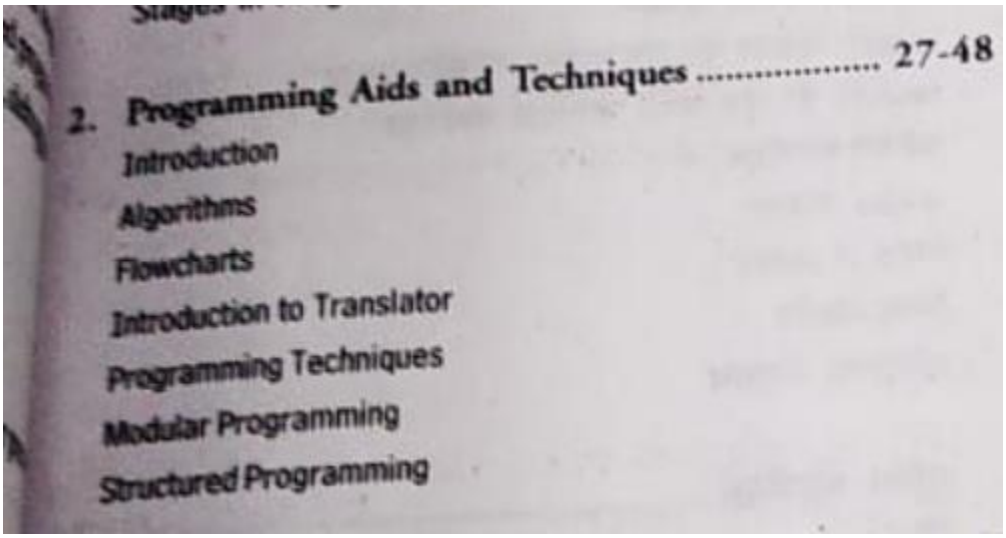
BSC FY NOTES

UNIT – 2



FLOWCHARTS

Flowcharts

A flowchart provides appropriate steps to be followed in order to arrive at the solution to a problem. It is a program design tool which is used before writing the actual program. Flowcharts are generally developed in the early stages of formulating computer solutions.

A flowchart comprises a set of various standard shaped boxes that are interconnected by flow lines. Flow lines have arrows to indicate the direction of the flow of control between the boxes. The activity to be performed is written within the boxes in English. In addition, there are connector symbols that are used to indicate that the flow of control continues elsewhere, for example, the next page.

Flowcharts facilitate communication between programmers and b usiness persons. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high-level language. Often flowcharts are helpful in explaining the program to others. Hence, a flowchart is a must for better documentation of a complex program.

Standards for flowcharts The following standards should be adhered to while drawing flow charts.

- Flowcharts must be drawn on white, unlined 81/2 × 11 paper, on one side only.″ ′
- Flowcharts start on the top of the page and flow down and to the right.
- Only standard flowcharting symbols should be used.
- A template to draw the final version of flowchart should be used.
- The contents of each symbol should be printed legibly.
- English should be used in flowcharts, not programming language.

- The flowchart for each subroutine, if any, must appear on a separate page. Each subroutine begins with a terminal symbol with the subroutine name and a terminal symbol labeled return at the end.
- Draw arrows between symbols with a straight edge and use arrowheads to indicate the direction of the logic flow.

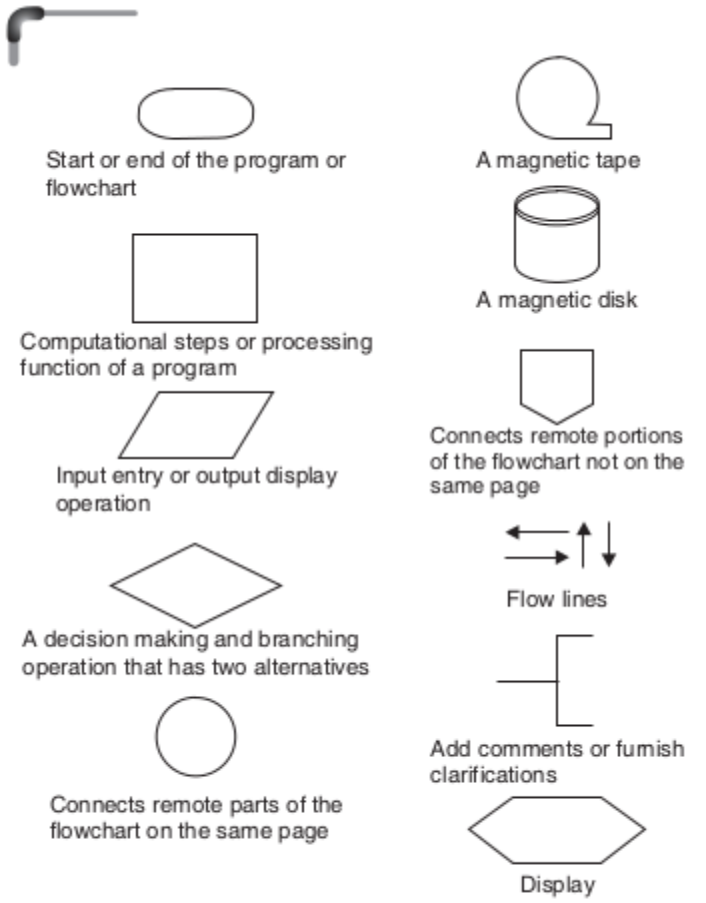


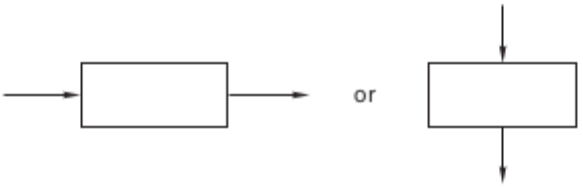
Figure 1.12 Flowchart symbols

Guidelines for drawing a flowchart Flowcharts are usually drawn using standard symbols; however, some special symbols can also be developed when required. Some standard symbols frequently required for flowcharting many computer programs are shown in Fig.1.12.

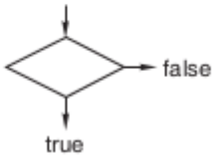
The following are some guidelines in flowcharting.

- In drawing a proper flowchart, all necessary requirements should be listed out in a logical order.
- There should be a logical **start** and **stop** to the flowchart.

- The flowchart should be clear, neat, and easy to follow. There should be no ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should emerge from a process symbol.



- Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, can leave the decision symbol.

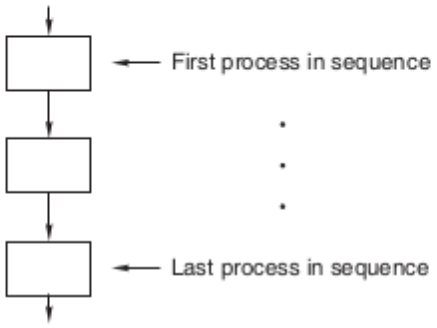


- Only one flow line is used in conjunction with a terminal symbol.

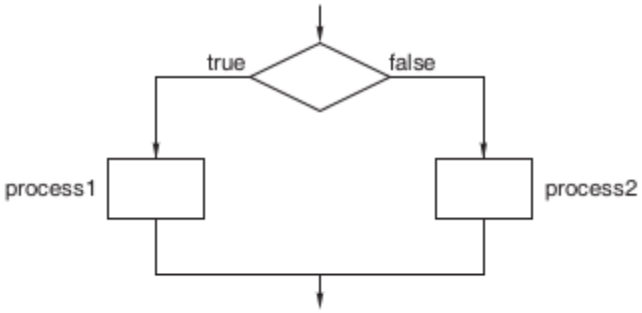
- The writing within standard symbols should be brief. If necessary, the annotation symbol can be used to describe data or computational steps more clearly.

- If the flowchart becomes complex, connector symbols should be used to reduce the number of flow lines. The intersection of flow lines should be avoided to make the flowchart a more effective and better way of communication.

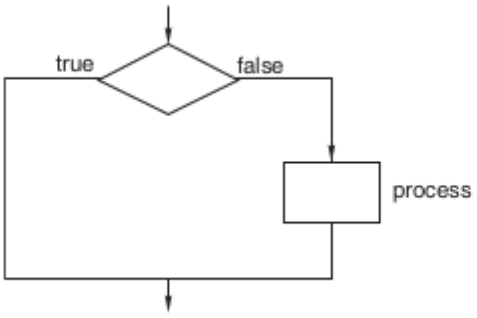
- The validity of the flowchart should be tested by passing simple test data through it.
- A *sequence* of steps or processes that are executed in a particular order is shown using process symbols connected with flow lines. One flow line enters the first process while one flow line emerges from the last process in the sequence.



- *Selection* of a process or step is depicted by the decision making and process symbols. Only one input indicated by one incoming flow line and one output flowing out of this structure exists. The decision symbol and the process symbols are connected by flow lines.



- *Iteration* or *looping* is depicted by a combination of process and decision symbols placed in proper order. Here flow lines are used to connect the symbols and depict input and output to this structure.



Advantages of using flowcharts

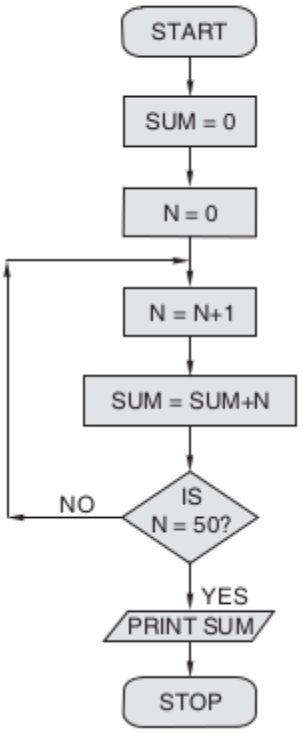
- *Communication*: Flowcharts are a better way of communicating the logic of a system to all concerned.
- *Effective analysis*: With the help of flowcharts, problems can be analyzed more effectively.
- *Proper documentation*: Program flowcharts serve as a good program documentation needed for various purposes.
- *Efficient coding*: Flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- *Proper debugging*: Flowcharts help in the debugging process.
- *Efficient program maintenance*: The maintenance of an operating program becomes easy with the help of a flowchart.

Limitations of using flowcharts

- *Complex logic*: Sometimes, the program logic is quite complicated. In such a case, a flowchart becomes complex and clumsy.
- *Alterations and modifications*: If alterations are required, the flowchart may need to be redrawn completely.
- *Reproduction*: Since the flowchart symbols cannot be typed in, the reproduction of a flowchart becomes a problem.
- *Loss of objective*: The essentials of what has to be done can easily be lost in the technical details of how it is to be done.

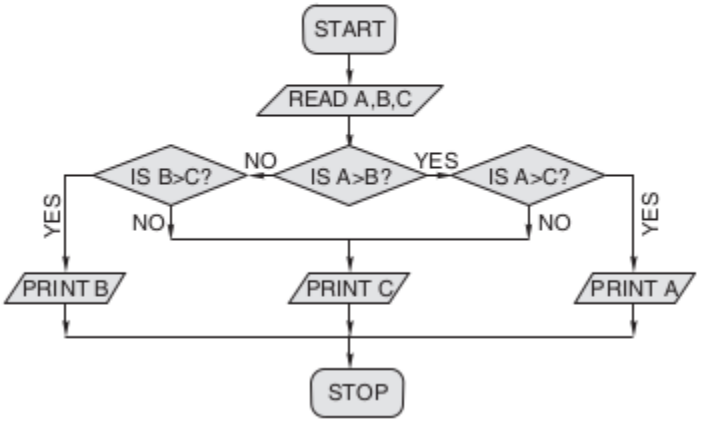
Draw a flowchart to find the sum of the first 50 natural numbers.

Solution



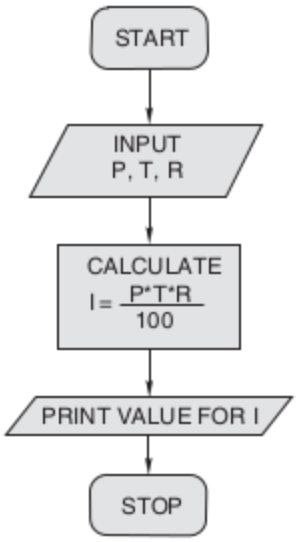
25. Draw a flowchart to find the largest of three numbers *A*, *B*, and *C*.

Solution



27. Draw a flowchart for calculating the simple interest using the formula $SI = (P * T * R)/100$, where P denotes the principal amount, T time, and R rate of interest. Also, show the algorithm in step-form.

Solution



Step 1: START

Step 2: Read P, T, R

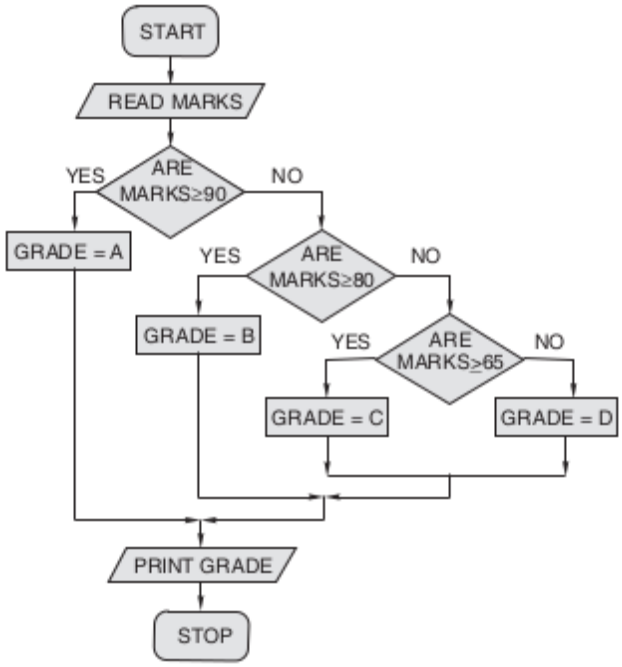
Step 3: Calculate I=P*R*T/100

Step 4: PRINT I

Step 5: STOP

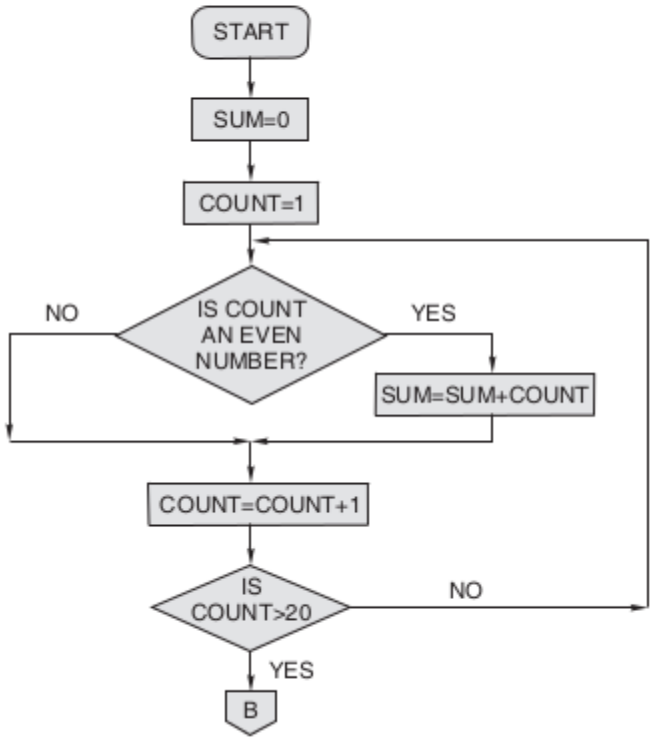
29. Prepare a flowchart to read the marks of a student and classify them into different grades. If the marks secured are greater than or equal to 90, the student is awarded Grade *A*; if they are greater than or equal to 80 but less than 90, Grade *B* is awarded; if they are greater than or equal to 65 but less than 80, Grade *C* is awarded; otherwise Grade *D* is awarded.

Solution



31. Draw a flowchart for printing the sum of even terms contained within the numbers 0 to 20.

Solution



ALGORITHM

1.9.1 What is an Algorithm?

Computer scientist Niklaus Wirth stated that

Program = Algorithms + Data

An algorithm is a part of the plan for the computer program. In fact , an algorithm is 'an effective procedure for solving a problem in a finite number of steps'.

It is effective, which means that an answer is found and it has a finite number of steps. A well-designed algorithm will always provide an answer; it may not be the desired answer but there will be an answer. It may be that the answer is that there is no answer. A well- designed algorithm is also guaranteed to terminate.

1.9.2 Different Ways of Stating Algorithms

Algorithms may be represented in various ways. There are four ways of stating algorithms.
These are as follows:

- Step-form
- Pseudo-code
- Flowchart

In the step form representation, the procedure of solving a problem is stated with written statements. Each statement solves a part of the problem and these together complete the solution. The step-form uses just normal language to define each procedure. Every statement, that defines an action, is logically related to the preceding statement. This algorithm has been discussed in the following section with the help of an example.

The pseudo-code is a written form representation of the algorithm. However it differs from the step form as it uses a restricted vocabulary to define its action of solving the problem. One problem with human language is that it can seem to be imprecise. But the pseudo-code, which is in human language, tends toward more precision by using a limited vocabulary.

Flowchart and Nassi-Schneiderman are graphically oriented representation forms. They use symbols and language to represent sequence, decision, and repetition

It can be seen that the algorithm has a number of steps and that some steps (steps 1, 3, and 5) involve decision making and one step (step 5 in this case) involves repetition, in this case the process of waiting for the kettle to boil.

From this example, it is evident that algorithms show these three features:

- Sequence (also known as process)
- Decision (also known as selection)
- Repetition (also known as iteration or looping)

Therefore, an algorithm can be stated using three basic constructs: sequence, decision, and repetition.

Sequence

Sequence means that each step or process in the algorithm is executed in the specified order. In the above example, each process must be in the proper place otherwise the algorithm will fail.

The decision constructs—if … then, if … then … else …

In algorithms the outcome of a decision is either true or false; there is no state in between.
The outcome of the decision is based on some condition that can only result in a true or false value. For example,

```
if today is Friday then collect pay
```

is a decision and the decision takes the general form:

```
if proposition then process
```

A proposition, in this sense, is a statement, which can only be true or false. It is either true that 'today is Friday' or it is false that 'today is not Friday'. It can not be both true and false. If the proposition is true, then the process or procedure that follows the then is executed. The decision can also be stated as:

```
if proposition
    then process1
else process2
```

This is the `if … then … else …` form of the decision. This means that if the proposition is true then execute process1, else, or otherwise, execute process2.

The first form of the decision if proposition then process has a null else, that is, there is no else.

The repetition constructs—repeat and while

Repetition can be implemented using constructs like the repeat loop, while loop, and `if.. then .. goto ..` loop.

The Repeat loop is used to iterate or repeat a process or sequence of processes until some condition becomes true. It has the general form:

```
Repeat
    Process1
    Process2
    ..............
    ..............
    ProcessN
    Until proposition
```

Here is an example.

```
Repeat
    Fill water in kettle
Until kettle is full
```

The process is 'Fill water in kettle,' the proposition is 'kettle is full'.

The Repeat loop does some processing before testing the state of the proposition.

What happens though if in the above example the kettle is already full? If the kettle is already full at the start of the Repeat loop, then filling more water will lead to an overflow.

This is a drawback of the Repeat construct.

In such a case the `while` loop is more appropriate. The above example with the `while` loop is shown as follows:

```
while kettle is not full
    fill water in kettle
```

Since the decision about the kettle being full or not is made before filling water, the possibility of an overflow is eliminated. The while loop finds out whether some condition is true before repeating a process or a sequence of processes.

If the condition is false, the process or the sequence of processes is not executed. The general form of `while` loop is:

```
while proposition
begin
Process 1
```

Process 2

……..

……...

Process N

end

1.9.4 What are Variables?

So long, the elements of algorithm have been discussed. But a program comprises of algorithm and data. Therefore, it is now necessary to understand the concept of data. It is known that data is a symbolic representation of value and that programs set the context that gives data a proper meaning. In programs, data is transformed into information. The question is, how is data represented in programs?

Almost every algorithm contains data and usually the data is 'contained' in what is called a variable. The variable is a container for a value that may vary during the execution of the program. For example, in the tea-making algorithm, the level of water in the kettle is a variable, the temperature of the water is a variable, and the quantity of tea leaves is also a variable.

Each variable in a program is given a name, for example,

- Water_Level
- Water_Temperature
- Tea_Leaves_Quantity

and at any given time the value, which is represented by Water_Level, for instance, may be different to its value at some other time. The statement

`if` the kettle does not contain water `then` fill the kettle could also be written as

`if` Water_Level is 0 `then` fill the kettle

or

`if` Water_Level = 0 `then` fill the kettle

At some point Water_Level will be the maximum value, whatever that is, and the kettle will be full.

Variables and data types

The data used in algorithms can be of different types. The simplest types of data that an algorithm might use are

- numeric data, e.g., 12, 11.45, 901, etc.
- alphabetic or character data such as 'A', 'Z', or 'This is alphabetic'
- logical data, that is, propositions with true/false values

Naming of variables

One should always try to choose meaningful names for variables in algorithms to improve the readability of the algorithm or program. This is particularly important in large and complex programs.

In the tea-making algorithm, plain English was used. It has been shown how variable names may be used for some of the algorithm variables. In Table 1.3, the right-hand column contains variable names which are shorter than the original and do not hide the meaning of the original phrase. Underscores have been given to indicate that the words belong together and represent a variable.

Table 1.3 Algorithm using variable names

Algorithm in Plain English	Algorithm using Variable Names
1. If the kettle does not contain water, then fill the kettle.	**1.** If kettle_empty then fill the kettle.
2. Plug the kettle into the power point and switch it on.	**2.** Plug the kettle into the power point and switch it on.
3. If the teapot is not empty, then empty the teapot.	**3.** If teapot_not_empty then empty the teapot.
4. Place tea leaves in the teapot.	**4.** Place tea leaves in the teapot.
5. If the water in the kettle is not boiling then go to step 5.	**5.** If water_not_boiling then go to step 5.
6. Switch off the kettle.	**6.** Switch off the kettle.
7. Pour water from the kettle into the teapot.	**7.** Pour water from the kettle into the teapot.

There are no hard and fast rules about how variables should be named but there are many conventions. It is a good idea to adopt a conventional way of naming variables.

The algorithms and programs can benefit from using naming conventions for processes too.

Points to Note

1. Data is a symbolic representation of value.
2. A variable, which has a name, is a container for a value that may vary during the execution of the program.

1.9.5 Subroutines

A simple program is a combination of statements that are implemented in a sequential order. A statement block is a group of statements. Such a program is shown in Fig. 1.11(i). There might be a specific block of statement, which is also known as a procedure, that is run several times at different points in the implementation sequence of the larger program. This is shown in Fig.1.11(ii). Here, this specific block of statement is named "procedure X". In this example program, the "procedure X" is written twice in this example. This enhances the size of the program. Since this particular procedure is required to be run at two specific points in the implementation sequence of the larger program, it may be treated as a separate entity and not included in the main program. In fact, this procedure may be called whenever required as shown in Fig.1.11(iii). Such a procedure is known as a subroutine.

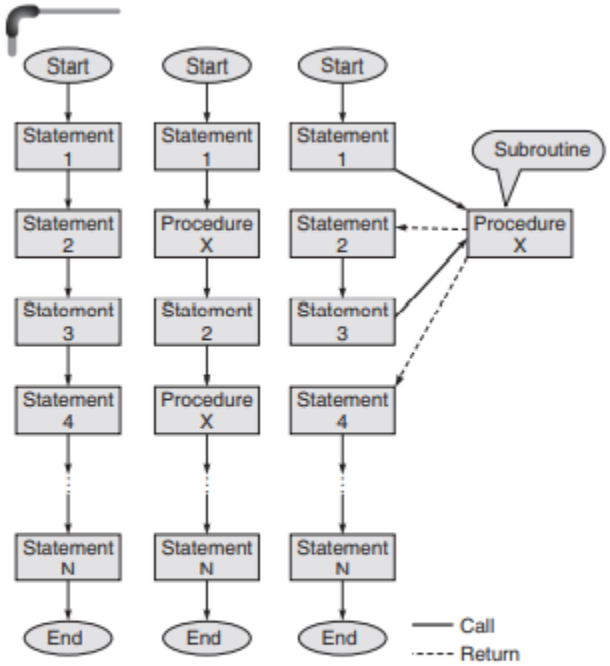


Figure 1.11 Program structures

Therefore, a subroutine, also known as procedure, method or function, is a portion of instruction that is invoked from within a larger program to perform a specific task. At the same time the subroutine is relatively independent of the remaining statements of the larger program. The subroutine behaves in much the same way as a program that is used as one step in a larger program. A subroutine is often written so that it can be started ("called") several times and/or from several places during a single execution of the program, including from other subroutines, and then branch back (*return*) to the next instruction after the "call", once the subroutine's task is done. Thus, such subroutines are invoked with a CALL statement with or without passing of parameters from the calling program. The subroutine works on the parameters if given to it, otherwise it works out the results and gives out the result by itself and returns to the calling program or pass the results to the calling program before returning to it.

The technique of writing subroutine has some distinct advantages. The subroutine reduces duplication of block of statements within a program, enables reuse of the block of statements that forms the subroutine across multiple programs, decomposes a complex task into simpler steps, divides a large programming task among various programmers or various stages of a project and hides implementation details from users.

However, there are some disadvantages in using subroutines. The starting or invocation of a subroutine requires some computational overhead in the call mechanism itself. The subroutine requires some well defined housekeeping techniques at it's entry and exit from it.

Points to Note

1. A subroutine is a logical collection of instructions that is invoked from within a larger program to perform a specific task.
2. The subroutine is relatively independent of the remaining statements of the program that invokes it.
3. A subroutine can be invoked several times from several places during a single execution of the invoking program.
4. After completing the specific task, a subroutine returns to the point of invocation in the larger program.

1. Each algorithm will be logically enclosed by two statements START and STOP.
2. To accept data from user, the INPUT or READ statements are to be used.
3. To display any user message or the content in a variable, PRINT statement will be used. Note that the message will be enclosed within quotes.
4. There are several steps in an algorithm. Each step results in an action. The steps are to be acted upon sequentially in the order they are arranged or directed.

4. The arithmetic operators that will be used in the expressions are
   (i) '←' ….Assignment (the left-hand side of '←' should always be a single variable)

   Example: The expression x ← 6 means that a value 6 is assigned to the variable x. In terms of memory storage, it means a value of 6 is stored at a location in memory which is allocated to the variable x.

   (ii) '+'….. Addition

   Example: The expression z ← x + y means the value contained in variable x and the value contained in variable y is added and the resulting value obtained is assigned to the variable z.

   (iii) '–'….. Subtraction

   Example: The expression z ← x – y means the value contained in variable y is subtracted from the value contained in variable x and the resulting value obtained is assigned to the variable z

   (iv) '*'….. Multiplication

   Example: Consider the following expressions written in sequence:

   x ← 5
   y ← 6
   z ← x * y

   The result of the multiplication between x and y is 30. This value is therefore assigned to z.

   (v) '/'….. Division

   Example: The following expressions written in sequence illustrates the meaning of the division operator :

   x ← 10
   y ← 6
   z ← x/y

The quotient of the division between x and y is 1 and the remainder is 4. When such an operator is used the quotient is taken as the result whereas the remainder is rejected. So here the result obtained from the expression x/y is 1 and this is assigned to z.

5. In propositions, the commonly used relational operators will include

   (i) '>' ..... Greater than

   Example: The expression x > y means if the value contained in x is larger than that in y then the outcome of the expression is true, which will be taken as 1. Otherwise, if the outcome is false then it would be taken as 0.

   (ii) '<=' .....Less than or equal to

   Example: The expression x <= y implies that if the value held in x is either less than or equal to the value held in y then the outcome of the expression is true and so it will be taken as 1.

   But if the outcome of the relational expression is false then it is taken as 0.

   (iii) '<' ..... Less than

   Example: Here the expression x < y implies that if the value held in x is less than that held in y then the relational expression is true, which is taken as 1, otherwise the expression is false and hence will be taken as 0.

(iv) '=' ..... Equality

Example: The expression x = y means that if the value in x and that in y are same then this relational expression is true and hence the outcome is 1 other wise the outcome is false or 0.

(v) '>=' ..... Greater than or equal to

Example: The expression x >= y implies that if the value in x is larger or equal to that in y then the outcome of the expression is true or 1, otherwise it is false or 0.

(vi) '!=' ..... Non- equality

Example: The expression x != y means that if the value contained in x is not equal to the value contained in y then the outcome of the expression is true or 1, otherwise it is false or 0.

Note: The 'equal to (=)' operator is used both for assignment as well as equality specification. When used in proposition, it specifies equality otherwise assignment. To differentiate 'assignment' from

'equality' left arrow (←) may be used. For example, a ←b is an assignment but a = b is a proposition for checking the equality.

6. The most commonly used logical operators will be AND, OR and NOT. These operators are used to specify multiple test conditions forming composite proposition. These are

   (i) 'AND'..... Conjunction

   The outcome of an expression is true or 1 when both the propositions AND-ed are true otherwise it is false or 0.

   Example: Consider the expressions

   x ← 2

   y ← 1

   x = 2 AND y = 0

   In the above expression the proposition 'x = 2' is true because the value in x is 2. Similarly, the proposition 'y = 0' is untrue as y holds 1 and therefore this proposition is false or 0. Thus, the above expression may be represented as 'true' AND 'false' the outcome for which is false or 0.

   (ii) 'OR' ..... Disjunction

   The outcome of an expression is true or 1 when anyone of the propositions OR-ed is true otherwise it is false or 0.

   Example: Consider the expressions

   x ← 2

   y ← 1

   x = 2 OR y = 0

   Here, the proposition 'x = 2' is true since x holds 2 while the proposition 'y = 0' is untrue or false. Hence the third expression may be represented as 'true' OR 'false' the outcome for which is true or 1.

   (iii) 'NOT' ..... Negation

   If outcome of a proposition is 'true', it becomes 'false' when negated or NOT-ed.

   Example: Consider the expression

   x ← 2

   NOT x = 2

   The proposition 'x = 2' is 'true' as x contains the value 2. But the second expression negates this by the logical operator NOT which gives an outcome 'false'.

Write the algorithm for finding the sum of any two numbers.

Solution Let the two numbers be *A* and *B* and let their sum be equal to *C*. Then, the desired algorithm is given as follows:

```
1. START
2. PRINT "ENTER TWO NUMBERS"
3. INPUT A, B
4. C ← A + B
5. PRINT C
6. STOP
```

Add values assigned to A and B and assign this value to C

Explanation The first step is the starting point of the algorithm. The next step requests the programmer to enter the two numbers that have to be added. Step 3 takes in the two numbers given by the programmer and keeps them in variables A and B. The fourth step adds the two numbers and assigns the resulting value to the variable C. The fifth step prints the result stored in C on the output device. The sixth step terminates the procedure.

Write the algorithm for determining the remainder of a division operation where the dividend and divisor are both integers.

Solution Let *N* and *D* be the dividend and divisor, respectively. Assume *Q* to be the quotient, which is an integer, and *R* to be the remainder. The algorithm for the given problem is as follows.

```
1. START
2. PRINT "ENTER DIVIDEND"
3. INPUT N
4. PRINT "ENTER DIVISOR"
5. INPUT D
6. Q ← N/D (Integer division)
7. R ← N - Q * D
8. PRINT R
9. STOP
```

Only integer value is obtained and remainder ignored

Explanation The first step indicates the starting point of the algorithm. The next step asks the programmer to enter the dividend value. The third step keeps the dividend value in the variable N. Step 4 asks for the divisor value to be entered. This is kept in the variable D. In step 6, the value in N is divided by that in D. Since both the numbers are integers, the result is an integer. This value is assigned to Q. Any remainder in this step is ignored. In step 7, the remainder is computed by subtracting the product of the integer quotient and the integer divisor from integer dividend N. The computed value of the remainder is an integer here and obviously less than the divisor. The remainder

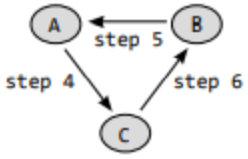
value is assigned to the variable R. This value is printed on an output device in step 8. Step 9 terminates the algorithm.

6. Construct the algorithm for interchanging the numeric values of two variables.

Solution Let the two variables be *A* and *B*. Consider *C* to be a third variable that is used to store the value of one of the variables during the process of interchanging the values.

The algorithm for the given problem is as follows.

```
1. START
2. PRINT "ENTER THE VALUE OF A & B"
3. INPUT A, B
4. C ← A
5. A ← B
6. B ← C
7. PRINT A, B
8. END
```



Explanation Like the previous examples, the first step indicates the starting point of the algorithm. The second step is an output message asking for the two values to be entered. Step 3 puts these values into the variables A and B. Now, the value in variable A is copied to variable C in step 4. In fact the value in A is saved in C. In step 5 the value in variable B is assigned to variable A. This means a copy of the value in B is put in A. Next, in step 6 the value in C, saved in it in the earlier step 4 is copied into B. In step 7 the values in A and B are printed on an output device. Step 8 terminates the procedure.

7. Write an algorithm that compares two numbers and prints either the message identifying the greater number or the message stating that both numbers are equal.

Solution This example demonstrates how the process of selection or decision making is implemented in an algorithm using the step-form. Here, two variables, A and B, are assumed to represent the two numbers that are being compared. The algorithm for this problem is given as follows.

```
1. START
2. PRINT "ENTER TWO NUMBERS"
3. INPUT A, B
4. IF A > B THEN
            PRINT "A IS GREATER THAN B"
5. IF B > A THEN
            PRINT "B IS GREATER THAN A"
6. IF A = B THEN
            PRINT "BOTH ARE EQUAL"
7. STOP
```

Explanation The first step indicates the starting point of the algorithm. The next step prints a message asking for the entry of the two numbers. In step 3 the numbers entered are kept in the variables A and B. In steps 4, 5 and 6, the values in A, B and C compared with the IF ...THEN construct. The relevant message is printed whenever the proposition between IF and THEN is found to agree otherwise the next step is acted upon. But in any case one of the message would be printed because at least one of the propositions would be true. Step 7 terminates the procedure.

8. Write an algorithm to check whether a number given by the user is odd or even.

Solution Let the number to be checked be represented by *N*. The number *N* is divided by 2 to give an integer quotient, denoted by *Q*. If the remainder, designated as *R*, is zero, *N* is even; otherwise *N* is odd. This logic has been applied in the following algorithm.

```
1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. Q ← N/2 (Integer division)
5. R ← N - Q * 2
6. IF R = 0 THEN
            PRINT "N IS EVEN"
7. IF R != 0 THEN
            PRINT "N IS ODD"
8. STOP
```

Explanation The primary aim here is to find out whether the remainder after the division of the number with 2 is zero or not. If the number is even the remainder after the division will be zero. If it is odd, the remainder after the division will not be zero. So by testing the remainder it is possible to determine whether the number is even or odd.

Explanation The primary aim here is to find out whether the remainder after the division of the number with 2 is zero or not. If the number is even the remainder after the division will be zero. If it is odd, the remainder after the division will not be zero. So by testing the remainder it is possible to determine whether the number is even or odd.

The first step indicates the starting point of the algorithm while the next prints a message asking for the entry of the number. In step 3, the number is kept in the variable N. N is divided by 2 in step 4. This operation being an integer division, the result is an integer. This result is assigned to Q. Any remainder that occurs is ignored. Now in step 5, the result Q is multiplied by 2 which obviously produces an integer that is either less than the value in N or equal to it. Hence in step 5 the difference between N and Q * 2 gives the remainder. This remainder value is then checked in step 6 and step 7 to either print out that it is either even or odd respectively. Step 8 just terminates the procedure.

9. Print the largest number among three numbers.

Solution Let the three numbers be represented by *A*, *B*, and *C*. There can be three ways of solving the problem. The three algorithms, with some differences, are given below.

```
1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A, B, C
4. IF A >= B AND B >= C
           THEN PRINT A
5. IF B >= C AND C >= A
           THEN PRINT B
   ELSE
           PRINT C
6. STOP
```

Explanation To find the largest among the three numbers A, B and C, A is compared with B to determine whether A is larger than or equal to B. At the same time it is also determined whether B is larger than or equal to C. If both these propositions are true then the number A is the largest otherwise A is not the largest. Step 4 applies this logic and prints A.

If A is not the largest number as found by the logic in step 4, then the logic stated in step 5 is applied. Here again, two propositions are compared. In one, B is compared with C and in the other C is compared with A. If both these propositions are true then B is printed as the largest otherwise C is printed as the largest.

Steps 1, 2, 3 and 6 needs no mention as it has been used in earlier examples.

Or

This algorithm uses a variable *MAX* to store the largest number.

```
1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A, B, C
4. MAX ← A
5. IF B > MAX THEN MAX ← B
6. IF C > MAX THEN MAX ← C
7. PRINT MAX
8. STOP
```

Explanation This algorithm differs from the previous one. After the numbers are stored in the variables A, B and C, the value of any one of these is assigned to a variable MAX. This is done in step 4. In step 5, the value assigned to MAX is compared with that assigned to B and if the value in B is larger only then it's value is assigned to MAX otherwise it remains unchanged. In step 6, the proposition " IF C > MAX " is true then the value in C is assigned to MAX. On the other hand, if the position is false then the value in MAX remains unchanged. So at the end of step 6, the value in MAX is the largest among the three numbers. Step 1 is the beginning step while step 8 is the terminating one for this algorithm.

Or

Here, the algorithm uses a **nested if** construct.

```
1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A, B, C
4. IF A > B THEN
     IF A > C THEN
      PRINT A
     ELSE
      PRINT C
     ELSE IF B > C THEN
      PRINT B
     ELSE
      PRINT C
5. STOP
```

Explanation Here, the nested if construct is used. The construct "IF p1 THEN action1 ELSE action2" decides if the proposition " p1" is true then action1 is implemented otherwise if it is false action2 is implemented. Now, action1 and action2 may be either plain statements like PRINT X or INPUT X or another "IF p2 THEN action3 ELSE action4" construct, were p2 is a proposition. This means that a second "IF p1 THEN action1 ELSE action2" construct can be interposed within the first "IF p1 THEN action1 ELSE action2" construct. Such an implementation is known as nested if construct.

Step 4 implements the nested if construct. First the proposition "A > B "is checked to find whether it is true or false. If true, the proposition "A > C " is verified and if this is found to be true, the value in A is printed otherwise C is printed. But if the first proposition "A > B" is found to be false then the next proposition that is checked is "B > C". At this point if this proposition is true then the value in B is printed whereas if it is false C is printed.

10. Take three sides of a triangle as input and check whether the triangle can be drawn or not. If possible, classify the triangle as equilateral, isosceles, or scalene.

Solution Let the length of three sides of the triangle be represented by *A*, *B*, and *C*. Two alternative algorithms for solving the problem are given, with explanations after each step, as follows:

1. `START`

   Step 1 starts the procedure.

2. `PRINT "ENTER LENGTH OF THREE SIDES OF A TRIANGLE"`

   Step 2 outputs a message asking for the entry of the lengths for each side of the triangle.

3. `INPUT A, B, C`

   Step 3 reads the values for the lengths that has been entered and assigns them to A, B and C.

4. `IF A + B > C AND B + C > A AND A + C > B THEN`
   `  PRINT "TRIANGLE CAN BE DRAWN"`
   `ELSE`
   `  PRINT "TRIANGLE CANNOT BE DRAWN": GO TO 6`

It is well known that in a triangle, the summation of lengths of any two sides is always greater than the length of the third side. This is checked in step 4. So for a triangle all the propositions "A + B > C", "B + C > A" and "A + C > B" must be true. In such a case, with the lengths of the three sides, that has been entered, a triangle can be formed. Thus, the message "`TRIANGLE CAN BE DRAWN`" is printed and the next step 5 is executed. But if any one of the above three propositions is not true then the message "`TRIANGLE CANNOT BE DRAWN`" is printed and so no classification is required. Thus in such a case the algorithm is terminated in step 6.

5. `IF A = B AND B = C THEN`
   `PRINT "EQUILATERAL"`
`ELSE`
   `IF A != B AND B != C AND C !=A THEN`
   `PRINT "SCALENE"`
`ELSE`
   `PRINT "ISOSCELES"`

After it has been found in step 4 that a triangle can be drawn, this step is executed. To find whether the triangle is an "`EQUILATERAL`" triangle the propositions "`A = B`" and "`B = C`" are checked. If both of these are true, then the message "`EQUILATERAL`" is printed which means that the triangle is an equilateral triangle. On the other hand if any or both the propositions "`A = B`" and "`B = C`" are found to be untrue then the propositions "`A != B`" and "`B != C`" and "`C !=A`" are checked.

If none of the sides are equal to each other then all these propositions are found to be true and so the message "SCALENE" will be printed. But if these propositions "A != B" and "B != C" and "C !=A" are false then the triangle is obviously an isosceles triangle and hence the message "ISOSCELES" is printed.

```
6. STOP
```

The procedure terminates here.

Or

This algorithm differs from the previous one and applies an alternate way to test whether a triangle can be drawn with the given sides and also identify its type.

```
1. START
2. PRINT "ENTER THE LENGTH OF 3 SIDES OF A
   TRIANGLE"
3. INPUT A, B, C
4. IF A + B > C AND B + C > A AND C + A > B
   THEN
          PRINT "TRIANGLE CAN BE DRAWN"
   ELSE
          PRINT "TRIANGLE CANNOT BE DRAWN"
              : GO TO 8
5. IF A = B AND B = C THEN
          PRINT "EQUILATERAL TRIANGLE"
              : GO TO 8
6. IF A = B OR B = C OR C = A THEN
          PRINT "ISOSCELES TRIANGLE"
              : GO TO 8
7. PRINT "SCALENE TRIANGLE"
8. STOP
```

Having followed the explanations given with each of the earlier examples, the reader has already understood how the stepwise representation of any algorithm of any problem starts, constructs the logic statements and terminates.

In a similar way the following example exhibits the stepwise representation of algorithms for various problems.

11. In an academic institution, grades have to be printed for students who appeared in the final exam. The criteria for allocating the grades against the percentage of total marks obtained are as follows.

Marks	***Grade***	***Marks***	***Grade***
91–100	O	61–70	B
81–90	E	51–60	C
71–80	A	<= 50	F

The percentage of total marks obtained by each student in the final exam is to be given as input to get a printout of the grade the student is awarded.

Solution The percentage of marks obtained by a student is represented by *N*. The algorithm for the given problem is as follows.

```
1. START
2. PRINT
   "ENTER THE OBTAINED PERCENTAGE MARKS"
3. INPUT N
4. IF N > 0 AND N <= 50 THEN
      PRINT "F"
5. IF N > 50 AND N <= 60 THEN
      PRINT "C"
6. IF N > 60 AND N <= 70 THEN
      PRINT "B"
7. IF N > 70 AND N <= 80 THEN
      PRINT "A"
8. IF N > 80 AND N <= 90 THEN
      PRINT "E"
9. IF N > 90 AND N <= 100 THEN
      PRINT "O"
10. STOP
```

12. Construct an algorithm for incrementing the value of a variable that starts with an initial value of 1 and stops when the value becomes 5.

Solution This problem illustrates the use of iteration or loop construct. Let the variable be represented by *C*. The algorithm for the said problem is given as follows.

```
1. START
2. C ← 1
3. WHILE C <= 5
4. BEGIN
5. PRINT C
6. C ← C + 1
7. END
8. STOP
```

While loop construct for looping till C is greater than 5

13. Write an algorithm for the addition of *N* given numbers.

Solution Let the sum of *N* given numbers be represented by *S*. Each time a number is given as input, it is assigned to the variable *A*. The algorithm using the loop construct 'if ... then goto ...' is used as follows:

```
1. START
2. PRINT "HOW MANY NUMBERS?"
3. INPUT N
4. S ← 0
5. C ← 1
6. PRINT "ENTER NUMBER"
7. INPUT A
8. S ← S + A
9. C ← C + 1
```

```
10. IF C <= N THEN GOTO 6
11. PRINT S
12. STOP
```

14. Develop the algorithm for finding the sum of the series 1 + 2 + 3 + 4 + ... up to *N* terms.

Solution Let the sum of the series be represented by *S* and the number of terms by *N*. The algorithm for computing the sum is given as follows.

```
1. START
2. PRINT "HOW MANY TERMS?"
3. INPUT N
4. S ← 0
5. C ← 1
6. S ← S + C
7. C ← C + 1
8. IF C <= N THEN GOTO 6
9. PRINT S
10. STOP
```

15. Write an algorithm for determining the sum of the series 2 + 4 + 8 + ... up to *N*.

Solution Let the sum of the series be represented by *S* and the number of terms in the series by *N*. The algorithm for this problem is given as follows.

```
1. START
2. PRINT "ENTER THE VALUE OF N"
3. INPUT N
4. S ← 0
5. C ← 2
6. S ← S + C
7. C ← C * 2
8. IF C <= N THEN GOTO STEP 6
9. PRINT S
10. STOP
```

16. Write an algorithm to find out whether a given number is a prime number or not.

Solution The algorithm for checking whether a given number is a prime number or not is as follows.

```
1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. IF N = 2 THEN
     PRINT "CO-PRIME" GOTO STEP 12
5. D ← 2
6. Q ← N/D (Integer division)
7. R ← N - Q*D
```

```
 8. IF R = 0 THEN GOTO STEP 11
 9. D ← D + 1
10. IF D <= N/2 THEN GOTO STEP 6
11. IF R = 0 THEN
     PRINT "NOT PRIME"
    ELSE
     PRINT "PRIME"
12. STOP
```

17. Write an algorithm for calculating the factorial of a given number *N*.

Solution The algorithm for finding the factorial of number *N* is as follows.

```
 1. START
 2. PRINT "ENTER THE NUMBER"
 3. INPUT N
 4. F ← 1
 5. C ← 1
 6. WHILE C <= N
 7. BEGIN
 8. F ← F * C
 9. C ← C + 1
10. END
11. PRINT F
12. STOP
```

While loop construct for looping till C is greater than *N*

17. Write an algorithm for calculating the factorial of a given number *N*.

Solution The algorithm for finding the factorial of number *N* is as follows.

```
 1. START
 2. PRINT "ENTER THE NUMBER"
 3. INPUT N
 4. F ← 1
 5. C ← 1
 6. WHILE C <= N
 7. BEGIN
 8. F ← F * C
 9. C ← C + 1
10. END
11. PRINT F
12. STOP
```

While loop construct for looping till C is greater than *N*

18. Write an algorithm to print the Fibonacci series up to *N* terms.

Solution The Fibonacci series consisting of the following terms 1, 1, 2, 3, 5, 8, 13, ... is generated using the following algorithm.

```
1. START
2. PRINT "ENTER THE NUMBER OF TERMS"
3. INPUT N
4. C ← 1
5. T ← 1
6. T1 ← 0
7. T2 ← 1
8. PRINT T
9. T ← T1 + T2
10. C ← C + 1
11. T1 ← T2
12. T2 ← T
13. IF C <= N THEN GOTO 8
14. STOP
```

19. Write an algorithm to find the sum of the series $1 + x + x^2 + x^3 + x^4 + \ldots$ up to *N* terms.

Solution

```
1. START
2. PRINT "HOW MANY TERMS"
3. INPUT N
```

```
4. PRINT "ENTER VALUE OF X"
5. INPUT X
6. T ← 1
7. C ← 1
8. S ← 0
9. S ← S + T
10. C ← C + 1
11. T ← T * X
12. IF C <= N THEN GOTO 9
13. PRINT S
14. STOP
```

20. Write the algorithm for computing the sum of digits in a number.

Solution

```
1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. S ← 0
5. Q ← N/10 (Integer division)
6. R ← N - Q * 10
7. S ← S + R
8. N ← Q
9. IF N > 0 THEN GOTO 5
10. PRINT S
11. STOP
```

21. Write an algorithm to find the largest number among a list of numbers.

Solution The largest number can be found using the following algorithm.

```
1. START
2. PRINT "ENTER,
       TOTAL COUNT OF NUMBERS IN LIST"
3. INPUT N
4. C ← 0
5. PRINT "ENTER THE NUMBER"
6. INPUT A
7. C ← C + 1
8. MAX ← A
9. PRINT "ENTER THE NUMBER"
10. INPUT B
11. C ← C + 1
12. IF B > MAX THEN
        MAX ← B
13. IF C <= N THEN GOTO STEP 9
14. PRINT MAX
15. STOP
```

22. Write an algorithm to check whether a given number is an Armstrong number or not. An Armstrong number is one in which the sum of the cube of each of the digits equals that number.

Solution If a number 153 is considered, the required sum is ($1^3 + 5^3 + 3^3$), i.e., 153. This shows that the number is an Armstrong number. The algorithm to check whether 153 is an Armstrong number or not, is given as follows.

```
1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. M ← N
5. S ← 0
6. Q ← N/10 (Integer division)
7. R ← N - Q * 10
8. S ← S + R * R * R
9. N ← Q
10. IF N > 0 THEN GOTO STEP 6
11. IF S = M THEN
      PRINT "THE NUMBER IS ARMSTRONG"
      ELSE PRINT "THE NUMBER IS NOT ARMSTRONG"
12. STOP
```

23. Write an algorithm for computing the sum of the series $1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots$ up to N terms.

Solution

```
1. START
2. PRINT "ENTER NUMBER OF TERMS"
3. INPUT N
4. PRINT "ENTER A NUMBER"
5. INPUT X
6. T ← 1
7. S ← 0
8. C ← 1
9. S ← S + T
10. T ← T * X/C
11. C ← C + 1
12. IF C <= N THEN GO TO STEP 9
13. PRINT S
14. STOP
```

TRANSLATOR

A translator or programming language processor is a generic term that can refer to anything that converts code from one computer language into another.

A program written in high-level language is called source program. These include translations between high-level and human-readable computer languages such as C++ and Java, intermediate-level languages such as Java bytecode, low-level languages such as the assembly language and machine code, and between similar levels of language on different computing platforms, as well as from any of the above to another.

There are 3 different types of translators as follows:

Compiler

A compiler is a translator used to convert high-level programming language to low-level programming language. It converts the whole program in one session and reports errors detected after the conversion. The compiler takes time to do its work as it translates high-level code to lower-level code all at once and then saves it to memory. A compiler is processor-dependent and platform-dependent. It has been addressed by alternate names as the following: special compiler, cross-compiler and, source-to-source compiler.

Interpreter

The interpreter is similar to a compiler, as it is a translator used to convert high-level programming language to low-level programming language. The difference is that it converts the program one line of code at a time and reports errors when detected, while also doing the conversion. An interpreter is faster than a compiler as it immediately executes the code upon reading the code. It is often used as a debugging tool for software development as it can execute a single line of code at a time. An interpreter is also more portable than a compiler as it is processor-independent, you can work between different hardware architectures

Assembler

An assembler is a translator used to translate assembly language into machine language. It has the same function as a compiler for the assembly language but works like an interpreter. Assembly language is difficult to understand as it is a low-level programming language. An assembler translates a low-level language, such as an assembly language to an even lower-level language, such as the machine code.

PROGRAMMING TECHNIQUES

STRUCTURED PROGRAMMING

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

- C
- C++
- Java
- C#

..etc

On the contrary, in the Assembly languages like Microprocessor 8085, etc, the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

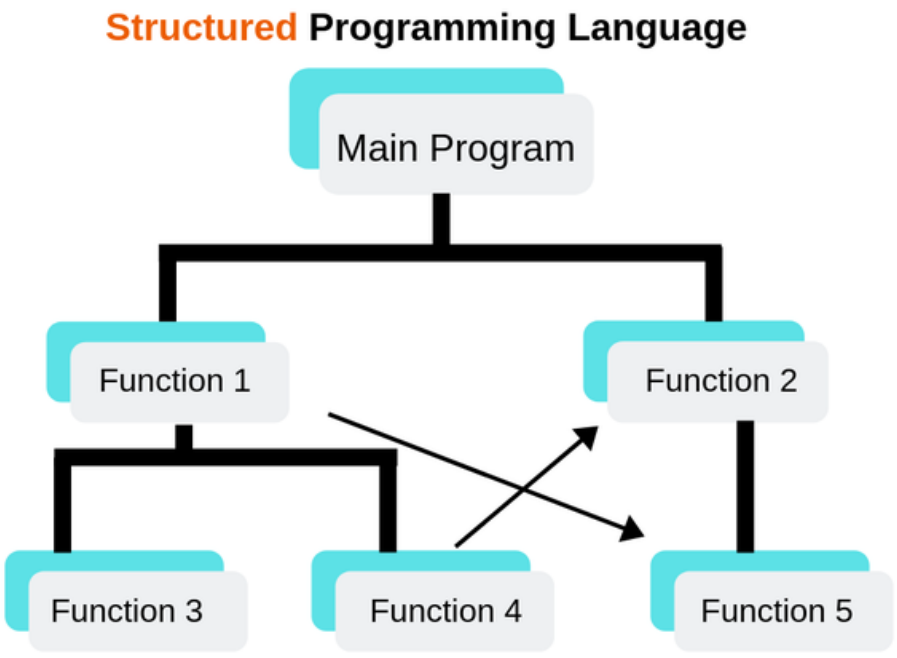
The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Advantages of Structured Programming Approach:

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages of Structured Programming Approach:

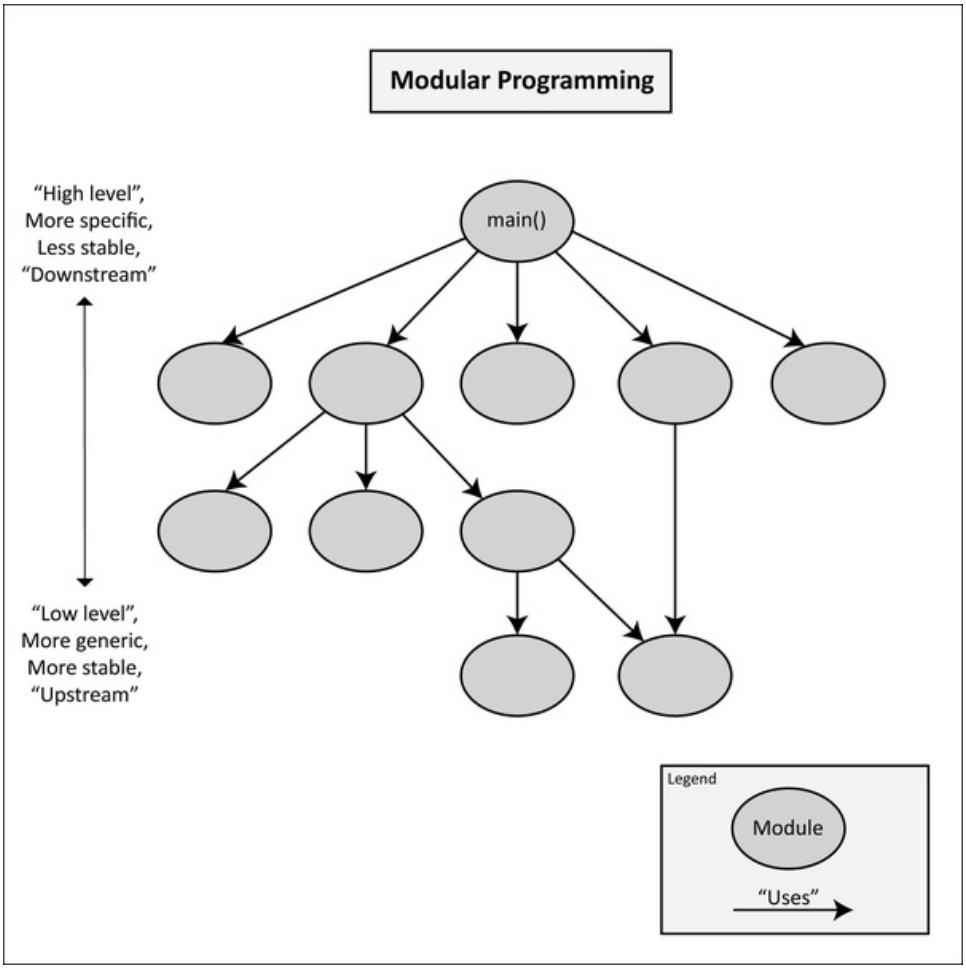
1. Since it is Machine-Independent, So it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4. Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.



MODULAR PROGRAMMING

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of **modular programming** approached.
- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.



Points which should be taken care of prior to modular program development:

1. Limitations of each and every module should be decided.
2. In which way a program is to be partitioned into different modules.
3. Communication among different modules of the code for proper execution of the entire program.

Advantages of Using Modular Programming Approach –

1. **Ease of Use :**This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability :**It allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance :** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.