Structured programming

In structured programming design, programs are broken into different functions these functions are also known as **modules**, **subprogram**, **subroutines** and **procedures**

Each function is design to do a specific task with its own data and logic. Information can be passed from one function to another function through parameters. A function can have local data that cannot be accessed outside the function's scope. The result of this process is that all the other different functions are synthesized in an another function. This function is known as main function. Many of the high level languages supported structure programming.

Functional abstraction was introduced with structured programming.

Abstraction simply means that how able one can or we can say that it means the ability to look at something without knowing about its inner details.

In structured programming, it is important to know that a given function satisfies its requirement and performs a specific task. Weather How that task is performed is not important

Advantages of structured programming

1. It is user friendly and easy to understand.
2. Similar to English vocabulary of words and symbols.
3. It is easier to learn.
4. They require less time to write.
5. They are easier to maintain.
6. These are mainly problem oriented rather than machine based.
7. Program written in a higher level language can be translated into many machine languages and therefore can run on any computer for which there exists an appropriate translator.
8. It is independent of machine on which it is used i.e. programs developed in high level languages can be run on any computer.

Algorithm

An algorithm is a set of self contained sequence of instructions or actions that contains finite space or sequence and that will give us a result to a specific problem in a finite amount of time.

It is a logical and mathematical approach to solve or crack a problem using any possible method.

Types of algorithm

1. Recursive algorithms
2. Dynamic programming algorithm
3. Backtracking algorithm
4. Divide and conquer algorithm
5. Greedy algorithm
6. Brute Force algorithm
7. Randomized algorithm

1) Simple recursive algorithm

Solves the base case directly and then recurs with a simpler or easier input every time (A base value is set at the starting for which the algorithm terminates).

It is use to solve the problems which can be broken into simpler or smaller problems of same type.

Example:

To find factorial using recursion, here is the pseudo code:

```
 Fact(x)

 { If x is 0      /*0 is the base value and x is 0 is base case*/

         return 1

     return (x*Fact(x-1))  /* breaks the problem into small problems*/}
```

2) Dynamic programming algorithm

A dynamic programming algorithm (also known as dynamic optimization algorithm) remembers the past result and uses them to find new result means it solve complex problems by breaking it down into a collection of simpler subproblems, then solving each of those sub problems only once ,and storing their solution for future use instead of recomputing their solutions again.

Example:
Fibonacci sequence, here is the pseudo code :

```
Fib(n)
   if n=0
             return 0
          else
             prev_Fib=0,curr_Fib=1
          repeat n-1 times  /*if n=0 it will skip*/
             next_Fib=prev_Fib+curr_Fib
             prev_Fib=curr_Fib
             curr_Fib=new_Fib
          return curr_Fib
```

3) Backtracking algorithm

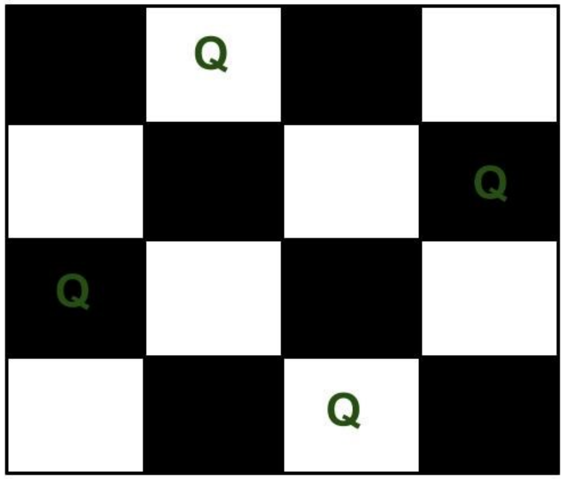
How about we learn backtracking using an example so let’s say we have a problem “Monk” and we divide it into four smaller problems “M, R, A, A”. It may be the case that the solution of these problems did not get accepted as the solution of “Monk”.

In fact we did not know on which one it depends. So we will check each one of them one by one until we find the solution for “Monk”.

So basically we attempt solving a subproblem but if we did not reach the desired solution undo whatever you have done and start from the scratch again until you find the solution.

Example:
Queens Problem, Depth First Search in Tree

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



4) Divide and conquer algorithm

Divide and conquer consist of two parts first of all it divides the problems into smaller subproblems of the same type and solve them solve them recusively and then combine them to to form the solution of the original problem.

Example:

Quick sort, Merge sort

5) Greedy algorithm

Greedy algorithm is an algorithm that solves the problem by taking optimal solution at the local level (without regards for any consequences) with the hope of finding optimal solution at the global level.

Greedy algorithm is used to find the optimal solution but it is not necessary that you will definitely find the optimal solution by following this algorithm.

Like there are some problems for which an optimal solution does not exist (currently) these are called NP complete problem.

Example:

Huffman tree, Counting money

6) Brute force algorithm
A brute force algorithm simply tries all the possibilities until a satisfactory solution is found.
Such types of algorithm are also used to find the optimal (best) solution as it checks all the possible solutions.
And also used for finding a satisfactory solution (not the best), simply stop as soon as a solution of the problem is found.
Example:
Exact string matching algorithm

7) Randomized algorithm

A randomized algorithm uses a random number at least once during the computation to make a decision.

Example:

Quick sort

As we use random number to choose the pivot point.

What is an algorithm?

An algorithm is an effective, efficient and best method which can be used to express solution of any problem within a finite amount of space and time and in a well-defined formal language. Starting from an initial state the instructions describe a process or computational process that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output“ and terminating at a final ending state.

In other words, we can say that,

- Step by step procedure for solving any problem is known as algorithm.
- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
- An algorithm is a sequence of computational steps that transform the input into a valuable or required output.
- Any special method of solving a certain kind of problem is known as algorithm.

All Algorithms must satisfy the following criteria -
1.Input: There are more quantities that are extremely supplied.

2.Output: At least one quantity is produced.

3.Definiteness: Each instruction of the algorithm should be clear and unambiguous.

4. Finiteness: The process should be terminated after a finite number of steps.

5.Effectiveness: Every instruction must be basic enough to be carried out theoretically or by using paper and pencil.

Properties of Algorithm

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm.

- **Non Ambiguity**
- **Range of Input**
- **Multiplicity**
- **Speed**
- **Finiteness**

Algorithm complexity

1. Performance required i.e., time complexity.
2. Memory requirement i.e., space complexity.
3. Programming requirements.

1. Space Complexity
The complexity of an algorithm, i.e., a program is the amount of memory; it needs to run to completion

Instruction space: Space needed to store the executable version of the program and it is fixed.

Data space: Space needed to store all constants; variable values and has further following components:

- Space needed by constants and simple variables. This space is fixed.
- Space needed by fixed size structured variable, such as array and structure.
- Dynamically allocated space. This space usually varies.

2. Time Complexity

The time complexity of an algorithm is the amount of time it needs to run a completion. In computer programming the time complexity any program or any code quantifies the amount of time taken by a program to run. The time complexity is define using some of notations like Big O notations, which excludes coefficients and lower order terms.

- The time complexity is said to be described asymptotically, when we describe it in this way i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most 5n3 + 3n for any n (bigger than some n0), the asymptotic time complexity is O(n3).
- It is commonly calculated by calculating the number of instructions executed by the program or the algorithm, where an elementary operation takes a fixed amount of time to perform

Algorithm complexity

There are basically two aspects of computer programming. One is the data organization i.e. the data and structure to represent the data of the problem in hand, and is the subject of present text. The other one involves choosing the appropriate algorithm to solve the problem in hand. Data structure and algorithm designing, both involved with each other.

As an algorithm is a sequence of steps to solve a problem, there may be more than one algorithm to solve a problem.

The choice of particular algorithm depends upon the following considerations:

1. Performance required i.e., time complexity.
2. Memory requirement i.e., space complexity.
3. Programming requirements.

Asymptotic Notations

Asymptotic Notations are languages that allow us to analyze an algorithm’s running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm’s growth rate.

- Does the algorithm suddenly become incredibly slow when the input size grows?
- Does it mostly maintain its quick run time as the input size increases?
- Asymptotic Notation gives us the ability to answer these questions.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

Best Case − Minimum time required for program execution.

Average Case − Average time required for program execution.

Worst Case − Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

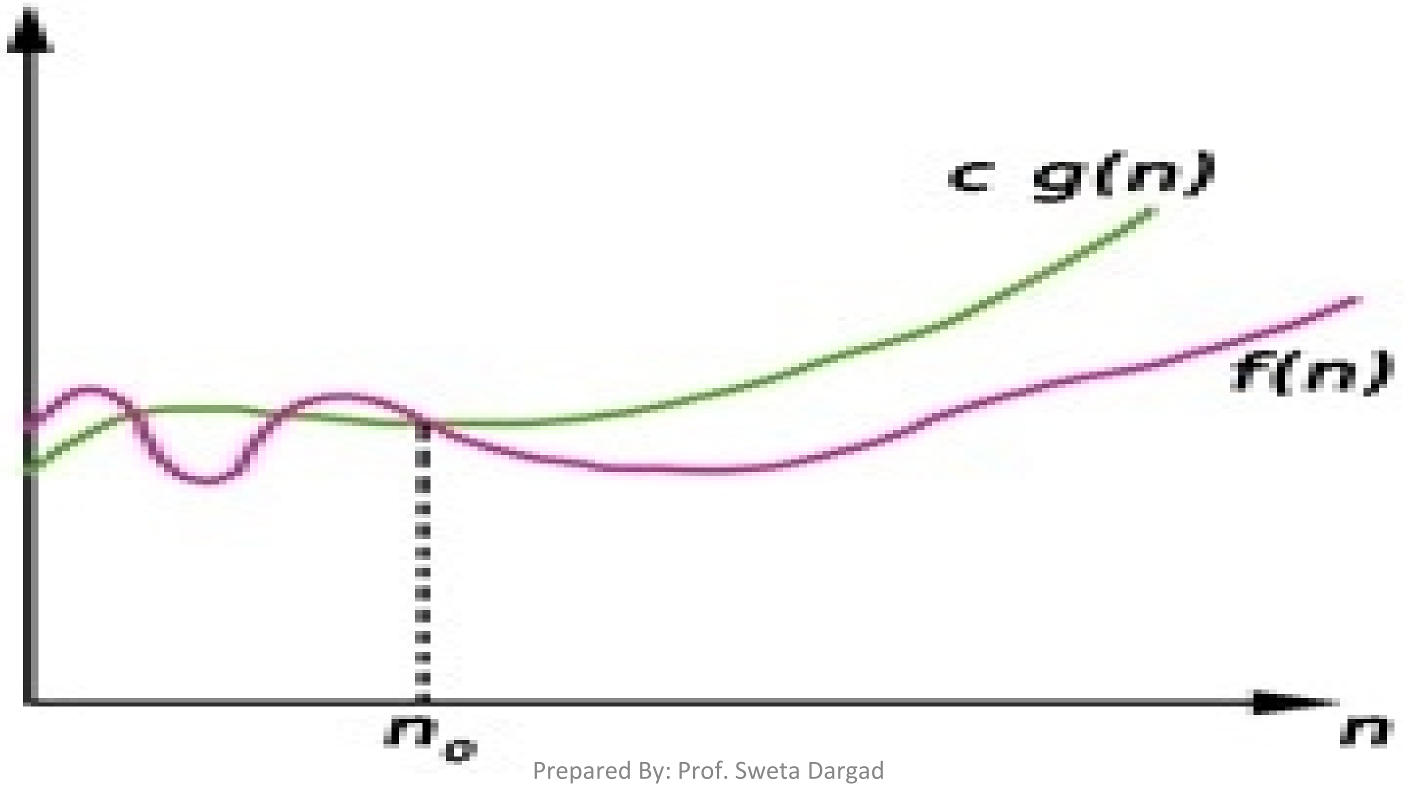
Ο Notation
Ω Notation
θ Notation

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

We write f(n) = O(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that
$0 \le f(n) \le c*g(n)$, for all $n \ge n0$}

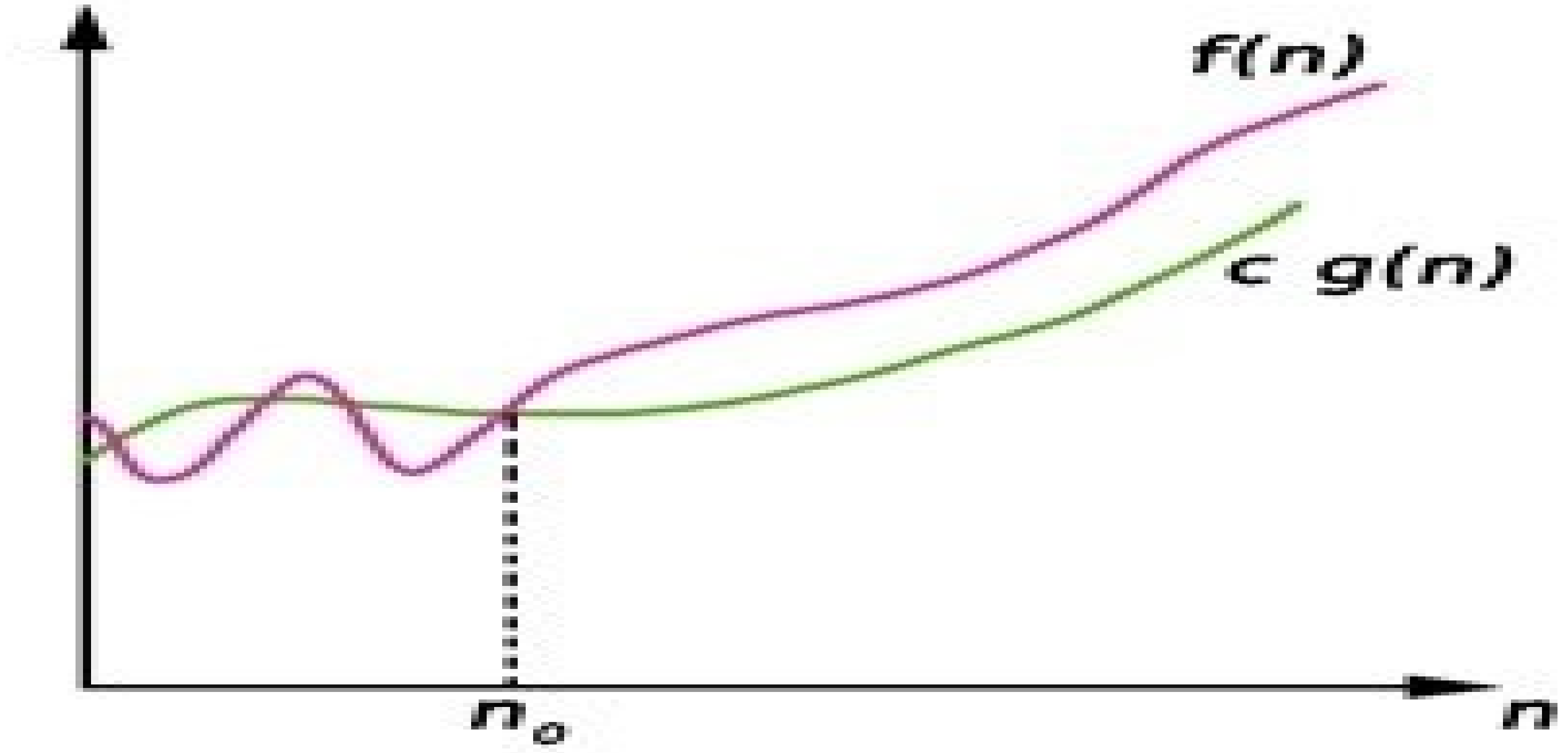


Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that
0 ≤ c*g(n) ≤ f(n), for all n ≥ n0}

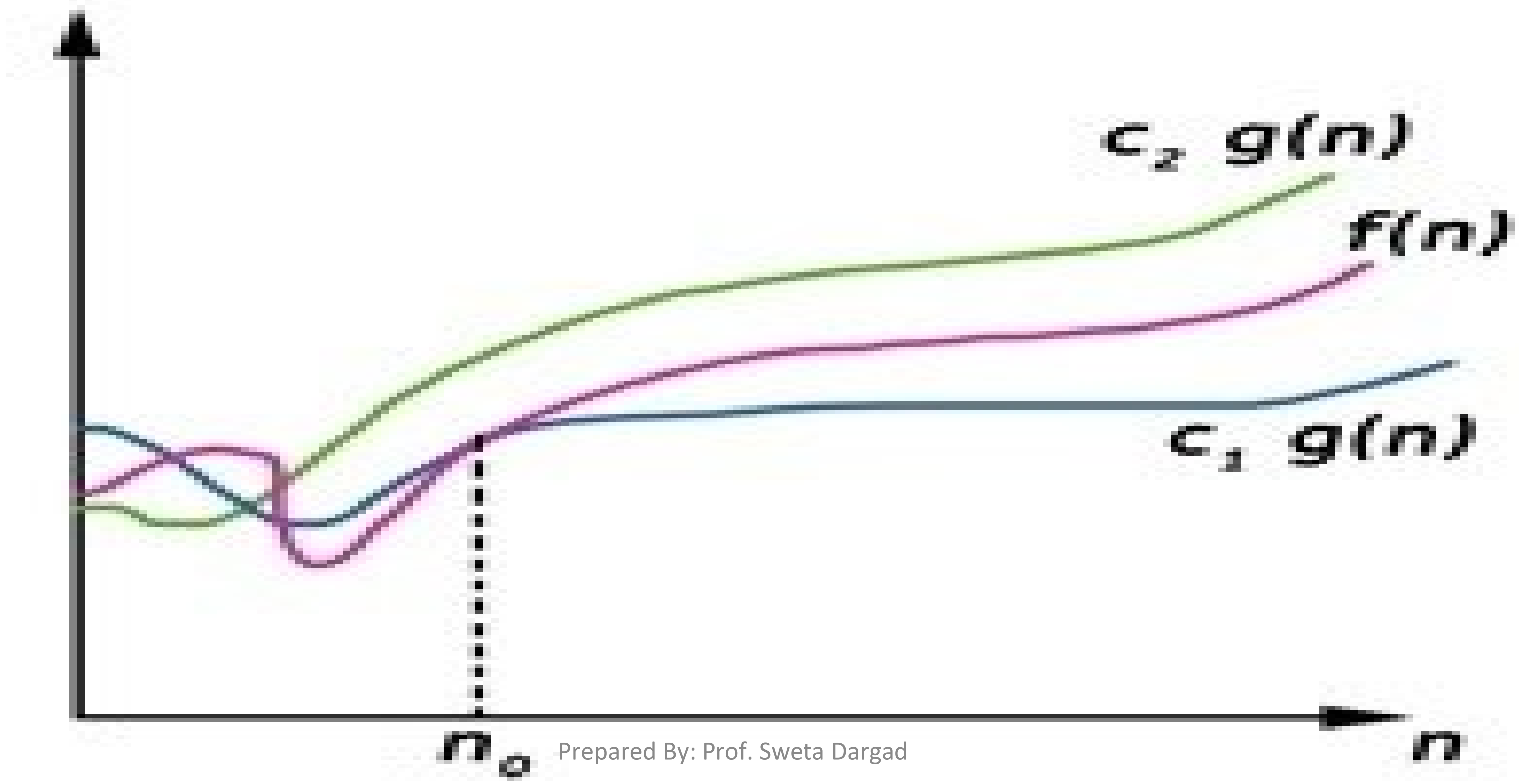


Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.

We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that
$0 \le c1*g(n) \le f(n) \le c2*g(n)$, for all $n \ge n0$}



Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	−	$O(1)$
logarithmic	−	$O(\log n)$
linear	−	$O(n)$
n log n	−	$O(n \log n)$
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

Algorithm

- An algorithm is a sequence of finite number of steps to solve a specific problem.

- A algorithm can be defined as well –defined step by step computational procedure which takes set of values as input and produce set of values as output.

- A finite set of instruction that followed to accomplish a particulartask.

Algorithm Terminology : format conventions

1. Name of algorithm:

   every algorithm is given an identifying name. the name written in capital letter.

2. Introductory comments:

   the algorithms name is followed by a brief introduction regarding the task the algorithm will perform. in addition it also discuss some assumptions.

For example: SUM(A,N): this algorithm find the sum

of all N elements in vector A. N is integer type variable indicate there are N elements in Vector A.

Algorithm Terminology : format conventions

3. Algorithm steps:

- Actually algorithm is a sequence of numbered steps.
- Each step is begin with a phrase enclosed in square brackets which gives the short description about the step.
- The phrase is followed by a set of statements which describe action to be performed in next line.

For example

```
2. [initialize variable I with 0]

        I <- 0
```

Algorithm Terminology : format conventions

4. Comments

- comments are optional part.
- an algorithm step may terminate with comments enclosed in rounding brackets just like we put comments in c program.
- Comment does not mean any action .

2. [initialize variable I with 0]

I <- 0

if S[I] = NULL //This statement checks the end of the String

Algorithm Terminology : format conventions

5. Assignment Statement ($\longleftarrow$)

- Assignment operation is indicated by arrow 🡐
- This operator assign the value of variable, constant, or expression of it right hand side to the left hand side variable.

For example:
X🡐 10
X 🡐 x+10

•Exchange of values between two variable is denoted by double sided arrow (<-->)

X $\longleftrightarrow$ Y

Algorithm Terminology : format conventions

- Conditional Statement : IF

we can use if statement in order execute some part of an algorithm based on condition.

Syntax is:

```
IF Condition
Then
    Statement 1
    Statement 2
     -----------
```

Note: -

We can not use { } to represent the body of if statement.

Algorithm Terminology : format conventions

- Conditional Statement : IF -- else

Syntax is:

```
        IF Condition
        Then
            Statement 1
            Statement 2
             -----------
       Else
            Statement 1
            Statement 2
             -----------
```

We can also form the else if ladder

Algorithm Terminology : format conventions

- Repeating statements:-- looping

Three form of looping structure:

1. Repeat for index variable= sequence of values(,)
2. Repeat while <logical expression>
3. Repeat for index= sequence of values while logical exp.

1.

Repeat for I = 1,2,3,4,5,6…….10.

// set of statements

This form is used when a steps are to be repeated for a counted number of times.

Algorithm Terminology : format conventions

```
Repeat for  I = 1,2,3,4,5,6…….10.
                          // set of statements
```

Here I is loop control variable.

Loop control variable take all values sequentially one by one.

Once all the set of Statements are in the range are executed ,the loop control variable assume the next value in a given sequence and again execute the same set of statements.

No need to write I<- I+1 statement .

Algorithm Terminology :format conventions

the loop can be extended over more then one steps in algorithm.like

```
1. [            ]
   //set of statements
2. [            ]
   Repeat thru step 4 for I= 1,2,3,..5
   //set of statements
3. [            ]
   //set of statements
4. [            ]
   //set of statements
5. [            ]
   //set of statements
```

Algorithm Terminology : format conventions

```
2.    Repeat while logical expression
              // set of statements
Repeat while X<10
        write(X)
        X <- X+1
```

Example:

```
  1. [     Phrase]
        //  set of statements
  2.  [    Phrase]
      Repeat while X<10
        write(X)
        X <- X+1
 3. [     Phrase]
              //set of statements
```

Algorithm Terminology : format conventions

```
2.   Repeat thru step N while logical expression
          // set of statements
Repeat thru step 3 while X<10
        write(X)
        X <- X+1
Example:
  1. [    Phrase]
        // set of statements
  2. [    Phrase]
       Repeat thru step 3 while X<10
        write(X)
        X <- X+1
  3. [    Phrase]
            //set of statements
```

Algorithm Terminology : format conventions

3. Repeat for index= sequence of values while logical exp.

```
            // set of statements
```

Or Repeat thru step N for index= sequence of values while logical exp.

```
          // set of statements
```

```
Repeat thru step 3 for I =1,2,3 while X<-10
      write(X)
      X <- X+1
```

Algorithm Terminology : format conventions

Go to Statement:

Unconditional transfer of execution control to step referenced.

Syntax

```
    Goto step N
```