Data Structures:

Data structures are used to organize and store data in a computer's memory.

- Provide a way to efficiently ***manage and manipulate data, allowing for easy access, insertion, deletion, and modification*** of data elements.
- Data structures are an essential part of ***computer science and software development*** and are typically implemented as classes or structures in programming languages.

Data Structure is a way of organizing and storing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.
For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type. We can organize this data as a record like Player record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33
In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

Data may be organized in many different ways. The logical or mathematical model of particular organization of data is called "Data Structure".
Or

The organized collection of data is called a 'Data Structure'.

Data Structure=Organized data +Allowed operations

Data Structure involves two complementary goals: The first goal is to identify and develop useful, mathematical entities and operations and to determine what class of problems can be solved by using these entities and operations. The second goal is to determine representation for those abstract entities to implement abstract operations on this concrete representation.

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

Data structure refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.
A data structure should be seen as a logical concept that must address two fundamental concerns.

1. First, how the data will be stored, and

2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

Basic types of Data Structures:

Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

• Linked List

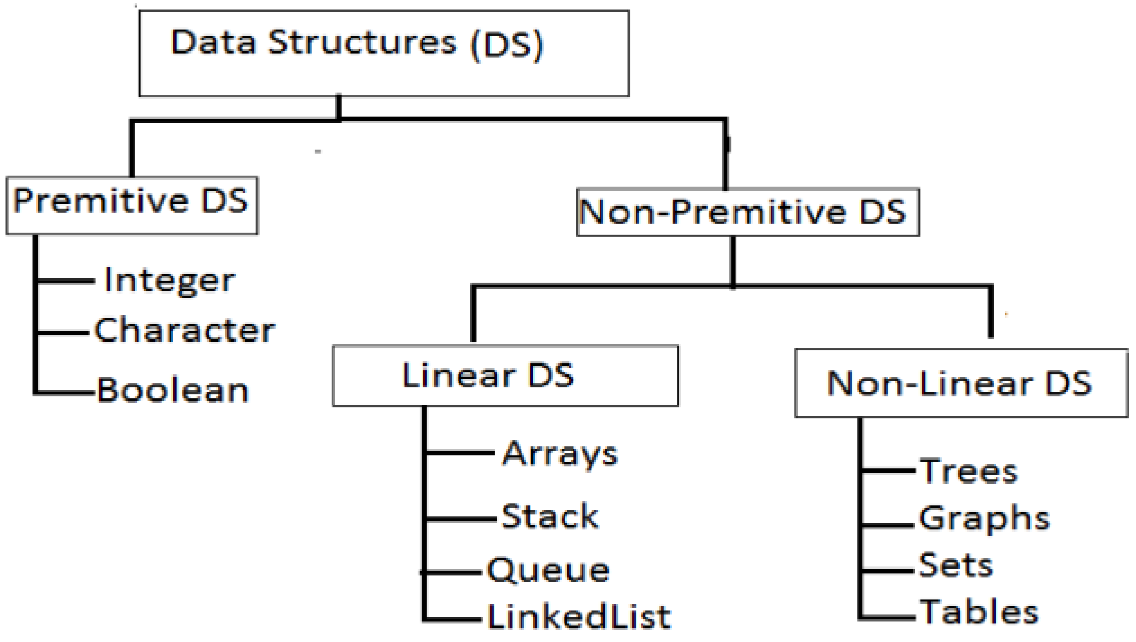
• Tree

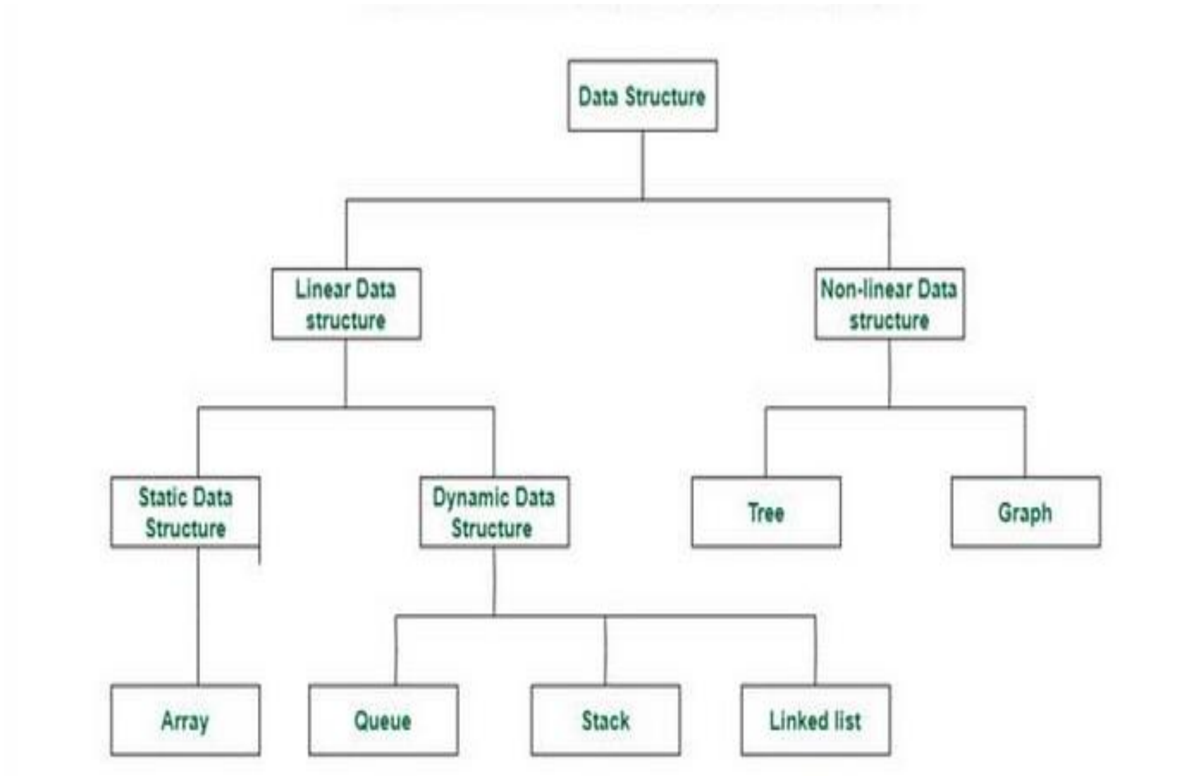
• Graph

• Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.

<u>Classification of Data Structures</u>:





Data structures are normally classified into two types:

They are **primitive data structures** and **non-primitive data structures**.

1. Primitive Data Structures:

Primitive data structures refer to the ***basic data types provided by programming languages***. These data types are usually *built-in and do not require any special implementation*. i.e.; any operation is directly performed in these data items. . These ***primitive data types*** are typically used to store only single value and perform basic arithmetic and logical operations. They are the foundation of data manipulation. The primitive data structures (also known as primitive data types) include int, char, float, double, boolean and pointers.

Some common primitive data types include:

1. ***Integer:*** Represents whole numbers, both positive and negative, without fractional parts. Examples include int, short, long, byte.
2. ***Floating-Point****:* Represents real numbers with fractional parts. Examples include float, double.
3. ***Character****:* Represents individual characters in a character set. Examples include char.
4. ***Boolean:*** Represents logical values, either true or false.

2. Non-Primitive Data Structures:

The non-primitive data structures are also known as the derived data structures as they are derived from primitive ones. Non-primitive data types are not defined by the programming language, but are instead created by the programmer. Non-primitive data structures are more complex and are composed of multiple primitive or non-primitive data elements. They are designed to organize and store collections of data in a structured manner.

A large number of values can be stored using the non-primitive data structures. The data stored can also be manipulated using various operations like insertion, deletion, searching, sorting, etc.

Some common Non-Primitive data structures include: arrays, Linked lists, structures, classes, stacks, Queues, Trees, Graphs, Structures, sets, Unions, Tables etc.

These non-primitive data structures provide more flexibility and functionality compared to primitive data types and are widely used to solve ***complex problems and handle larger data collections.***

The non-primitive data structures are further be divided into two categories:

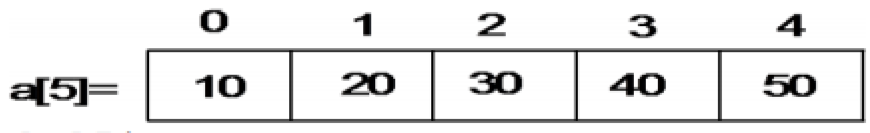
Linear data structures and Non Linear data structures.

(a) Linear data structures:

- In Linear data structures, the data elements are connected in a sequence manner.
- Line Linear data structures organize their data elements in a sequential or linear fashion, where data elements are attached one after the other.
- The memory location of each element stored can be accessed sequentially. The elements may not be present adjacently in the memory; however, each element is attached to the next element in some way.
- Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.
- Examples are Arrays, Linked Lists, Stacks, Queues, etc.

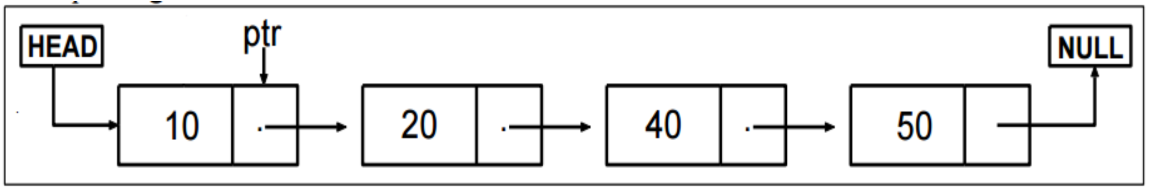
Array:

- Array is a Linear and fixed size data structure.
- It is the collection of homogeneous data elements (same type data elements),
- In array elements are referenced by a set of n consecutive numbers or scripts, usually 1, 2, 3, …………., n.
- It occupies the block of contiguous memory locations in memory of computer.
- The elements of array are stored in successive (contiguous) memory locations.



Linked List:

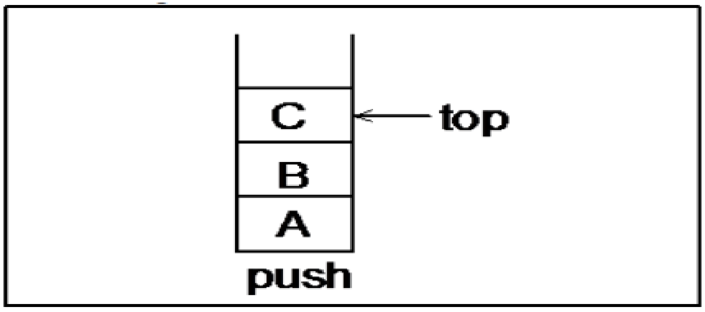
- Linked list or single linked list is a linear data structure.
- It is a sequence of elements in which every element has link to its next element in the sequence by means of pointers.
- Every element in linked list is called as a "node". Every "node" contains two fields, data field and link field. The data field holds the actual value or string and link field holds the address of next node.
- The starting node is called HEAD or List pointer variable, which is an empty node, contains an address of the first node of the list. It indicates the location where list starts from. So it link to the first node.
- The first node link to the second node and so on.
- The last node does not link to address but link to NULL, which indicates the end of list.
- Let ptr be a pointer to the linked list. The example is given below:



<u>Stack:</u>

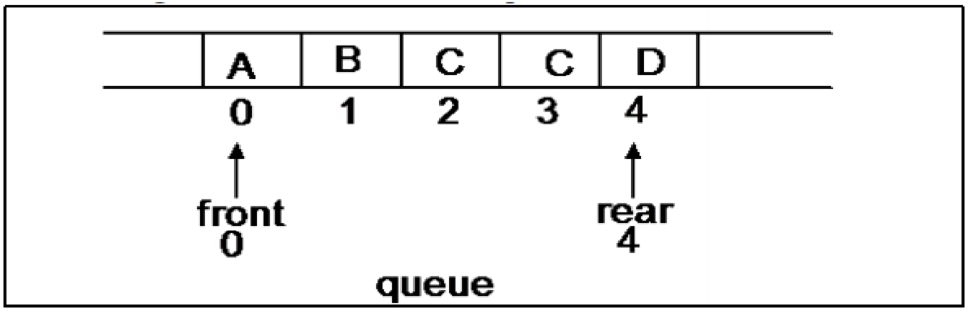
- ❖ A stack is a linear data structure.
- ❖ In stack insertion (addition) and deletion of data elements can take place only at one end, which is called **top** of the stack.
- ❖ It is also known as LIFO (Last In First Out) system, This means that the last item to be added to stack, is the first item to be removed.
- ❖ In stack items are removed in the reverse order as they were inserted, so last item inserted, will be removed first.
- ❖ Examples: 1. Stack of books. 2. Stack of CDs 3. Stack of plates
- ❖ We can perform two basic operations on stack.

1. Inserting (Adding) elements into the stack known as **push**.
2. Deleting elements from the stack known as **pop**



<u>Queue:</u>

- ❖ A queue is a linear data structure.
- ❖ In queue insertion (addition) of data elements can take place at one end (called the rear) and deletion can take place at the other end (called the front).
- ❖ It is also known as FIFO (First In First Out) System. This means that the elements are removed from the queue in the same order as they were inserted. , so first element to be inserted, will be removed first.
- ❖ Examples:
    1. Line of people waiting in NADRA office, for getting token.
    2. Line of people waiting in Bank for submitting utility bills.
- ❖ We can perform two basic operations on queue:
    1. Adding elements into the queue known as insertion at **rear.**
    2. Deleting elements from the queue known as deletion from **front.**

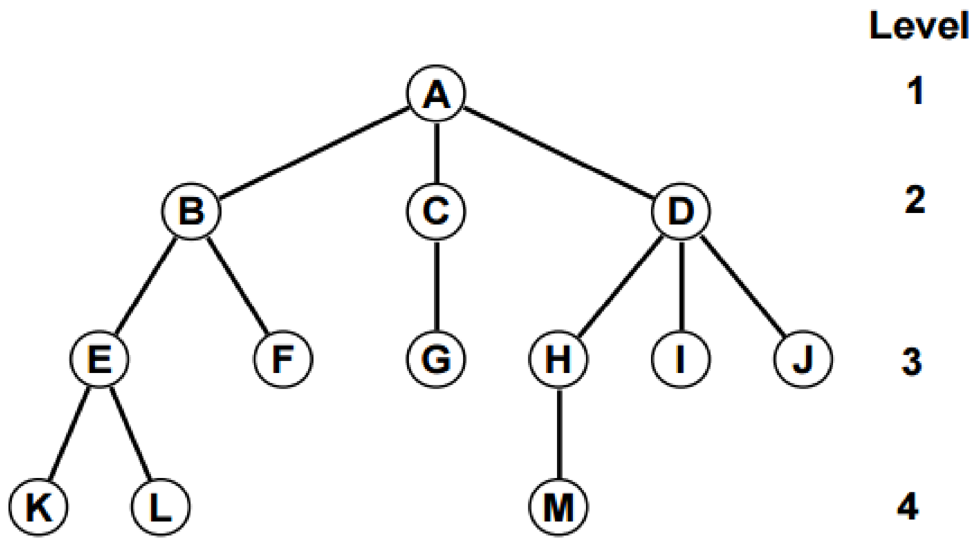


(b) Non-linear data structures:

- In Non-Linear data elements are not connected in a sequence manner, but in hierarchical manner.
- Non-linear data structures organize their data elements in a non-sequential manner.
- The data elements are not stored in a sequential fashion, but stored in multiple levels.
- , in which all elements, except one element, have unique parent (predecessor) but may have zero or many children (successors). The unique parentless element The implementation of non-linear data structures is more complex than linear data structures.
- Examples are Trees and Graphs etc.

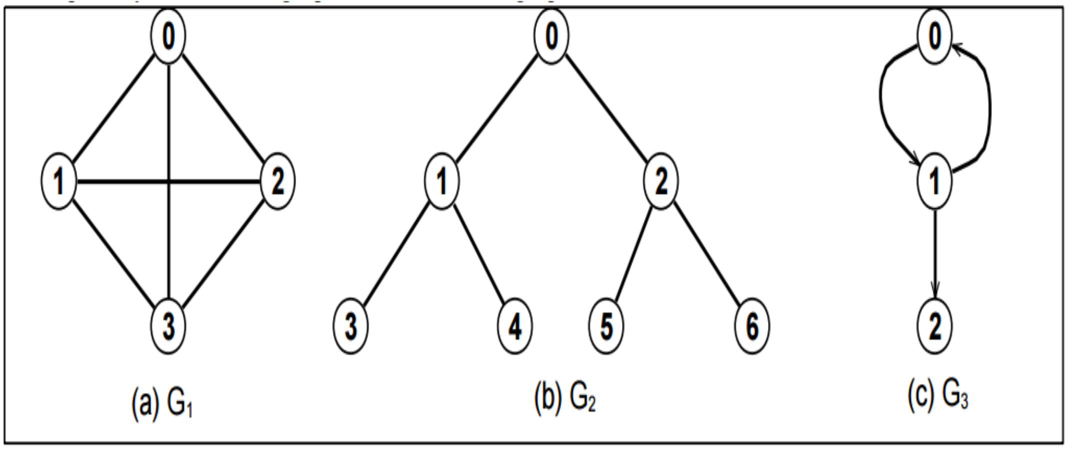
Tree:

- Tree is a non-linear data structure that represents a hierarchical relationship among the data elements is called root of the tree.
- The elements of tree are called nodes.
- Every node has unique path containing it to root of the tree.
- Terminal node that has no any child is called Leaf node.
- The tree is defined as a finite set of one or more nodes such that
  1. One node is called a root node and
  2. Remaining nodes partitioned into sub trees of the root.
- **Examples**: Family tree, Organization Chart, Table of contents, Genealogies,
- Computer file structure on right considering as tree (exploring)
- In this example, the root named Desktop, it has eight children including one named mail.



<u>Graph</u>:

- ❖ Graph is non-linear data structure, in which relationship among the data elements is not necessarily hierarchical in nature.
- ❖ Graph is mathematical model that is used to represent arbitrary relationship among the data elements.
  We often need to represent such arbitrary relationship among the data objects while dealing with problems in computer science, engineering, and many other disciplines.
- ❖ Examples: Flow chart of program, Route map of cities, electronic circuits etc.
- ❖ Data elements or nodes in graph are often called vertices. (Sometimes vertices are referred to as nodes or points)
- ❖ A graph is a pictorial representation of a set of points or nodes termed as vertices, and the links that connect the vertices are called edges.
- ❖ A Graph (G) consists of two sets V and E where V is called vertices and E is called edges. We also write
- ❖ G = (V, E) to represent a graph.
- ❖ A Graph may be directed graph and undirected graph.



- ❖ The Fig(a), Fig(b) are called undirected graph & Fig(c) is called directed graph.

Differences between Linear and Non Linear Data Structures:

Linear Data Structure	**Non-Linear Data Structure**
1. Linear data structures organize data elements in sequential manner, i.e., one by one attached to each other. Where data elements are stored one after other.	1. Non-Linear data structures do not organize data elements in sequential manner but in hierarchical manner. Where data is stored in multiple levels.
2. Examples: Array, Linked List Stack, Queue,	2. Examples : Tree, Graph
3. In linear data structure, every data element is attached to its previous & next one.	3. In Non-Linear data structures every data element is attached to many other data elements in specific way to represent relationship.
4. In Linear data structures data can be traversed in a single run.	4. In Non-Linear data cannot be traversed in a single run.
5. Implementation of Linear data structures is very easy, since memory of computer is also organized in a linear fashion.	5. Implementation of Non-Linear data structures is more difficult than Linear data structures.
6. To implement Linear data structures we do not need Non-Linear data structures, but we use Arrays & Linked Lists.	6. To implement Non-Linear data structures, we need Linear data structures, we use Linked list only.
7. Using Linear data structures, application software is created.	7. Using Non-Linear data structures , advanced concepts are implemented, viz: Game theory, AI, Image scanning.

Classification of Data Structures:

The data structures can also be classified on the basis of the following characteristics:

Characterstic	Description
Linear:	In Linear data structures,the data items are arranged in a linear sequence. Example: Array
Non-Linear:	In Non-Linear data structures,the data items are not in sequence. Example: Tree, Graph
Homogeneous :	In homogeneous data structures,all the elements are of same type. Example: Array.
Non-Homogeneous:	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: Structures
Static :	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: Array
Dynamic :	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations are changed. Example: Linked List created using pointers

On the basis of size, the data structures in C can also be classified as:

- **Static Data Structures:** (static means fixed/not flexible)
  - ➢ The size of Static data structures is fixed and predefined, as the memory is allocated to them during the compile time. However, the values of the elements stored are not static and can be modified at any time.
  - ➢ Memory is contiguous (Sequential memory locations) and reserved at the compile time. Once it is created, it cannot be modified.
  - ➢ Due to fixed maximum size, it cannot grow or shrink (its size cannot be increased or decreased) during execution time (run time).
    - Fast access to data elements.
    - Expansive to insert/delete elements.
    - Example: Array.
    - When declaring an array data structure, its size has to be specified.
    - Such as **int a[50],** here **int** is type data in array, a is name of array, 50 is a having size of **50** is number of elements/size of array

- **Dynamic Data Structures**: (Dynamic means flexible)
  - ➢ The size of dynamic data structures is flexible, can be modified at run time. The required amount of memory space is allocated to them according to number of data elements during run time.
  - ➢ The new memory space is allocated to them when it is needed for storing new data element and unused memory space is deallocated when any element is removed from the data structure.
  - ➢ The memory space allocated to such data structures can be modified (increased or decreased), so its size can grow and shrink as needed to contain data we want to store it.
  - ➢ Example: Linked list.
  - ➢ Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations are changed.
    - Fast insertion/deletion of elements.
    - Slow access to elements.
    - Pointers and links can be used to build these data structures.

Differences between Static & Dynamic memory allocation:

Static Memory Allocation	Dynamic Memory Allocation
1. Static memory allocation is a concept in which memory is allocated will in advance.	1. Dynamic memory allocation is a concept in which memory is allocated during run time as and when required.
2. We can use only that amount of memory for our programming purpose and need.	2. We can allocate memory as per requirement during run time.
3. Static memory allocation is done during writing the program.	3. Dynamic memory allocation is done during running the program.
4. In case of static memory allocation, we cannot increase /decrease the size of memory allocation during run time.	4. In case of dynamic memory allocation, we can increase as well as decrease allocated memory.
5. In case of static memory allocation, there are chances of wastage storage space.	5. In case of dynamic memory allocation, there is no chance of memory loss.
6. Example: Array	6. Example: Linked list

Operations on the Data Structures:

The data which is stored in our data structures are processed by some set of operations.

The different operations can be performed on the data structures are:

1. Traversing 2. Searching 3. Inserting 4. Deleting
5. Sorting 6. Merging 7. Splitting

1. **Traversing**: (Travel through the data structures) It is used to access or visit each and every data element of the list in systematic manner.(one by one from first to last).
2. **Searching**- It is used to find out the required data item and its location if it exists in the given collection of data items with the given key value (primary key) , which satisfies one or more conditions.
3. **Inserting**- It is used to add a new data item in the given collection of data items.
4. **Deleting**- It is used to remove an existing data item from the given collection of data items.
5. **Sorting**- It is used to arrange the data items in particular order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. **Merging**- It is used to combine the data items of two sorted files into a single sorted list.
7. **Splitting**: It is the process of partitioning single list to multiple lists.

What is an Algorithm ?

An algorithm is a step-by-step procedure or set of rules designed to perform a specific task or solve a particular problem. It is a **finite sequence of well-defined, unambiguous instructions** that a computer or human can follow to accomplish a specific computational or mathematical task.

Algorithms serve as the foundation for computer programs and are crucial in ***various fields, including computer science, mathematics, and engineering***. They are used to ***automate processes, manipulate data, and solve problems efficiently***.

Algorithms can range from simple, like ***sorting*** a list of numbers, to complex, such as those used in ***artificial intelligence, data analysis, and cryptography.***

Algorithms can be found in various aspects of everyday life, not just in computer science. For example, a recipe for baking a cake can be seen as an algorithm, with each step specifying a precise action to be taken.

In computer science, algorithms serve as the ***building blocks*** for software development, data analysis, artificial intelligence, and more.
An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as pseudocode or using a flowchart.
An algorithm is a Step by Step process or sequence of steps to solve a problem, where each step indicates an intermediate task.
Algorithms are problem solving procedures that are used to manipulate the data in these data structures. Algorithms manipulate the data in these data structures in various ways, such as inserting a new data item, searching for a particular item or sorting the data items etc.
We can think of an algorithm as a step by step recipe, a list of detailed instructions for carrying out an activity.
Example: Addition, conversion from decimal to binary, the process of boiling an egg, the process of mailing a letter, sorting, searching etc.

Better definition: **An algorithm is a precise sequence of a limited number of unambiguous, executable steps that must terminate in the form of solution.**
In this definition, three requirements are:

1. Sequence is (i) precise (ii) consists of limited number of steps.
2. Each step is (i) unambiguous (clearly defined) (ii) executable.
3. The sequence of steps must terminate in the form of solution.

- When an algorithm gets coded in a specified programming language such as C, C++, or Java, it becomes a program that can be executed on a computer.
- Multiple algorithms can exist to solve the same problem or complete the same task.
- ***Algorithms*** can be represented using various forms such as ***natural language, pseudocode, flowcharts, or programming languages***. The choice of representation depends on the context and the intended audience.
- The word algorithm is derived from well-known 9th century mathematician of Iran, Abu-Jaffar Ibn Moosa Al-Khwarizmi, who developed methods to solve problem using specific instructions step by step. Originally the term was algorism but later on it changed to algorithm.

Properties /Characteristics of an Algorithm:-

1. Input: An algorithm takes one or more inputs, which are the data or information that the algorithm operates on.

2. Output: An algorithm produces an output, which is the result or solution obtained after executing the steps of the algorithm.

3. Definiteness: Algorithms have well-defined instructions that are clear, precise, and unambiguous. Each step must be executable and leave no room for interpretation.

4. Finiteness: Algorithms have a finite number of steps. They must eventually terminate after a finite number of operations.

5. Effectiveness: Algorithms are effective in solving a problem or accomplishing a task. They provide a correct solution or output for all valid inputs within a reasonable amount of time. Also Each and every step in an algorithm can be converted into programming language statements easily

6. Determinism: Algorithms are deterministic, meaning that for a given input, the same set of instructions will always produce the same output.

Major Goals of an Algorithm:

1. Correctness: The algorithm should produce the correct output for all possible valid inputs. It should accurately solve the problem it was designed for.

2. Clarity: The algorithm should be easy to understand. It should have unambiguous instructions, making it accessible to both humans and computers.

3. Efficiency: An efficient algorithm uses the least amount of resources, such as ***time and memory***, to accomplish its task. It should be designed to run in a reasonable amount of time and not waste unnecessary resources.

4. Generality: The algorithm should apply to a range of inputs and should not be overly specific. A general-purpose algorithm is versatile and can be adapted to various situations.

5. Robustness: An algorithm should be able to handle different types of inputs, including unexpected or erroneous data, without crashing or producing incorrect results.

6. Optimality: In some cases, achieving the best possible efficiency in terms of time or space complexity is a goal. An optimal algorithm performs its task using the minimum necessary resources.

7. Scalability: The algorithm should be designed to handle larger inputs or datasets without a significant decrease in performance. It should scale well to accommodate growing or changing requirements.

Algorithm notation/ Structure of Algorithm:

Usually format of algorithm consists of two parts:

1. Paragraph: It describes the purpose of algorithm, identifies the variables which occur in algorithm and lists the input data.
2. List of steps that are to be executed.

To write an effective algorithm, we have to follow some algorithm notations, certain conventions that we will use in presenting algorithm:

1. Identifying name/number of algorithm: Each algorithm must be given a unique name for identification before writing it. For example: Algorithm 2.3 refers to 3rd algorithm in chapter 2
2. Step number: Each statement of algorithm must be given unique number for identification. For example: 1,2,3,4 etc. We may use word Step to before starting new step.
3. Comments/Remarks: We can use non-executable statements, lines, comments or remarks in our algorithm inside large brackets [&]. Also /* & */ and // are used for this purpose.
4. Variable name: Variable is programmer defined identifier (name), whose value can be changed during the execution of program. Alphanumeric characters are used for variable names as X, y, NUM, sum. Single letter variable used as counters , subscripts or index numbers, C is case sensitive language . Reserved words /Key words are not used as variable names.
5. Assigning value to variable: **:=** is used as assignment statement. Some texts use ← or = sign for this operation. Word "Set " may be used to assign value to variable. Computer itself sets the value for variable.
6. Getting Input from user: The words "Read" or "Input" can be used to get input from user. Computer asks the user for value. scanf(), cin() , gets(), getch etc are used in C language.
7. Providing Output to the user: The word "write" , "print" or "output" can be used to provide output to the user. printf(), cout(), puts etc are used in C language.
8. Control transfer/Control structures: We can use control structures to control the logic flow and execution in our algorithm. Use "**Go to step n**" to transfer control to step n. Selection logics (If—Then, If—then-- Else if, Switch—case) and iteration logics (for , While and Do-While loops) are also used for this purpose.
9. Sub algorithms (procedures/functions) : Sub algorithm is like a sub program or function in main program. It is independent algorithm module, which solve the particular problem, and called by main algorithm.
10. Algorithm termination: The words "Exit" or "Stop" are used to terminate the algorithm.

<u>Categories of Algorithms:</u>

Based on the different types of steps in Algorithms, they can be divided into three categories, namely

- Sequential Algorithms
- Conditional Algorithms
- Iterative Algorithms

i. <u>Sequential Alorithms:</u>

The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

Example 1:

```
// adding two numbers
Step 1: start
Step 2: read a,b
Step 3: Sum=a+b
Step 4: write Sum
Step 5: stop
```

Example 2: Write an algorithm to find the Simple Interest for given Time and Rate of Interest.

```
// Algorithm to find simple interest
Step 1: Start
Step 2: Read P,R,S,T.
Step 3: Calculate S=(PTR)/100
Step 4: Print S
Step 5: Stop
```

ii. <u>Conditional Algorithms</u>:

The sequence types of algorithms are not sufficient to solve the problems, which involves decision and conditions. In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm.

- Here we use if condition and the condition is checked only one time.
- If a condition is satisfied then next statement is executed otherwise else statement is executed.

The general format of Selection type of statement is as shown below:

```
if(condition)
Statement-1;
else
Statement-2;
```

The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed. In case the operation is unsuccessful. Then sequence of algorithm should be changed/ corrected in such a way that the system will re execute until the operation is successful.

Example 1:

```
// identifying the given number is Odd or Even
Step 1 :  start
Step 2 :  read n
Step 3 :   if (n%2==0) then
                print “Odd no”
           else
                print “Even no”
Step 4 :  stop
```

Example 2:

```
// biggest among two numbers:
Step 1:    start
Step 2:     read  a,b
Step 3:     if  a>b   then
                Write “a is greater than b”
            Else
                Write “ b is greater than a”
Step  4:     Stop
```

Example 3: . Write an algorithm to find the largest among three different numbers entered by user

```
// biggest among three numbers
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If   a>b  And   If a>c
              Display  a  is the largest number.
             else If    b>a  And  b>c
              Display b is the largest number.
              else
              Display c is the greatest number.
Step 5: Stop
```

Example 4 : Write an algorithm for roots of a Quadratic Equation?

```
// Roots of a quadratic Equation
Step 1 : start
Step 2 : read a,b,c
Step 3 : if (a= 0) then step 4 else step 5
Step 4 : Write “ Given equation is a linear equation “
Step 5 : d=(b * b) _ (4 *a *c)
Step 6 : if ( d>0) then step 7 else step8
Step 7 : Write “ Roots are real and Distinct”
Step 8: if(d=0) then step 9 else step 10
Step 9: Write “Roots are real and equal”
Step 10: Write “ Roots are Imaginary”
Step 11: stop.
```

iii. Iterative Algorithms:

Iteration type algorithms are used in solving the problems which involves repetition of statement. In this type of algorithms, a particular number of statements are repeated 'n' no. of times.

Here we use loops such as: **while**, **do-while** and **for** & the condition is checked number of times.

i.e. The statements in an iteration block are executed no. of times based on some condition.

Example 1A:

```
 // algorithm to print 1 to N numbers using while loop.
 Step1:  Declare variable N,  I
Step2:  Read N
 Step3:  Initialize I=1
 Step4 : Repeat While(I<=N)
              Write I,
               Set I=I+1
          [End of While loop]
  Step 7 : Exit.
```

Example 1B:

```
// algorithm to print 1 to N numbers using Do/While loop.
 Step1:   Declare variable N, I
 Step2:             Read N
 Step3:   Initialize I=1
 Step4 :   Do
              Write I,
               Set I=I+1
             While (I<=N)
          [End of Do/While loop]
 Step 7 : Exit.
```

Example 1C:

```
 // algorithm to print 1 to N numbers using For  loop.
 Step1:  Declare variable N,  I
Step2:  Read N
Step3 : Repeat  For  I:= 1 to N by 1
              Write I,
          [End of For  loop]
  Step4 : Exit.
```

<u>Example2</u>:

```
// calculate the SUM of digits of a given number
Step 1 :  start
Step2:  Initialize sum=0,
Step3 :  read n,
Step 4 : repeat step 4 while  n>0
Step 5 : (a) r=n mod 10              //  r is remainder
         (b) sum=sum+r
          (c) n=n/10
Step 6: write sum
Step 7 : stop
```

<u>Example 3:</u> .Write an algorithm to find the factorial of a number entered by user.

```
Step 1: Start
Step 2: Declare variables n,  factorial and i.
Step 3: Initialize variables:    factorial←1,  i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
          5.1: factorial←factorial*i
          5.2: i←i+1
Step 6: Display factorial
Step 7: Stop
```

Some examples of algorithm :

Examples1: Write an algorithm that gets three integers from user, then calculates and displays the average of three numbers.

```
Algorithm:  [Average of three numbers]
Step1:  Declare variables A,B,C, AVG
Step2:  Read  A,B,C
Step3:  Set AVG := (A+B+C)/3
Step4:   Write AVG
Step5:  Exit
```

Example2: Write an algorithm that receives two values for two different variables, interchange (swap) their values then displays them.

```
Algorithm:      [swap values of variables]
    Step1:      Declare variables A,B, TEMP
    Step2:      Read A,B
    Step3:      Write  "A= ", A   & "B=",B
    Step4:      TEMP := A
    Step5:      A := B
    Step6:      B :=TEMP
    Step7:      Write  "A= ", A   & "B=",B
    Step8:      Exit
```

Home Work:

1. Write an algorithm that calculates the area of circle by getting the value of radius from user. (Area of circle = $A = \pi r^2$)
2. Write an algorithm to input the temperature in Fahrenheit and convert it into Centigrade and print it. ($C = \frac{5}{9}(F - 32)$)

Example3:

```
Algorithm:      [To identify odd or even number]
Step1:          Declare variable N
Step2:          Read N
Step3:          If (N mod 2 == 0) then
                Write "The given number is Even"
                Else
                Write "The given number is Odd"
                [End of If/Else structure]
Step4:          Exit
```

Example4:

```
Algorithm:  [Largest among two numbers]
Step 1:    Declare  variables A,B
Step 2:     Read A,B
Step 3:     If  A>B  then
                 Write "A  is greater than B"
              Else
                  Write " B is greater than A"
             [End of  If/Else  structure]
Step 4:    Exit
```

Example 5:

```
Algorithm:  [Largest among three numbers]
Step 1: Declare variables A,B and C.
Step 2: Read variables a,b and c.
Step 3: If    ( A>B    AND    A>C) then
                        Write " A is the largest number".
            Else if  (B>A    AND    B>C)
                        Write " B is the largest number"
            Else
                        Write " C  is the largest number".
            [End of If/Else if structure]
Step4:  Exit.
```

Example 6:

```
Algorithm: [Factorial of  the given number]
Step1:  Declare integer variables N,I, FACT
Step2:  Set  FACT := 1
Step3:  Read N
Step4:  Repeat For  I := N to 1 by -1
                        FACT := FACT * I
            [End of  For loop]
Step5:  Write" Factorial =" , FACT
Step6:  Exit
```

Example7: Write an algorithm to calculate the value of number raised to power of another (y^x).

```
Algorithm: [Value of number raised to power of another number of number ($y^x$)]
Step1:  Declare integer variable N, POW, I, VALUE
Step2: SET  VALUE := 1
Step3:  Read  N, POW
Step4:  Repeat For  I := 1 to POW by 1
                VALUE := VALUE *N
        [End of For loop]
Step5: Write "Value =", VALUE
Step6:  Exit
```

Example8: Write an algorithm that identify the given integer number is prime or not.

```
Algorithm: [To identify Prime Number]
Step1:  Declare integer variables N, I, IS_PRIME
Step2:  Read N
Step3:  If (N= =1) then
        Write "Given integer number is neither prime nor composite"
        Exit
        [End of If structure]
Step4: Set IS_PRIME  := 1    [Set flag status ON means number is prime (yes)]
Step5:  Repeat For I :=2 to N-1 by 1
                If ( N mod I = = 0 ) then
                Set IS_PRIME := 0     [Set fag OFF  means not prime (no)]
                Break                 [ Exit from loop]
                [ End of If structure ]
        [End of For loop]
Step6:  If ( IS_PRIME  $\neq$ 0 )  then
        Write "The Given integer is Prime"
        Else
        Write "The Given integer is not prime"
        [End of  If/Else structure]
Step7:  Exit
```

Control Structures:

Control structures are used to control the flow of execution in algorithms.
Three types of control structures are used:

(1) Sequence Logic / Sequential flow/ Sequence structure:
(2) Selection logic/ Conditional flow/ Selection structure
(3) Iteration logic/ Repetitive flow/ Repetition structure

(1) **Sequence Logic:** In sequence logic the steps/ statements /modules are executed in the same order in which they are written within algorithm. Sequence logic Algorithm is executed from top to bottom. Every step is carried out only once, none is repeated, and none is omitted.

Structure of sequence logic:

```
Step 1: Module A
Step 2: Module B
Step 3: Module C
```

(2) Selection logic: (conditional flow)

The selection logic uses number of conditions which lead to select one out of several alternatives. The structures which implement this logic are called conditional structures or If structures.
The end of If structure is frequently indicated by the statement **[End of If structure]** or some equivalent statement.

There are three types of conditional structures are used in algorithms:

1. Single Alternative/ If(condition)
2. Double Alternative/ If—else
3. Multiple Alternative/if—else if—else / switch—case

1. Single Alternative/ If(condition):

In this structure if the condition is true, do something, but if it is not true then skip and do nothing and control transfer to next step of the program.
Example: If marks>60% then admit otherwise nothing.
If the condition is true, then execute Module A (which may consist of one or more statements) , otherwise Module A is skipped and control transfer to next step of the algorithm.
This structure has the form:

```
If (condition) then
    [Module A]
[End of If structure]
```

2. **Double Alternatives: If (condition) /else :**

This structure selects one action out of two different sets of actions. The first action or set of actions is performed when the condition is true and second set is performed if the condition is false.

```
If (a condition is true) then
 (do something)
Else
 (do different thing)
```

The structure has the form:

```
If (condition) then
    [ Module  A]
Else
    [Module B]
[End of if/else  structure ]
```

If the condition is true , then Module A is executed, otherwise Module B is executed. If/else perform an action if the condition is true and perform a different action if the condition is false.

3. **Multiple Alternative: If(condition)/elseif(condition/-----------/else. Switch/case**

This structure selects one action out of many actions.

This structure has the form:

```
If (condition-1) then
   [Module A]
Elseif (condition-2) then
   [Module B]
Elseif (condition-3) then
   [Module C]
Elseif [condition-4) then
   [Module D]
Else
   [Module E]
[End of If/elseif structure]
```

(3) Iteration logic (repetitive flow)

Iteration is often referred to as loop, because the algorithm will keep repeating the activity, until condition becomes false.

Loops simplify programming, because they enable the programmer to specify certain instructions only once to have the computer execute many times.

Three types of loops are used in algorithms: For loop , While loop, Do-While loop.

- Each type begins with a Repeat statement
- Followed by a module is placed, called body of loop.
- For the clarity, The end of the structure is frequently indicated by the statement, **"[End of loop]"**

1. **<u>Repeat For</u>**: uses an index variable, such as k, to control the loop. For loop is used for definite (limited) number of repetitions. It defines the number of repetitions that will occur during execution.
   The form of loop will be
   ```
   Repeat For K= R  to S    by    T
         [Module ]
   [end of For loop]
   ```
   Where **k** is index variable, **R** is initial value**, S** is final or test value and **T** is increment or decrement.

2. **<u>Repeat While loop</u>**: uses a condition to control the loop at beginning. In this logic the loop depends on the condition.
   The form of loop will be:
   ```
   Repeat While (condition)
        [Module]
   [end of While loop]
   ```
   The While and Do-While loops are used for indefinite repetition. That is, they do not define the number of repetitions that will occur when the loop is executed. They merely give conditions for looping to stop.

3. **<u>Do-While loo</u>**p: uses a condition to control the loop at the end of structure.
   The loop will usually have the form:
   ```
   Do
       [Module]
   Repeat While (condition)
   [End of While loop]
   ```

Sub-Algorithms:

Sub-Algorithm is a complete and independent algorithm module which is called by some main algorithm or any some other algorithm.

A sub algorithm contains parameters or arguments that receive values from calling algorithm (Originating algorithm) perform some computations and then send back result to the calling algorithm.

The sub-algorithm is defined independently, so that it may be called by many different algorithms or called at different times.in the same algorithm.

The relationship between an algorithm and a sub algorithm is similar to the relationship between a main program and sub-program in a programming language.

Difference between sub-algorithm and algorithm

1. The sub-algorithm will usually have a heading of the form:
   NAME (Par 1, Par 2, -------,Par k)
   Where **NAME** refers to the name of sub-algorithm, which is used when the sub-algorithm is called.
   And **Par 1, Par 2,-------, Par k** refer to the parameters , which are used to transmit data between the sub-algorithm & the main algorithm.
2. The sub-algorithm will have **Return** statement in the last rather than an **Exit** statement. This means that control is transferred back to the calling program, when the execution of the sub-algorithm is completed.

Types of sub-algorithms:

Sub-algorithms fall into two basic categories: Function sub algorithms and Procedure sub algorithms.

1. **Function sub-algorithms** : The Function sub-algorithm returns only a single value to the calling algorithm.

Example:

```
Algorithm:   [Addition of three numbers]
Step 1:  Declare variables A,B,C,SUM.
Step 2: Read A,B,C
Step 3: Set SUM := SUM(A,B,C)   [Calling sub-algorithm]
Step 4:   Write SUM
Step5:  Exit
```

```
Function: SUM(X,Y,Z)
Step1:  Declare variable D
Step 2: Set D :=X+Y+z
Step 3: Return (D)
```

2. **Procedure sub-algorithms:** The procedure sub-algorithm may send back more than one value.

Example:

```
Algorithm:  [swap values]
Step 1: Declare values A,B
Step 2:  Set  A:=10    &     B:=20
Step 3: Write A,B
Step 4:  Call SWAP(A,B)  [calling sub-algorithm]
Step 5: Write A, B
Step 6: Exit
Procedure:  SWAP(X,Y)
Step1 : Declare  variable T
Step 2: Set T:=X,
Step 3: Set  X := Y
Step 4:  Set  Y := T
Step 5: Return
```

Parameters: are variables, which are used to transfer data between a subprogram and its calling program

Each function consists of three components :

1. Function Declaration: (function prototype) declares function
   void line(void);   /* Function prototype , at the end semi colon must be used*/
2. Function Call : Executes function
   Line(); /* Function call , at the end semi colon must be used* /
3. Function definition: It is the function itself.
   void line(void)   /* no semicolon */

Local variables and Global variables:

Local or Automatic variables:

Variables defined within a block of a function are called local variables. Their scope is local to the block of a function only. These variables not assessable from outside the block and thus their visibility remain to the block.

They are declared inside a particular function and used only there, are called "local" or "automatic" variable. The local variable is only "visible" or "accessible" within a specific part of program, the sub algorithm or module in which they were declared.

Storage: memory

Default initial value: Garbage

Scope: Local to block in which the variable is defined.

Life: Till the control remains within the block in which the variable is defined:

Example:

```
#include <stdio.h>
void myFunction() {
   int localVar = 10;   // Local variable
   printf("Local variable: %d\n", localVar);
}

int main()
   myFunction();
    printf("%d", localVar); // This will cause an error since localVar is not accessible here
   return 0;
}
```

In this program variables a,b and c are local variables and cannot be used outside the main function

Global or External variable:

Variable defined at the top of a program before the main() function are called global variables. These variables are accessible by all the functions of the same program. In order to declare global variables we simply have to declare the variable outside any function or block; that means , directly in the body of the program.

Global variables are declared outside the main program/algorithm. It holds the value of variable during entire execution of the program. The value of global variable can be shared with different functions. The global variable can be accessed by all the program modules in computer program.

Each programming language has its own syntax for declaring such variables

These variables are visible to all functions that care to use them.

Storage: memory

Default initial value: 0

Scope: Global

Life: As long as program‘s execution does not come to end.

Example:

```
#include <stdio.h>
Int  globalVar = 20;  // Global variable

void myFunction()
{
   printf("Global variable: %d\n", globalVar);
}

int main()
{
   myFunction();
   printf("Global variable: %d\n", globalVar); // Accessing global variable directly
   globalVar = 30; // Modifying global variable
   printf("Modified global variable: %d\n", globalVar);
   return 0;
}
```

In this example, the variables a, b, c are defined globally and have therefore been accessed in t main() function and addition() function.

Local/automatic variables Page: 142
Global/External variables Page: 159
Turbo C   by Robrt Lafore

Examples of Algorithms with implementation in C-language.

Example 1 : ***adding two numbers***

// Algorithm for adding two numbers

```
Step 1: Start
Step 2: Declare variables
Step 3: Read the numbers
Step 4: Add the numbers
Step 5: Output the sum
Step 6: Stop
```

<u>OR</u>

Algorithm: [adding two numbers]

```
Step 1: start
Step2: Declare integer variables  a,b, sum
Step 3: read a,b
Step 4: Sum=a+b
Step 5: write Sum
Step 6: stop
```

```c
#include <stdio.h>

    // Step 1: Start
    int main() {

    // Step 2: Declare integer vriables
     int  a, b, sum;

    // Step 3: Read the numbers
    printf("Enter the first number:  ";)
    scanf("%d”,&a);
    printf("Enter the second number:  ");
    scanf("%d”,&b);

    // Step 4: Add the numbers
     sum = a + b;

    // Step 5: Output the sum
    printf("The sum is:  %d\n”, sum;

    // Step 6: Stop
    return 0;
    }
```

Example 2: ***Calculating the factorial of a number.***

// ***calculating the factorial of a number.***

```
Step 1: Start
Step 2: Initialize variables
Step 3: Calculate factorial
Step 4: Return the factorial
Step 5: Stop
```

```
// Function or method definition
int factorial(int n) {
    // Step 1: Start
    // Step 2: Initialize variables
    int result = 1;

    // Step 3: Calculate factorial
    for (int i = 1; i <= n; i++) {
        result *= i;
    }

    // Step 4: Return the factorial
    return result;
    // Step 5: Stop
}
```

Example 3: **Grade Classification**:

An algorithm that classifies student grades based on a score can use conditional statements to determine the grade category (e.g., “Excellent,” “Good,” “Average,” “Fail”) based on the score range.

Step 1: Start

Step 2: Classify the grade based on the score

Step 3: If score is >= 90, print “Excellent”

Step 4: If score is >= 80, print “Good”

Step 5: If score is >= 70, print “Average”

Step 6: If score is < 70, print “Fail”

Step 7: Stop

```
// Function or method definition

void classifyGrade(int score) {

// Step 1: Start

// Step 2: Classify the grade based on the score

    if (score >= 90) {

   // Step 3: If score is >= 90, print "Excellent"

        print("Excellent");

    } else if (score >= 80) {

    // Step 4: If score is >= 80, print "Good"

        print("Good");

    } else if (score >= 70) {

    // Step 5: If score is >= 70, print "Average"

        print("Average");

    } else {

     // Step 6: If score is < 70, print "Fail"

        print("Fail");

    }

    // Step 7: Stop

}
```

Example 4: calculates the sum of the digits of a given number.

Step 1: Start.

Step 2: Read the value of `n`.

Step 3: Repeat Step 4 until `n > 0`.

Step 4:

(a) Calculate the remainder `r` when `n` is divided by 10 (`r = n mod 10`).

(b) Add the value of `r` to the running sum `s` (`s = s + r`).

(c) Update the value of `n` by dividing it by 10 (`n = n / 10`).

Step 5: Write the value of `s`, which represents the sum of the digits of the original number.

Step 6: Stop.

```
// calculate the SUM of digits of a given number
Step 1 :  start
Step2: Declare  variables  n, r, s   &    Initialize  s=0,
Step3 :  read n,
Step 4 : repeat step 4 while  n>0
Step 5 : (a) r=n mod 10
         (b) s=s+r
          (c) n=n/10
Step 6: write s
Step 7 : stop
```

N.B:

- This algorithm repeatedly extracts the rightmost digit of the number `n`, adds it to the sum `s`, and then removes that digit by dividing `n` by 10.
- It continues this process until all the digits of `n` have been processed.
- Finally, it outputs the calculated sum `s`, which represents the sum of the digits of the original number.
- It's worth mentioning that the provided algorithm assumes that `s` is initialized to 0 before the iteration begins.

<u>Program in C-language</u>

```
#include <iostream>

    int main() {
        // Step 1: Start

        //  Step 2: Declare variables n, r , s  &  initialize s =0
            int n, r, s = 0;

    // Step 3: Read the value of n
    std::cout << "Enter a number: ";
    std::cin >> n;

    // Step 4: Repeat Step 4 until n > 0
    while (n > 0) {
    // Step 5(a): Calculate the remainder r when n is divided by 10
        r = n % 10;

   // Step 5(b): Add the value of r to the running sum s
        s += r;

   // Step 5(c): Update the value of n by dividing it by 10
        n /= 10;
    }

    // Step 5: Write the sum s
    std::cout << "Sum of digits: " << s << std::endl;

    // Step 6: Stop
    return 0;
}
```

➢ **<u>The appropriate algorithm can be determined based on number of factors</u>**:
1. How long the algorithm takes to run?
2. What resources are required to execute the algorithm?
3. How much space or memory is required?
4. How exact is the solution provided by the algorithm?

<u>Different Approaches to Design an Algorithm</u>:
An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.
Practical Algorithm Design Issues:
1. To save time (Time Complexity): A program that runs faster is a better program.

2. To save space (Space Complexity): A program that saves space over a competing program is considerable desirable.

<u>Complexity of Algorithms /Efficiency of Algorithms/Performance Analysis an Algorithm</u>:
An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance or efficiency of an algorithm is measured on the basis of following properties:
1. Time Complexity
2. Space Complexity
The performances of algorithms can be measured on the scales of time and space. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.
The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.
The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.
The time complexity of an algorithm can be computed either by an empirical or theoretical approach. The empirical or posteriori testing approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

The complexity of an algorithm is studied with respect to the following 3 cases:

(a)**Worst Case Analysis:**
In the worst case analysis, we calculate upper bound on running time of an algorithm.
We must know the case that causes maximum number of operations to be executed.
For Linear Search the worst case happens when the element to be searched is not present in the array.

(b)Average Case Analysis:
In the average case analysis, we calculate average on running time of an algorithm. We must know the average number of operations to be executed.
In the linear search problem, the average case occurs when x is present at average of its location.
(c) Best Case Analysis:

In the best case analysis, we calculate lower bound on running time of an algorithm.
We must know the case that causes minimum number of operations to be executed
In the linear search problem, the best case occurs when x is present at the first location.