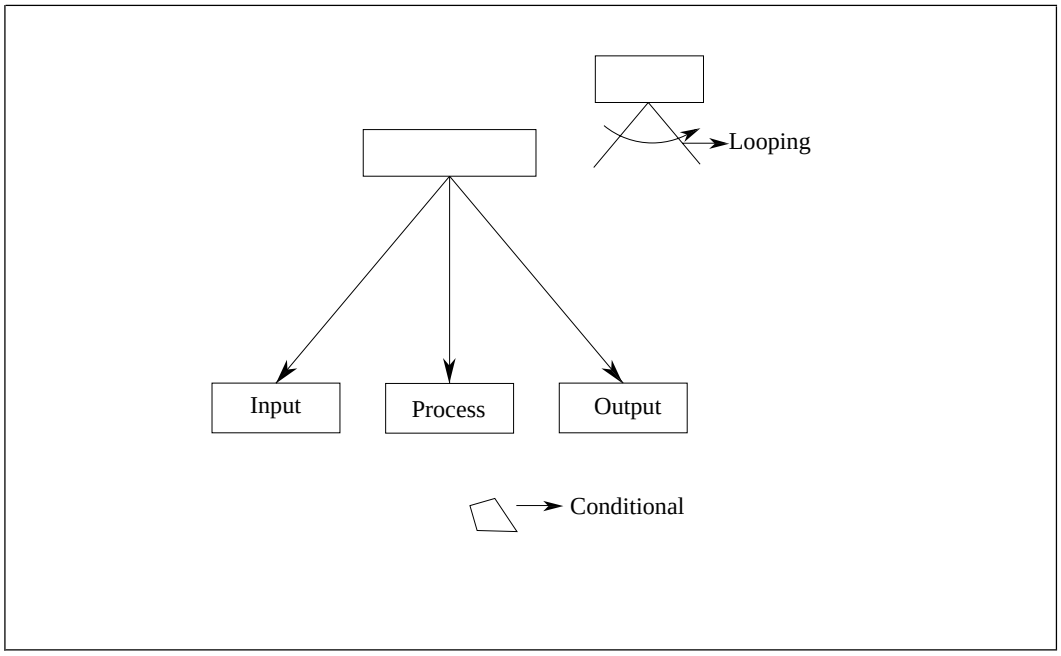

Figure 1.59 Looping and conditional in structure chart

diagram entire system is shown as a *single* process. This diagram identifies the external entities, the inputs and the outputs.

1.15 System Manual

The system manual is a statement of requirements which defines specifically what is to be accomplished by the proposed computer system. It is a fairly detailed document. Most of the data needed for it is collected in the feasibility study. The document serves both as a summary of the proposed system for internal purposes and as a statement used in inviting equipment proposals from vendors of data processing equipment.

On the basis of a preliminary screening, four or five vendors may be invited to submit proposals. Each vendor is provided with a copy of the manual of specifications and the rules for submitting proposals. These may be followed-up by interviews with vendors to clarify any misunderstandings or uncertainties in the specifications. When the proposals are ready, the manufacturer's representative is usually provided with an opportunity for a

presentation to the study group. This group studies the manual and decides the selection of equipment and the supplier.

1.16 Source Code

The source code is the language in which a program is written by the programmer. Source code is translated into object code by assemblers and compilers, or a line at a time by an interpreter. In some cases, the source code may be automatically converted into another dialect or language by a conversion program. The source code is not executable by the computer directly. It must be converted into machine language first.

1.16.1 Command File

The command file is a machine language program that can be loaded and executed in the computer. For example in Microsoft DOS operating system, the *Command.Com* is one such file which is used to boot the computer. Command files have a suffix .COM. These files are designed to work only in specific memory locations. They contrast with .EXE files, which can be relocated anywhere in memory.

1.17 Input/Output Report Formats

The reports that are generated for the management or for the working of an organization should be different from the data format which is entered at the input stage of the processing cycle. Thus the system analysis shows the relationship between the individual items on each output and the items available as inputs to the system.

The output reports and input documents should be documented in terms of data contents and approximate layout. It is possible to work back from the output contents, through the system, to the inputs required. This is done by determining which output data items are derived by calculation or by logical deduction. All other items can then be broken down into those which require fresh input every time, as part of input transaction documents, and those which can be stored on file because they are historical or relatively static.

1.18 Data-item Dictionaries

Data-item dictionary is used to define data, including identifiers, location, and format for storage characteristics. It holds the name, type range of values, source, and authorization for access of each data element in the organization's files and databases. It also indicates which application program what data, so that whenever a change in a data structure is contemplated, a list of the affected programs can be generated. The data dictionary may be a stand-alone information system used for management and documentation purposes, or it may be an integral part of data-base management system where it is used to actually control its operation. Data integrity and accuracy is better insured in the latter case. The data-item dictionary will contain the following type of information:

(a) What tables and columns are included in the present structure.

(b) The names of the current tables and columns.

(c) The characteristics of each item of data, such as its length and data type.

(d) Any restrictions on the value of certain columns.

(e) The meaning of any data fields that are not evident.

(f) The relationships between items of data.

1.19 Testing Results

In order to completely test the program logic, the test data must test each logical function of the program. The test data selected for testing a program should include:

(a) Normal data which will test the generally used program paths.

(b) Unusual but valid data, which will test the program paths used to handle exceptions. Such data might be encountered occasionally in running the program.

(c) Inappropriate data which will test the error handling capabilities of the program.

If a program runs successfully with the test data and produces correct results, it is normally released for use. However, even at this stage errors may remain. There are certain errors in complex systems that remain hidden for months and years together.

1.20 Review Reports and Management Decisions/Orders

Once the computer run system has become operational, it will need to be examined to see if it has met its objectives. For example, the costs and benefits will be compared with the estimates produced at the system inception. This particular activity is often known as "*Post Audit*"

The new system will also need to be reviewed and maintained periodically for the following reasons:

(a) To deal with unforeseen problems arising in operation. For example, programs may need to be modified to deal with unforeseen circumstances.
(b) To confirm that the planned objectives are being met and to take action if they are not.
(c) To ensure that the system is able to cope with the changing requirements of business.

The results of a systems review would be used in future systems analysis assignments.

1.20.1 Management Decision/Orders

In the event of adverse remarks given by the post audit team in the functioning of the new computer system, it may be necessary for the management to take some bold decision or issue orders for the change/review of the system. For example, if the system is not able to cope with the load of the work for which it was originally designed, then it may not be worthwhile to continue with the *automatic* new system. Let us take the case of the system designed for the school fee collection. The system may be required to meet the immediate collection of the fee. But if the system does not meet this need as the working of the software is not reliable or there is some other inherent defect in the system created, then it will be necessary for the management to look into the new system and ask for the review or change of the earlier system.

Test Paper Based on Chapter 1

Time allowed : 3 hr
Max.Marks : 100

Answer all questions

Q 1. Answer the following

(a) What are the advantages and limitation of pseudocodes ?

(b) List the program preparation techniques that are often included under the term 'Structured Programming'.

Q 2. What is a Flowchart ? List the flowcharting rules.

Q 3. What are the advantages and limitations of flowcharts ?

Q 4. Differentiate between the following

(a) Pseudocode and Flowchart

(b) Compiler and Interpreter

(c) Top-down and Bottom-up design techniques

(d) Testing and Debugging

Q 5. What do you understand by structured programming ? State the characteristics of structured programs.

Q 6. Write short notes on the following

(a) Data flow diagram (DFD)

(b) Compiler

(c) Trailer Record

(d) Algorithm

Q 7. What is modular concept in programming ? Mention a few essential requirements of modular programming.

Q 8. What are the two broad types of programming errors ? How are they detected?

Q 9. What do you understand by documenting a program ? Why is it necessary ?

Q 10. (a) What is meant by program maintenance ? How can proper program design make it easier to maintain programs ?

(b) What are the different ways of debugging and testing a program ?