2.1 Algorithm:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. **Algorithms** are generally created independent of underlying languages, i.e. an **algorithm** can be implemented in more than one programming language

Standard Defination: " An algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm".

Control structures used in algorithms

An algorithm has finite number of steps and some steps may involve decision-making and repetition. Broadly speaking, an algorithm uses three control structures, namely **sequence, decision, and repetition.**

Sequence: sequence means that each step of the algorithm is executed in the specified order.

Decision: decision statements are used when the execution of a process depends on the outcome of some condition. A condition in this context is any statement that may evaluate to either a true value or a false value. In decision statements using conditions are TRUE, FALSE, IF, ELSE, IF-ELSE**.**

Repetition: Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as WHILE, DO-WHILE, and FOR loops. These loops execute one or more steps until some condition is true.

Advantages of Algorithms:

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

Disadvantages of Algorithms:

1. Algorithms is Time consuming.
2. Difficult to show Branching and Looping in Algorithms.
3. Big tasks are difficult to put in Algorithms.

Characteristics of Algorithms:

1. **Precision** – the steps are precisely stated(defined).
2. **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
3. **Finiteness** – the algorithm stops after a finite number of instructions are executed.
4. **Input** – the algorithm receives input.
5. **Output** – the algorithm produces output.
6. **Generality** – the algorithm applies to a set of inputs.

Write an algorithm to add two numbers entered by the user..

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

Write an algorithm to find the largest among three different numbers entered by the user.

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

If a > c

Display a is the largest number.

Else

```
        Display c is the largest number.
    Else
      If b > c
        Display b is the largest number.
      Else
        Display c is the greatest number.
Step 5: Stop
```

2.2 Flowchart:

A **flowchart** is a type of diagram that represents a workflow or process. A **flowchart** can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. The **flowchart** shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows

or

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. Each step in the sequence is noted within a diagram shape. Steps are linked by connecting lines and directional arrows. This allows anyone to view the flowchart and logically follow the process from beginning to end.

A flowchart is a powerful business tool. With proper design and construction, it communicates the steps in a process very effectively and efficiently.

Terminal: The terminal (oval) symbol as the name implies is used to indicate the beginning (START), ending (STOP) and pause (HALT) in the program logic flow. It is the first and last symbol in the program logic.

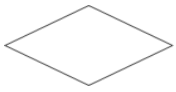
Input/Output: All Input/output instructions in the program are indicated with Input/output (Parallelogram) Symbol

.

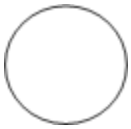
Processing: All Arithmetic and data movement instructions in the program are indicated with processing (Rectangle) Symbol.

Flow Lines: Flow lines with arrow heads are used to indicate the flow of operation i.e., 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.

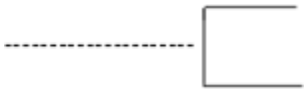
Decision: The decision (Diamond) Symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternate points is possible.



Connectors: If a flowchart becomes very long the flow lines start crossing at many places that causes confusion and reduces understandability of the flowchart. Also when a flowchart becomes a too long to fit in a single page and the use of flow lines become impossible. A connector symbol is represented by a circle. A pair of identically labeled connector symbols is commonly used to indicate a continued flow when the use of a line is confusing.



Annotation: The annotation symbol is used in flowcharts to indicate the descriptive comments or explanation of the instruction.



Advantages of Flowcharts

1. It helps to understand the flow of program control in an easy way.

2. Developing program code by referring its flow chart is easier in comparison to developing the program code from scratch.

3. It helps in avoiding semantic errors.

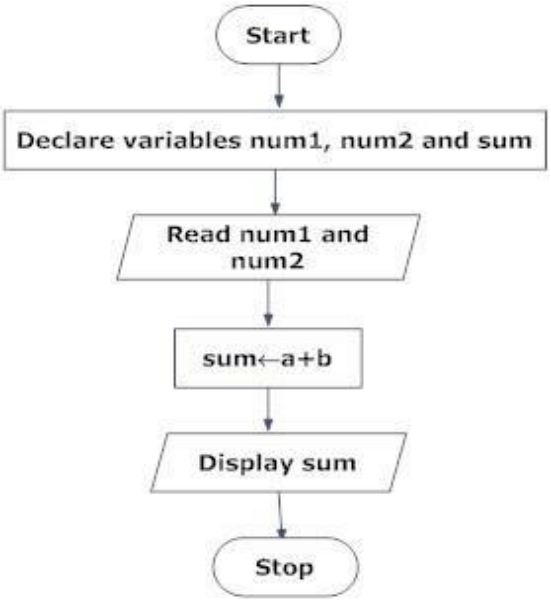
4. A flowchart acts as documentation for the process or program flow.

5. The use of flowcharts works well for small program design

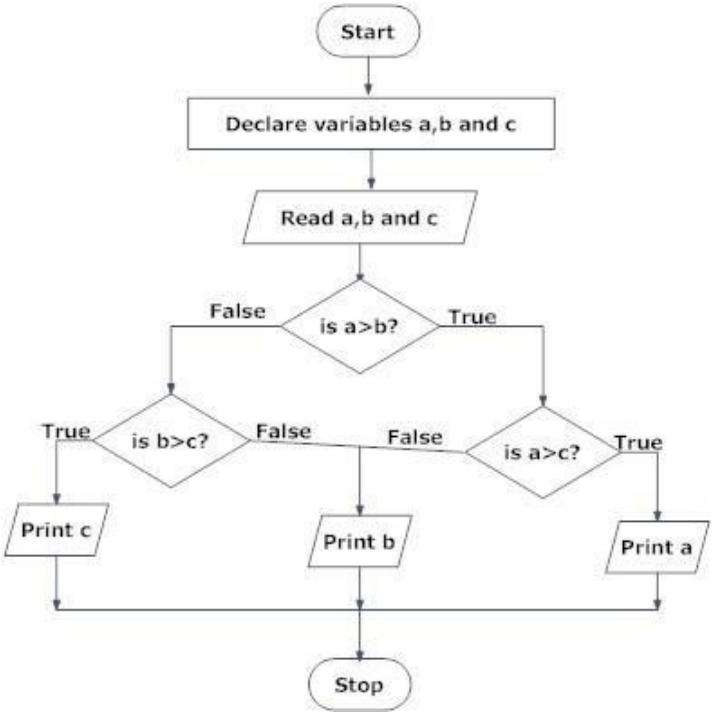
Disadvantages of Flowcharts

- Drawing flowcharts is a laborious and time consuming activity. Just imagine the effort required to draw a flowchart of a program having 50000 statements in it.
- Often, the flowchart of complex program becomes complex and clumsy.
- At times, a little bit of alteration in the solution may require complete re-drawing of the flowchart.
- The essential of what is to be done may get lost in the technical details of how it is done.
- They are no well-defined standards that limit the details that must be incorporated into a flowchart.

Draw a flowchart to add two numbers entered by user.



Draw flowchart to find the largest among three different numbers entered by user.



Looping:

In a programming, loop is a programming structure that repeats a sequence of instructions until a specific condition is met.
Loops are supported by all modern programming languages, though their implementations and syntax may differ. Two of the most common types of loops are the

1. For loop:
For loop is a programming language conditional iterative statement which is used to check for certain conditions and then repeatedly execute a block of code as long as those conditions are met.

The for loop is distinguished from other looping statements through an explicit loop counter or loop variable which allows the body of the loop to know the exact sequencing of each iteration.

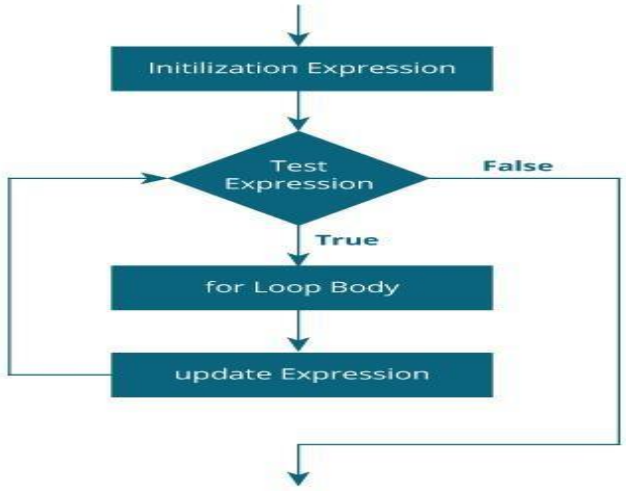
Syntax of the for loop

```
1. . for (initializationStatement; testExpression; updateStatement)
2. {
3. // statements inside the body of loop
4. }
```

How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
- Again the test expression is evaluated.
- This process goes on until the test expression is false. When the test expression is false, the loop terminates.
- To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

for loop Flowchart



While Loop: A **while loop** is one of the most common types of loop. The main characteristic of a while loop is that it will repeat a set of instructions based on a condition. As far as the loop returns a boolean value of TRUE, the code inside it will keep repeating. We use this kind of loop when we don't know the exact number of times a code needs to be executed.

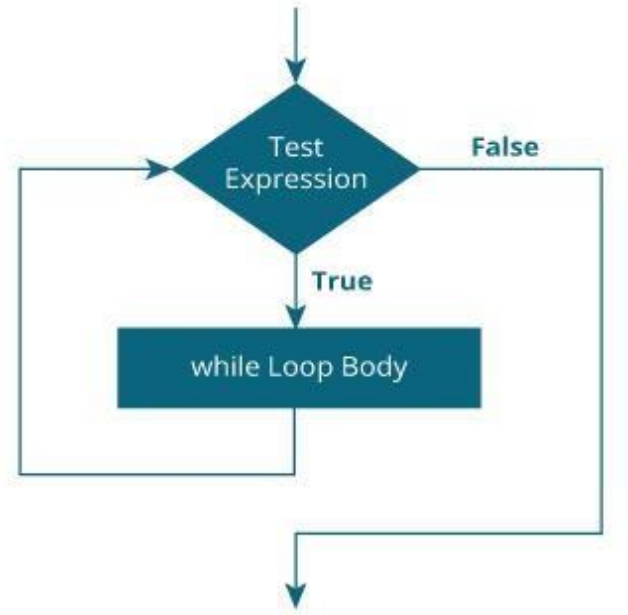
The syntax of the while loop is:

```
While(test Expression)

{

// statements inside the body of the loop

}
```

How while loop works?

- The while loop evaluates the test expression inside the parenthesis ().
- If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.
- If the test expression is false, the loop terminates (ends).
  To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

Flowchart of while loop



Do-while:

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

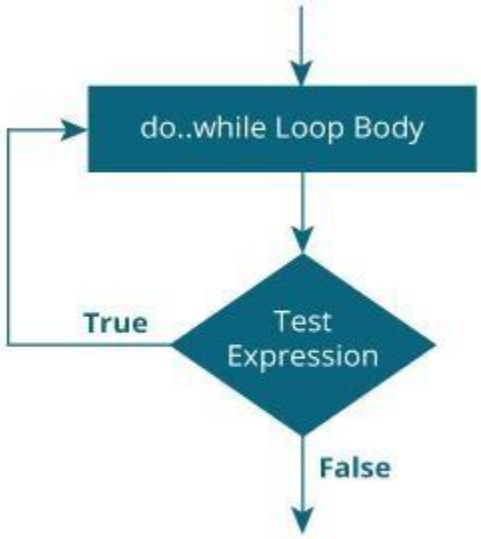
```
1. do
2. {
3. // statements inside the body of the loop 4. }

5. while (testExpression);
```

How do...while loop works?

- The body of do...while loop is executed once. Only then, the test expression is evaluated.
- If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
- This process goes on until the test expression becomes false.

- If the test expression is false, the loop ends.

Flowchart of do...while Loop



Some Programming Features:

Character set:

A computer language uses a set of characters to write a program refered as character set. A character set may also be referred to as character map, charset or character code. As every language contains a set of characters used to construct words, statements, etc., programming language also has a set of characters which include **alphabets, digits**, and **special symbols**.

Every program language contains statements. These statements are constructed using words and these words are constructed using characters from character set. Program language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

Alphabets

C language supports all the alphabets from the English language. Lower and upper case letters together support 52 alphabets.

lower case letters - **a to z**

upper case letters - **A to Z**

Digits

Programming language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols

Programming language supports a rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, backspaces, and other special symbols.

Special Symbols - **~ @ # $ % ^ & * ( ) _ - + = { } [ ] ; : ' " / ? . > , < , //**

CONSTANTS:

In **computer** programming, a **constant** is a value that cannot be altered by the program during normal execution. The digits or sequence of digits are also constants. Usually constants signify a real number which has special functionalities in the background of the problem or the scenario in which it is being used.

Constants can also be used to represent decimals or irrational numbers of interest, as well as very large numbers, which are not easily manipulated in a mathematical expression, in its fully fleshed-out numerical representation.

Types of constants:

1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

Constant type	data type (Example)
Integer constants	int (53, 762, -478 etc) unsigned int (5000u, 1000U etc) long int, long long int (483,647 2,147,483,680)
Real or Floating point constants	float (10.456789) doule (600.123456789)
Octal constant	int (Example: 013 /*starts with 0 */)
Hexadecimal constant	int (Example: 0x90 /*starts with 0x*/)
character constants	char (Example: 'A', 'B', 'C')
string constants	char (Example: "ABCD", "Hai")

VARIABLES:

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used throughout your program.

RULES FOR VARIABLE:

1. Variable name must begin with letter or underscore.
2. Variables are case sensitive
3. They can be constructed with digits, letters.
4. No special symbols are allowed other than underscore.
5. sum, height, _value are some examples for variable name

Type	Syntax
Variable declaration	data_type variable_name; Example: int x, y, z; char flat, ch;
Variable initialization	data_type variable_name = value; Example: int x = 50, y = 30; char flag = 'x', ch='l';

COMMENTS:

In computer programming, a **comment** is a programmer-readable explanation or *annotation* in the source code of a computer program. They are added with the purpose of making the source code easier for humans to understand, and are generally ignored by compilers and interpreters

Comments are sometimes processed in various ways to generate documentation external to the source code itself by documentation generators, or used for integration with source code management systems and other kinds of external programming tools.

Ex:

// are used for single line comments

/* multiple line */

[ ] can be used for algorithms as comments

Pseudo code:

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program's flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly

Advantages:

- Can be done easily on a word processor
- Easily modified
- Implements structured concepts well
- Clarify algorithms in many cases.
- Impose increased discipline on the process of documenting detailed design
- Provide additional level at which inspection can be performed
- Help to trap defects before they become code.
- Increases product reliability.
- May decreases overall costs
- It can be easily modified as compared to flowchart
- Its implementation is very useful in structured design elements.

- It can be written easily.
- It can be read and understood easily
- Converting a pseudocode to programming language is very easy as compared with converting a flowchart to programming language.

Disadvantages:

- It's not visual
- Create an additional level of documentation tomaintain.
- Introduce error possibilities in translating to code.
- May require tool to extract pseudocode and facilitatedrawing flowcharts.
- There is no accepted standard, so it varies widely fromcompany to company
- We do not get a picture of the design.
- There is no standardized style or format, so one pseudocode may be different from another
- For a beginner, it is more difficult to follow the logicor write pseudocode as compared to flowchart.

The one-zero game:

We will now look at an algorithm designed to enable a user to choose the best strategy for playing the one-zero game**.**

One-zero game can played by several players using a 6-sided dice. Each person takes turns at throwing the dice. On each turn, a player may throw the dice as many times as she wishes and her scores for that turn will normally be sum of the values thrown. however, the catch is that , if she ever throws a 1, her turn ends and her score for that turn is 0.obviously, the longer one keeps on throwing the dice, the greater the chance of throwing a 1. A game consists of a predetermined accumulates the most points.

There are a numbers of strategies a player can adopt for her turn.

1.she can play it by 'feel'; whenever she feels it is time to stop, she stops(unless, of course, she is forced to by throwing a1)

2. she can set a maximum number of points, say 15, for each turn. If her points tally for a turn reaches or exceeds 15, she stops.

3. she can set a maximum number of throws , say 5, for each turn. If she makes 5 throws(without throwing a 1),she stops.

We will develop an algorithm to test strategy(3). The user will specify the maximum number of throws she wishes to make each turn, and the number of turns for which a game will last . for

example , she may wish to find out what her score is likely to be if she uses a maximum of 5 throws for each of 20 turns. The algorithm is an example of a more general topic called simulation,

In this example, we have to get the computer to mimic the throwing of a dice . when we throw a dice in real life, we get a number from 1 to 6. The numbers on successive throws do not come in any particular order ; we say they come in random order; for example:

3,5,1,2,3,6,4,4,2,etc.

Thus we must get the computer to give us random numbers from 1 to 6. Most computers contain what is called a 'random number generator'.

There are many algorithms for generating random numbers, and each computer will use a chosen one. From the user's point of view, the particular method used to generate the random numbers is not important; what is important is how to use the facility on any given computer. We will assume that the instruction(or function):

RANDOM (1,6)

Gives us a random number from 1 to 6. We can use it as in:

Set **Throw Value** to RANDOM (1, 6)

Each time this instruction is executed, a new random number from 1 to 6 is stored in **Throw Value.** For example, suppose we want to simulate 10 throws of a dice and print the values thrown. This can be accomplished by the following:

FOR **L=1 TO** 10 DO

Set **Throw Value** to RANDOM (1, 6)

Print (**Throw Value**)

ENDFOR

When the above is executed, the following values may be printed:

3 5 1 2 3 6 4 4 2 6

Some structured programming concepts:

We can illustrate some of the concepts of structured programming by developing an algorithm for the following problem:

Problem:

Request the user for the number of turns in a game and the maximum number of throws per turn. Simulate the playing of the game, using strategy (3), above, and print the score obtained for the game.

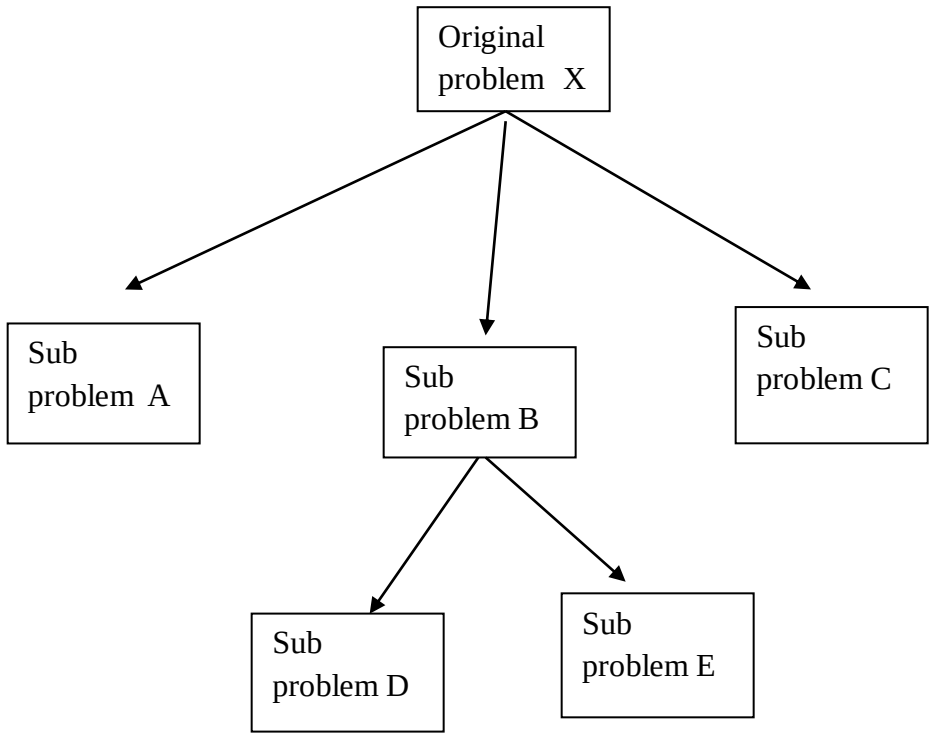
The concepts that will be illustrated are:

1. top-down program design (using modules);
2. Stepwise refinement (of a module).

Top –down design:

The goal of top –down design is to divide a given problem into sub problems . A sub problem, in turn, can be thought of as simply another problem, so it can be divided into sub problems, each of which can be further subdivided, and so on

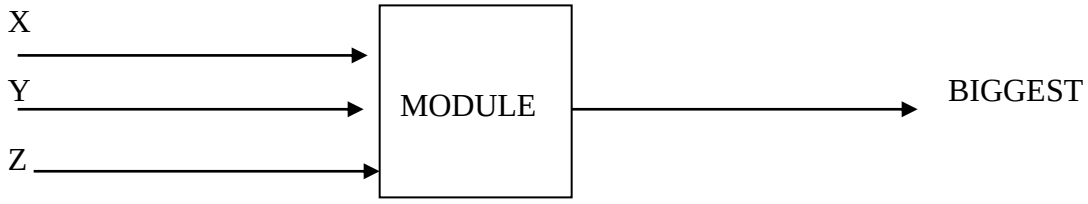
Figure 6.11 shows that, in order to solve problem X, we must solve sub problem A,B and C; but in order to solve sub problem B, we must solve sub problems D and E. The division of problems starts at the top level, and we then work our way down. This is why this process is called **top-down design.**



Each module should do a single, self-contained job. We should be able to describe its action in a sentence or two, similar to the following:

Given the data X, Y, AND Z, this module finds the largest and sets BIGGEST to this value.

We should be able to view a module and what comes out, without worrying too much about what takes place inside. For example



Advantages of using modules:

1. They enable a program to be developed in stages; the programmer can concentrate on one task at a time.
2. They allow a large program to be written by several people.
3. Because a module performs a single, well- defined task, there is the creation of a user library, in which commonly used modules are placed.
4. Progress on a large project can be more easily measured.
5. Modules can be tested individually.
6. Program modification is easier, since changes can be isolated within specific modules.

Testing a modular program:

The job of testing a computer program is much simplified if it is written using modules. It is then possible to test each module separately.

Module testing goes through the following stages:

1. When the algorithm is developed, it is tested 'by hand,; that is, the programmer 'executes' the instructions the way the computer would, noting the effect of statements on the data. Many errors in logic can be discovered this way. When the programmer is sure that the algorithm is correct, go to stage2**.**
2. The algorithm is coded in the chosen programming language.

3. The program checking used to 'interpreted' or 'compiler'. An interpreted or compiler is just another computer program which can check user's programs for errors. Any syntax errors are found at this stage. These are corrected and step3 repeated until all these errors have been removed.
4. **The program** is run using appropriate 'test data'. This involves using data where the correct results are known. These results from running the program are noted, if they are incorrect, then it means the program still contains large errors. The programmer must then study the algorithm and /or program listening (using the data which produced the incorrect results) and try to discover what is wrong – this is called **debugging**

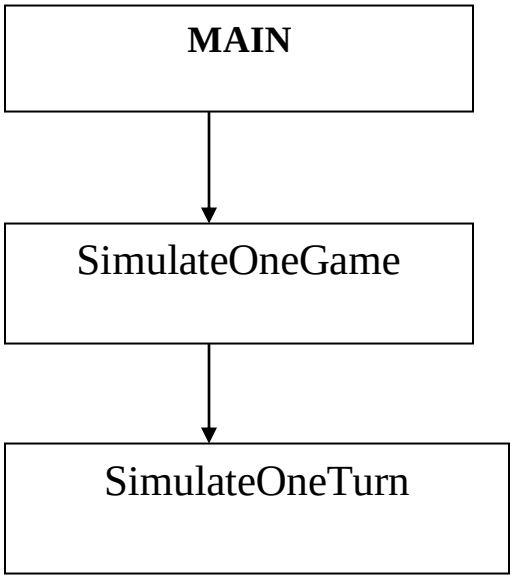
Even if the results from the test data are correct, it doesn't necessarily mean that all bugs have been removed. It is still possible that a logical error exists, but was not discovered because the test data was not comprehensive enough. Test data ought to check all possible paths through the program by limiting the size of the module the job of creating test data for it can be simplified, and we can have more confidence that it is bug – free.

Module for one-zero

When a program is being designed, the functions that need to be performed become the programs module. Consider our problem of simulating the one-zero game. In order to play a game (consisting of a number of terms), we need to solve the problem of how to simulate one turn the functions which must be performed are:

Get the data(number of terms for game maximum throws for turn)
Simulate a game(consisting of many turn)
Simulate one turn (consisting of many throws)
Print results

The indentation indicates the level of the function to be performed; at the outer most (or top) level, we need a module to get data and print results. We will call this the main module. Below the main module is the one which simulates an entire game . At the lowest level is the module to simulate a signal turn. Our solution will consist of three modules



The diagram also illustrated which modules depend on other modules. For instance simulate one game will need the one service of simulate one turn.

As well as specifying the modules required we need to say how modules communicative with a each other. For example, when simulate one game calls into service simulate one turn, it must indicate the maximum number of throws allowed on a turn, and when simulate one turn is finished, it must inform simulate one game what score was obtained for the turn. This problem is partly solved when we difine the module, for then we specify what the module is given and what is produce (on 'turns'). When developing our modules, we will follow these guidelines:

(1) Each module is given a, by which other modules can refer to it. We will call the highest-level module main.

(2) Foe each module, we must specify what values are given and what values are returned. Foe example, the module simulate one turn is given the maximum number of throws for turn and it returns the score obtained for that turn.

(3) Suppose module X requires the services of module Y.

(4) When we get to the end of a module, control returns to the point from which the module was called

Using these guidelines, we can give the specifications of the three modules needed to simulate the one-zero game

(I) MODULE main

Get the data

Call upon other modules, as necessary

Print the results

Stop

ENDMODULE

(II) MODELU **Simulate One Turn**

Given the number of turns in game, and the maximum number of throws for one turn, this module simulates a game and returns the score obtained for the game.
ENDMODULE

(iii). MODULE **Simulate One Turn**

Given the maximum number of throws for one turn,

this module simulates one turn and returns the score obtained for the turn.

END MODELU

Step Wise Refinement

Stepwise refinement is a technique used in developing the internal workings of a module. As the name implies, we start with asset of general steps; we then go through a process in which these steps are expanded(refined);this refinement continues until each step is simply a program statement. We can look at this procedure by developing the algorithms for our three modules in the One- Zero game.

1. Module main

MODULE main
Get the data
Call upon other modules, as necessary Print results
Stop

ENDMODULE.

This module consists of three general steps. We can expand `Get the data' into:

Prompt the user for the number of turns per game, MaxTurns
Prompt the user for the maximum number of throws per turn, **MaxThrowsPerTurn**

2. Module Simulate One Game : Since **Main** would need to know the score obtained for the game, it must also pass a variable to the module so that the module can return the result; the variable used is **Game Score**. The refinement gives:

Simulate One Game (Max Turns , Max Throws Per turn, Game Score)

Finally `print the results' is expanded into:

Print("This game consisted of", **Max Turns,"** turns ")

Print("Each turn consisted of a maximum of", **Max Throws per turn ,**"Throws')

Print ("The score obtained for the game was", **Game Score)**

3. Module Simulate One Turn

To make the discussion easier to follow, we will assume that the value of **Max Throws per turn** is 5 .When logic has been developed, we will replace 5 by **Max Throws per turn**

MODULE **Simulate One Turn (Max Throws Per Turn, score This Turn)**
Initialize
Process a maximum of 5 throws END
MODULE

One obvious variable which needs to be initialized is **Score This Turn** , since the score is accumulated , throw by throw . In this module, another variable needs to be initialized. However, This will not become apparent until after we have refined `process a maximum of 5 throws `. For this reason, it is sometimes a good idea to leave the expansion of `Initialize ' Until after the other steps have been expanded.

A first attempt at expansion might give:

WHILE Number of Throwns < 5 DO
Throw the dice
Process the throw END
WHILE

Since the WHILE condition does not make sense unless **Number of Throws** has been give a value, this variable would need to be initialized; we would initialize ;we would initialize it to 0 . Also , each time a throw is made, we would need to add 1 to **Number of Throws**.

Our expansion now becomes:
Set Number of Throws to 0 WHILE
Number of Throws <5 DO
Throw the dice
Add 1 to **Number of Throws**

Process the throw END WHILE

MODULE **MAIN**
Prompt the user for the number of turns per game, **Max Turns**
Prompt the user for the maximum number of Throws per turn ,
MaxThrowsPerTurn
WHILE **MaxThrowsPerTurn**>0 DO
SimulateOneGame (MaxTurns, MaxThrowsPerTurn, Game Score)
Print ("this game consisted of ", **Max Turns,"** turns")

Print ("each turn consisted of a maximum of", **MaxThrowsPerTurn**, "throws")
Print ("the score obtained for the game was" **Game Score**)
ENDWHILE
STOP
ENDMODUE

MODULE **SimulateOneGame (MaxTurns, MaxThrowsPerTurn, Game Score)**
Set **GameScore to 0**
FOR **Turn = 1 TO MaxTurns**
SimulateOneTurn (**MaxThrowsPerTurn, ScoreThisTurn**)
Add **ScoreThisTurn** to **GameScore**
ENDFOR
ENDMODULE

MODULE **SimulateOneTurn** (MaxThrowsPerTurn, ScoreThisTurn)
Set **ScoreThisTurn** to 0
Set **NumberOfThrows** to 0
WHILE **NumberOfThrows** < MaxThrowsPerTurn
Set ThrowValue to RANDOM(1,6)
Add 1 to NumberOfThrows
IF Throw value=1 THEN
Set **ScoreThisTurn** to 0
Set **NumberOfThrows** to **MaxThrowsPerTurn** [force loopexit]
Else
Add **Throwsvalue** to **ScoreThisTurn**
ENDIF
ENDWHILE
END MODULE

Programming language:

A Programming language is a language specifically designed to express computations that can be performed by computer. Programming languages are used to create programs that control the behaviour of system or to express algorithms.

Usually, Programming language have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. Syntax refers to the grammar of language, that is, the rules used for constructing the permitted phrases of the languages, where as semantics expresses the meanings these phrases.

There are basically two types of computer programming languages given below:

1. Low level language
2. High level language

Low Level Languages

A low-level language is a programming language that deals with a computer's hardware components and constraints. It has no (or only a minute level of) abstraction in reference to a computer and works to manage a computer's operational semantics.

A low-level language may also be referred to as a computer's native language. The programming languages that are very close to machine code (0s and 1s) are called low-level programming languages. The program instructions written in these languages are in binary form.

Low-level languages are designed to operate and handle the entire hardware and instructions set architecture of a computer directly.

Low-level languages are considered to be closer to computers. Programs and applications written in a low-level language are directly executable on the computing hardware without any interpretation or translation.

low- level languages are divided into :

- machine language
- assembly language

Machine Language

Machine language is also known as first generation of programming language. Machine language is the fundamental language of the computer and the program instructions in this language is in the binary form (that is 0's and 1's).

Machine Language was used to program the first stored program computer system. This is the lowest level of programming language and is the only language that a computer understands.

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 0s and 1s. Although machine languages programs are typically displayed with the binary numbers represented in octal or hexadecimal number systems, these programs are not easy for programmers to read, write, or debug.

The main advantage of machine language is that the execution of the code is very fast and efficient since it is directly executed by the CPU. Machine language is difficult to learn and is far more difficult to debug if errors occur. The code written in machine language is not portable, and to transfer the code to a different computer, it needs to be completely rewritten since the machine language for one computer could be significantly different from that for another computer.

Advantage of Machine Language:

- Code can be directly executed by the computer.
- Execution is fast and efficient.
- Program can be written to efficiently utilize memory
- No translation program is required for the CPU.

Disadvantage of Machine Language

Here are some of the main disadvantages of machine languages:

- **Machine Dependent** - the internal design of every computer is different from every other type of computer, machine language also differs from one computer to another. Hence, after becoming proficient in the machine language of one type of computer, if a company decides to change to another type, then its programmer will have to learn a new machine language and would have to rewrite all existing program.
- **Difficult to Modify** - it is difficult to correct or modify this language. Checking machine instructions to locate errors is very difficult and time consuming.
- **Difficult to Program** - a computer executes machine language program directly and efficiently, it is difficult to program in machine language. A machine language programming must be knowledgeable about the hardware structure of the computer.

Assembly Language

It is another low-level programming language because the program instructions written in this language are close to machine language. Assembly language is also known as second generation

of programming language. With assembly language, a programmer writes instructions using symbolic instruction code instead of binary codes.

Symbolic codes are meaningful abbreviations such as SUB is used for substation operation, MUL for multiply operation and so on. Therefore this language is also called the low-level symbolic language. The set of program instructions written in assembly language are also called as mnemonic code. Assembly language provides facilities for controlling the hardware.

The code written in assembly language will be very efficient in terms of execution time and main memory usage, as the language is similar to machine language. The programs written in assembly language need to a translator, often know as the assembler, to convert them into machine language .

Advantage of Assembly Language

Here are some of the main advantages of using assembly language:

- **Easy to understand and use** - due to the use of mnemonic instead of numeric op-codes and symbolic names for data location instead of numeric addresses, it is much easier to understand and use in contrast with machine language.
- **Easier to locate and correct errors** - the programmers need not to keep track of storage location of the data and instruction, fewer errors are made while writing programs in assembly language and those that are made, are easier to find and correct.
- **Easy to modify** - assembly language are easier to understand, it is easier to locate, correct and modify instruction of an assembly language program.
- **Efficiency of machine language** - an assembly language program will be just as long as the resulting machine language program. Hence, leaving out the translation time required by the assembler, the actual execution time for an assembly language program and its equivalent machine language program.

Disadvantage of Assembly Languages

And here are some of the main disadvantages of using assembly language:

- **Machine dependent** - each instructions of assembly language program is translated into exactly one machine language instruction, an assembly language programs are dependent on machine language.
- **Knowledge of hardware required** - assembly languages are machine dependent, an assembly language programmer must have a good knowledge of characteristics and logical structure of his/her computer to write a good assembly language computer code.

- **Machine level coding** - assembly language instruction is substituted for one machine language instruction. Hence like machine language programs, write assembly language program is also time consuming and difficult.

High Level Language

High-level languages are designed to be used by the human operator or the programmer. They are referred to as "closer to humans." In other words, their programming style and context is easier to learn and implement than low-level languages, and the entire code generally focuses on the specific program to be created.

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context and is generally independent of the computer's hardware architecture.

A high-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and registers utilization.

A high-level language does not require addressing hardware constraints when developing a program. A translator is needed to translate the instructions written in a high-level language into the computer – executable machine language .such translators are commonly know as interpreters and compilers. Each high-level language has many compilers, and there is one for each type of computer.

3rd generation languages make it easy to write and debug a program and give programmers more time to think about its overall logic. Programs written in such languages are portable between machines.

The high level programming languages are further divided into:

- Procedural languages
- Non procedural languages
- Object oriented programming languages

Advantages of High Level Languages

There are several advantages of high level programming languages. The most important advantages are:

- **Easy to learn** - the high level languages are very easy to learn than low level languages. The statements written for the program are similar to English-like statements.

- **Easy to understand** - the program written in high level language by one programmer can easily be understood by another because the program instructions are similar to the English language.
- **Easy to write program** - in high level language, a new program can easily be written in a very short time. The larger and complicated software can be developed in few days or months.
- **Easy to detect and remove errors** - the errors in a program can be easily detected and removed. mostly the errors are occurred during the compilation of new program.
- **Built-in library functions** - Each high level language provides a large number of built-in functions or procedures that can be used to perform specific task during designing of new programs. In this way, a large amount of time of programmer is saved.
- **Machine Independence** - program written in high level language is machine independent. It means that a program written in one type of computer can be executed on another type of computer.

Limitation of High Level Language

There are two main limitation of high level languages are:

- **Low efficiency** - a program written in high level languages has lower efficiency than one written in a machine/assembly language to do the same job. That is, program written in high level languages result in multiple machine language instruction that may not be optimize, taking more time to execute and requiring more memory space.
- **Less flexibility** - high level languages are less flexible than assembly languages because they do not normally have instructions or mechanism to control a computer's CPU, memory and register.

Procedural Language

A procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program.

Procedural language is also known as imperative language.

A **procedural language** is a computer programming language that follows, in order, a set of commands. Examples of computer procedural languages are BASIC, C, FORTRAN, Java, and Pascal.

Procedural languages are some of the common types of programming languages used by script and software programmers. They make use of functions, conditional statements, and variables to create programs that allow a computer to calculate and display a desired output.

Using a procedural language to create a program can be accomplished by using a programming editor or IDE, like Adobe Dreamweaver, Eclipse, or Microsoft Visual Studio. These editors help users develop programming code using one or more procedural languages, test the code, and fix bugs in the code.

A procedure is a sequence of instructions having a unique name. The instructions of the procedure are executed with the reference of its name.

Non Procedural Language

Non procedural programming languages are also known as fourth generation languages. In non procedural programming languages, the order of program instructions is not important. Ihe importance is given only to, what is to be done.

With a non procedural language, the user/programmer writes English like instructions to retrieve data from databases. These languages are easier to use than procedural languages. These languages provide the user-friendly program development tools to write instructions. The programmers have not to spend much time for coding the program.

The most important non procedural languages and tools are discussed below:

- **SQL** - it stands for structured query language. it is very popular database access language and is specially used to access and to manipulate the data of databases. The word query represents that this language is used to make queries (or enquiries) to perform various operations on data of database. However, SQL can also be used to create tables, add data, delete data, update data of database tables etc.
- **RPG** - it stands for report program generator. This language was introduced by IBM to generate business reports. Typically, RPG is used for application development on IBM midrange computers, such as AS/400.

Object Oriented Programming Languages

The object oriented programming concept was introduced in the late 1960s, but now it has become the most popular approach to develop software.

In object oriented programming, the software is developed by using a set of interfacing object. An object is a component of program that has a set of modules and data structure. The modules are also called methods and are used to access the data from the object. The modern technique to design the program is object oriented approach. It is a very easy approach, in which program designed by using objects. Once an object for any program designed, it can be re-used in any other program.

- **Abstraction:** The process of picking out (abstracting) common features of objects and procedures.
- **Class:** A category of objects. The class defines all the common properties of the different objects that belong to it.
- **Encapsulation:** The process of combining elements to create a new entity. A procedure is a type of encapsulation because it combines a series of computer instructions.
- **Information hiding:** The process of hiding details of an object or function. Information hiding is a powerful programming technique because it reduces complexity.
- **Inheritance:** a feature that represents the "is a" relationship between different classes.
- **Interface:** the languages and codes that the applications use to communicate with each other and with the hardware.
- **Messaging:** Message passing is a form of communication used in parallel programming and object-oriented programming.
- **Object:** a self-contained entity that consists of both data and procedures to manipulate the data.
- **Polymorphism:** A programming language's ability to process objects differently depending on their data type or class.
- **Procedure:** a section of a program that performs a specific task.

Compiler, interpreter and assembler

Complier	**Interpreter**	**Assembler**
Complier converts high level language into machine language	Complier converts high level language into machine language	Assembler converts assemble language into machine language
It translates the entire program in one time	It executes one statement at a time	It translates the one instruction to one instruction
Execution of code is fast	Execution of code is slow	Execution of code is direct
Generates intermediate object code.	No intermediate object code is generated.	Generates object code.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met	Its Debugging is very difficult
Compiler is more intelligent than assembler	interpreter is less intelligent than compiler	Assembler is less intelligent than compiler
It requires more memory space	It requires less memory space	It depends the processor instructions
Examples: c,c++,c#, COBOL	**Ex:python,VB, perl,ruby,php**	**Ex: gas, gnu**

Compiler:

A compiler is a computer program that transforms source code written in programming language into machine language and then executes it. Normally, Compilers can take time because they have to translate high-level code to lower-level machine language at once and then save the executable object code to the memory. Also in a compiler, after conversion of all the code at once an error report for the whole program is generated.

Compilers can be classified based on what function it performs or how it has been constructed. Compilers can be classified as **single-pass, load-and-go, debugging, multi-pass and optimization.**

Compilation is performed in the following phases**: lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer, symbol table and error handle**.

Details About Compiler

1. A compiler converts high-level language program code into machine language and then executes it. High-level languages are C and C#
2. Complier scans the entire program first before translating into machine code.
3. Compiler takes entire program as input.
4. Intermediate object code is generated in case of compiler.
5. Compiler takes less execution time when compared to interpreter.
6. Examples include C, COBOL, C#, C++, etc
7. Compiler requires more memory than interpreter.
8. If you happen to make any modification in program you have to recompile entire program i.e scan the whole program every time after modification.
9. Compiler is faster when compared to interpreter.
10. There is usually no need to compile program every time (if not modified) at execution time.
11. Compiler gives you the list of all errors after compilation of whole program.
12. Compiler converts the entire program to machine code when all errors are removed execution takes place.
13. Compiler is slow for debugging because errors are displayed after entire program has been checked.

14. The assembly code generated by the compiler is a mnemonic version of machine code.

Interpreter:

Interpreter is a computer program that translates high level instructions into an intermediate form and then converts that intermediate code into machine language and; carries out specific actions. Interpreters are often used in software development tools as debugging tools because they can execute a single code at a time. In Java language, compiler and interpreter work together to generate machine code.

Details About Interpreter

1. Interpreter converts source code into the intermediate form and then converts that intermediate code into machine language. The intermediate code looks same as assembler code.
2. Interpreter scans and translates the program line by line to equivalent machine code.
3. Interpreter takes single instruction as input.
4. In case of interpreter, No intermediate object code is generated.
5. Interpreter takes more execution time when compared to compiler.
6. Examples include Python, Perl, VB, PostScript, LISP etc.
7. Interpreter needs less memory when compared to compiler.
8. If you make any modification and if that line has not been scanned then no need to recompile entire program.
9. Interpreter is slower when compared to compiler.
10. Every time program is scanned and translated at execution time.
11. Interpreter stops the translation at the error generation and will continue when error get solved.
12. Each time the program is executed; every line is checked for error and then converted into equivalent machine code.
13. Interpreter is good for fast debugging. An interpreter continues translating the program until the first error is arrived at, in which case it stops.
14. At the output of assembler is re-locatable machine code generated by an assembler represented by binary code.

Assembler:

An assembler is a program that takes basic computer instruction or instructions and then converts them into a pattern of bits that the computer processor can use to perform its basic operations. Usually, language used to program the assembler is referred to as **assembly language**. Assembler converts source code to an object code first then it converts the object code to machine language with the help of the linker programs.

1. Assembler converts source code written in assembly language into machine code and then that machine code is executed by a computer.
2. Assembler converts assembly language to machine language at once.
3. It converts a source code to an object first then it converts the object code to machine language with the linker programs.
4. Input to the assembler is assembly language code.
5. GAS, GNU is an example of an assembler.
6. At the output of assembler is re-locatable machine code generated by an assembler is represented by binary code.
7. Assembler cannot convert the whole code into machine language at once.
8. Assembler is less intelligent than compiler.
9. Assembler makes two phases over the given input first phase and the second phase.
10. It is difficult to debugging.