UNIT 1

Syllabus: Introduction to Algorithm and Data Structures Algorithm: Introduction - Algorithm Design – Complexity- Asymptotic notations. Data Structures: Introduction- Classification of Data structure - Abstract Data Type (ADT).

Introduction Data Structures

Definition & Concept of Data Structure

The intimate relationship between data and programs can be traced to the beginning of computing. In any area of application, the input data, (internally stored data) and output data may each have a unique structure. **Data structure** is representation of the **logical relationship** existing between individual elements of data. In other words, a data structure is a **way of organizing all data items** that considers not only the elements stored but also their relationship to each other.

Data structure mainly specifies the following **four things** :

(i) Organization of data
(ii) Accessing methods
(iii) Degree of Associativity
(iv) Processing alternatives for information

Data structures are the **building blocks of a program,** and it **affects the design** of both **structural and functional** aspects of a program. And hence the selection of a particular data structure stresses on the following two things.

1. The **data structures must be rich enough** in structure to reflect the relationship existing between the data.
2. The **structure should be simple** so that we can process data effectively whenever required.

Classification of Data structure

Data structures are normally **divided into two broad** categories.

(i) Primitive data structures
(ii) Non-primitive data structures

Primitive Data Structure

These are **basic structures and are directly operated** upon by **the machine instructions.** In general, they have different representations on different computers. **Integer, floating point numbers, character constants, string constants, pointers** etc.

Non-Primitive Data Structures

These are **more sophisticated data structures**. These are **derived from the primitive data structures**. The non-primitive data structures emphasize on

structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. **Arrays, lists and files** are examples.

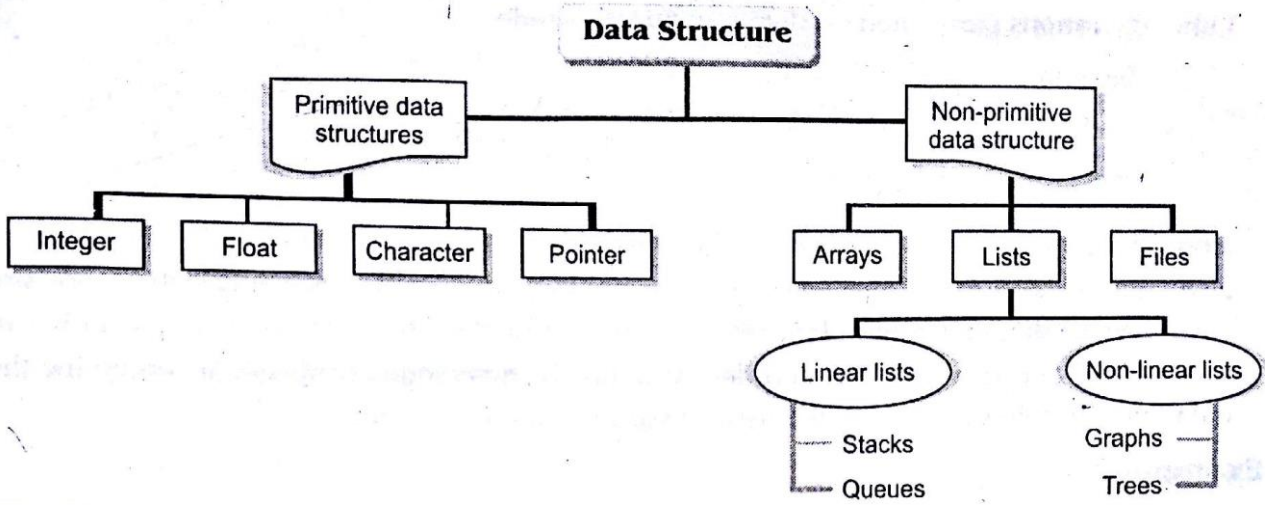


Figure 1. Data Structure classification

1. **Arrays -** An array most commonly used **non-primitive data** structures. It is defined as a **set of finite number of homogeneous elements** or data items. It means an array can contain **one type of data only**, either all integers, all floating-point numbers, or all characters. Declaration of arrays is as follows :

   int a[10] ;

   Where **int** specifies the data type or type of elements array stores. **"a"** is the name of array, and the **number specified inside the square brackets** is the number of **elements an array** can store this is also **called size or length of array**.

2. **Lists -** A list (Linear Linked list) can be defined as a **collection of variable number of data items.** Lists are the most commonly used **non-primitive data structures**. An **element of list** must contain least **two fields, one for storing data** or information and **other for storing address of next element**.
3.
4. For **storing address** we have a **special data structures called pointers,** hence the second field of the list must be pointer type. Technically each such element is referred to as a **node.**

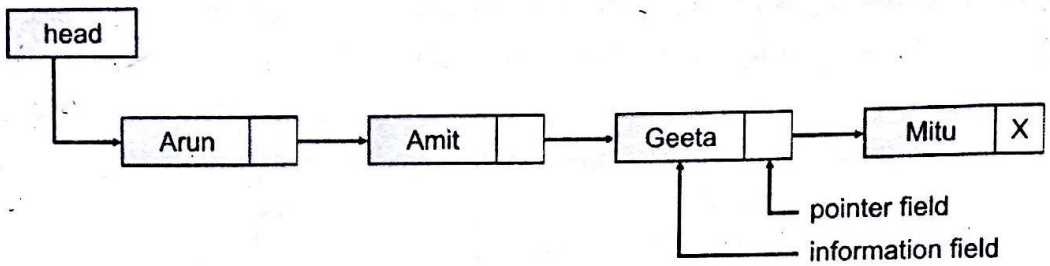


Figure 2. Linear Linked List

List is categories into **two types:**

i. Linear List or Linear data structures
ii. Non Linear list or Non Linear data structures

4.1 Linear data structures – Those data structure where the **data elements** are **organised in some sequence** is called Linear data structures. Here operation on data structure are possible in a sequence. **Stack, queue, array** are example of linear data structure.

4.1.1 Stack - A stack is commonly used **non-primitive linear data structures**. A stack is also an ordered collection of elements like arrays, but it has a **special feature** that **deletion and insertion** of elements can be **done only from one end, called the top of stack** (TOP). Due to this property it is also called as **last in first out** type of data structure (LIFO).

When an element is inserted into a stack or removed from the stack, its base remains fixed whereas the top of stack changes. **Insertion** of element into stack is **called Push and Deletion** of element from stack **is called Pop.** The figure 3 shows how the operations take place on a stack.

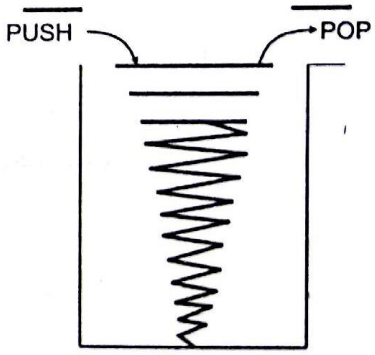


Figure 3. Operations on stack.

The stacks can be implemented in two ways:

(i) Using arrays (static implementation).
(ii) Using pointers (dynamic implementation)

4.1.2 Queues - Queues are **first in first out** type of Data Structures (i.e., FIFO). In a queue, **new elements are added** to the queue from **one end called REAR** end, and the **elements** are always **removed** from **other end** called the **FRONT end**.

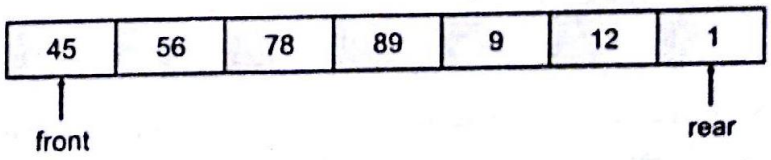


Figure 4. Operations on Queue.

The Queues can also be implemented in two ways:

(i) Using arrays (static implementation).
(ii) Using pointers (dynamic implementation)

4.2 Non Linear data structures – Those data structure where the **data elements** are **not organised in some sequence**, organised in some arbitrary function without any sequence is called Non linear data structures. **Graph, Tree are example** of linear data structure.

4.2.1 Trees - A Tree can be defined as **finite set of data items (nodes)**. Tree is **non primitive non-linear type of data structures** in which data items are arranged or stored in a sorted sequence. Trees represent the **hierarchical relationship** between various elements. In trees:

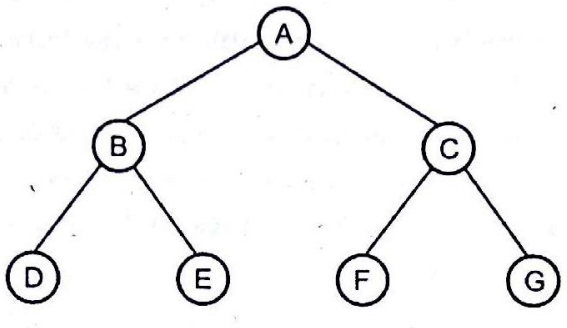


Figure 5 Binary. Tree

1. There is a **special data item** at the top of hierarchy **called the Root** of the tree.

2. The remaining data items are partitioned into number of mutually exclusive subsets, each of which is itself, a tree, which is **called the subtree**.
3. The tree always grows in length towards bottom in data structures, unlike natural trees which grow upwards.

The tree structure organizes the data into branches, which relate the information.

2.2.2 Graph - Graph is a mathematical **non primitive non-linear data structure** capable of representing many kinds of physical structures. A **graph G (V, E)** is a **set of vertices V** and **a set of edges E**. An **edge connects a pair of vertices** and many have weight such as length, cost or another measuring instrument for recording the graph. **Vertices** on the graph are shown as **point or circles** and edges are drawn as arcs or line segment. Thus an edge can be representing as E (V, W) where V and W are pair of vertices. The vertices V and W lie on E Vertices may be considered as cities and edges, arcs, line segment as roads.

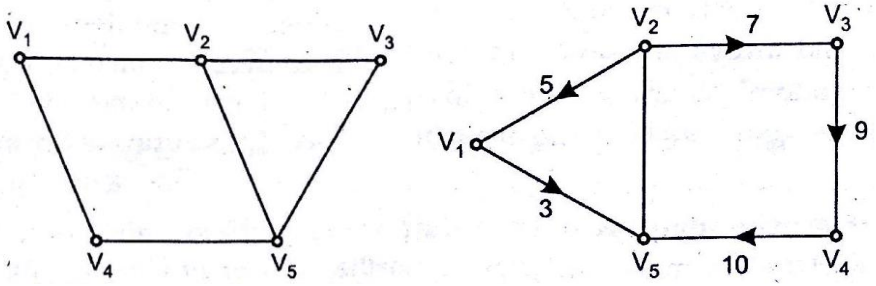


Figure 6 (a)Undirected graph (b) Directed and Weighted graph.

Types of Graph - The graphs are classified in the following categories: -

1. Simple Graph
2. Directed Graph
3. Non-directed Graph
4. Connected Graph
5. Non-connected Graph
6. Multi-Graph

The **most commonly used operations on Primitive and Non-primitive data structures** are broadly categorized into four types.

1. **CREATE -** This operation results in **reserving memory** for the program elements. This can be done by **declaration statement**. The creation of data structure may take place either **during compile time or during run time.**

2. **SELECTION -** This operation deals with **accessing a particular data** within a data structure.

3. **DESTROY or DELETE -** This operation **destroys the memory space** allocated for the specified data structure. **Malloc() and free()** function of C language are used for these two operations respectively.

4. **UPDATION -** As the name implies this operation **updates or modifies** the data in the data structure. Probably new data may be entered or previously stored data may be deleted.

Operation On Data Structure - Other operations performed on data structure includes -:

1. **SEARCHING –** Searching operation f**inds the presence** of the desired data item in the list of data item. It may also **find the locations** of all elements that satisfy certain conditions.

2. **SORTING –** Sorting is the process of **arranging all data items** in a data structure in a particular order say for example, either in ascending order or in descending order.

3. **MERGING -** Merging is a process of **combining the data items** of two different sorted lists into a single sorted list.

Abstract Data Type (ADT)

There are **different in-built data types** that are provided to us. Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a **situation when we need operations** for our **user-defined data type** which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we **can create data structures** along with **their operations**, and such data structures that are not in-built are **known as Abstract Data Type** (ADT).

Abstract data type model

Before knowing about the abstract data type model, we should know about **abstraction and encapsulation.**

1. **Abstraction:** It is a technique of **hiding the internal details** from the user and only showing the necessary details to the user.
2. **Encapsulation:** It is a technique of **combining the data** and the **member function in a single unit** is known as encapsulation.

Abstract Data type (ADT) is a **type or class for objects** whose behaviour is defined by a **set of values** and a **set of operations**. The definition of ADT only mentions **what operations** are to be performed but **not how** these operations will be implemented. It does **not specify how data** will be organized in memory and what algorithms will be used for implementing the operations. It is **called "abstract"** because it gives an **implementation-independent view**. The process of **providing** only the **essentials and hiding the details** is known as abstraction.

The below figure shows the ADT model. There are **two types of models** in the ADT model, i.e., the **public function** and the **private function**. The ADT model also contains the data structures that we are using in a program. In this model, **first encapsulation** is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the **abstraction is performed** means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

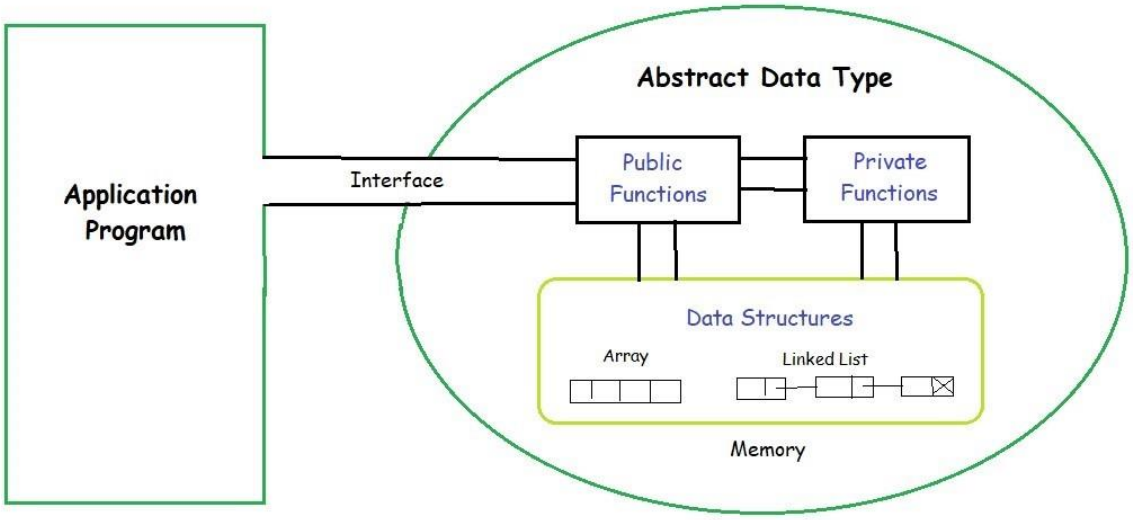
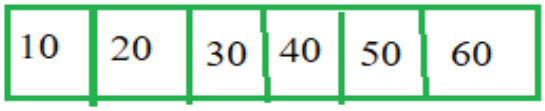


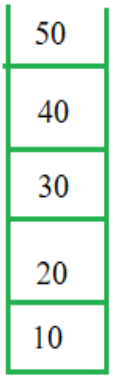
Figure 7. ADT

The **user of data type** does **not need to know** how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of **ADT as a black box** which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. List ADT - The data is generally stored in **key sequence** in a list which has a head structure consisting of ***count, pointers* and *address of compare function*** needed to compare the data in the list. The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list. The **List ADT Functions** is given below:



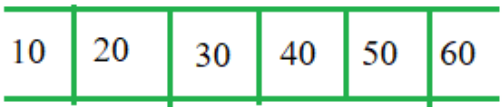
1. get() – Return an element from the list at any given position.
2. insert() – Insert an element at any position of the list.
3. remove() – Remove the first occurrence of any element from a non-empty list.
4. removeAt() – Remove the element at a specified location from a non-empty list.
5. replace() – Replace an element at any position by another element.
6. size() – Return the number of elements in the list.
7. isEmpty() – Return true if the list is empty, otherwise return false.
8. isFull() – Return true if the list is full, otherwise return false.



2. Stack ADT - In Stack ADT Implementation **instead of data** being stored in each node, the **pointer to data is stored**. The program allocates memory for the ***data* and *address*** is passed to the stack ADT. The **head node and the data nodes** are encapsulated in the ADT. The **calling function** can only see the **pointer to the stack**. The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

1. pop() – Remove and return the element at the top of the stack, if it is not empty.
2. push() – Insert an element at one end of the stack called top.
3. peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
4. size() – Return the number of elements in the stack.
5. isEmpty() – Return true if the stack is empty, otherwise return false.
6. isFull() – Return true if the stack is full, otherwise return false.

5. **Queue ADT -** The queue abstract data type (ADT) **follows** the basic **design of the stack abstract data type**. Each node contains a void pointer to the ***data*** **and the** ***link pointer*** to the next element in the queue. The program's responsibility is to allocate memory for storing the data.



1. enqueue() – Insert an element at the end of the queue.
2. dequeue() – Remove and return the first element of the queue, if the queue is not empty.
3. peek() – Return the element of the queue without removing it, if the queue is not empty.
4. size() – Return the number of elements in the queue.
5. isEmpty() – Return true if the queue is empty, otherwise return false.
6. isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

1. **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
2. **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
3. **Robust:** The program is robust and has the ability to catch errors.
4. **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
5. **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
6. **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
7. **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
8. **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner. Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

Advantages:

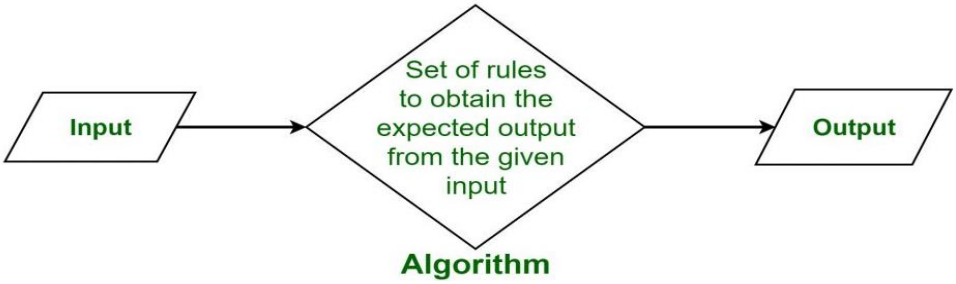
1. **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
2. **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
3. **Data Structure Independence**: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
4. **Information Hiding**: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
5. **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

1. **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
2. **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
3. **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
4. **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
5. **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Introduction to Algorithm

An algorithm named after ninth century is defined as - An **algorithm is a set of rules** for carrying out calculations either **by hand or on a machine.** It is a **sequence of computational steps** that transform the input into the output or a sequence of operations performed on data that have to be organized in data structures. We can also say that algorithm is an obstruction of a program to be executed on a physical machine.



Algorithm + Data structure = Program

Figure 8

An algorithm is a **step-by-step procedure** to solve a particular function. That is, it is a set of instructions written to carry out certain tasks and the data structure is the way of organizing the data with their logical relationship retained. To develop a program of an algorithm, we should select an **appropriate data structure** for that algorithm. Therefore algorithm and its associated data structures form a program.

Why do we need Algorithms? - We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Characteristics of an Algorithm - The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Example: Suppose we want to make a Orange juice, so following are the steps required to make a Orange juice:

Step 1: First, we will cut the Orange into half.
Step 2: Squeeze the Orange as much you can and take out its juice in a container.
Step 3: Add two tablespoon sugar in it.
Step 4: Stir the container until the sugar gets dissolved.
Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.
Step 7: Now, it's ready to drink.

Example: Write an algorithm to add two numbers entered by the user.
The following are the steps required to add two numbers entered by the user:

Step 1: Start
Step 2: Declare three variables a, b, and sum.
Step 3: Enter the values of a and b.
Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.
Step 5: Print sum
Step 6: Stop

Factors of an Algorithm - The following are the factors that we need to consider for designing an algorithm:

- **Modularity:** If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.
- **Correctness:** The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.
- **Maintainability:** Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.
- **Functionality:** It considers various logical steps to solve the real-world problem.
- **Robustness:** Robustness means that how an algorithm can clearly define our problem.
- **User-friendly:** If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.
- **Simplicity:** If the algorithm is simple then it is easy to understand.
- **Extensibility:** If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

Importance of Algorithms

1. **Theoretical importance:** When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.
2. **Practical importance:** As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

Issues of Algorithms - The following are the issues that come while designing an algorithm:

- **How to design algorithms:** As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.
- **How to analyze algorithm efficiency**

Algorithm Design

There are primarily main designing methods of algorithms are categories can be named in this type of classification. There are several types of algorithms available. Some important algorithms are:

1. **Brute Force Algorithm:** It is the simplest approach for a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. **Recursive Algorithm:** A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again. **Example:** The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.

3. **Backtracking Algorithm:** The backtracking algorithm basically builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution or all possible solutions are looked after. **Example:** N-queen problem, maize problem.

4. **Searching Algorithm:** Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. **Sorting Algorithm:** Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner. **Example:** For NP-Hard Problems,

6. **Hashing Algorithm:** Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. **Divide and Conquer Algorithm:** This algorithm breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together to get the final solution. It consists of the following three steps: 1. Divide, 2. Solve, 3. Combine **Example:** Merge sort, Quicksort.

8. **Greedy Algorithm:** In this type of algorithm the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part. The one solution giving the most benefit will be chosen as the solution for the next part. **Example:** Fractional Knapsack, Activity Selection.

9. **Dynamic Programming Algorithm:** This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them. The approach of Dynamic programming is similar to divide and conquer.

The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table. **Example:** 0-1 Knapsack, subset-sum problem.

10. **Randomized Algorithm:** In the randomized algorithm we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Classification by Design Approaches:

There are two approaches for designing an algorithm. these approaches include

1. Top-Down Approach
2. Bottom-up approach

- **Top-Down Approach:** In the top-down approach, a large problem is **divided into small sub-problem** and keep **repeating the process** of decomposing problems until the complex problem is solved. Breaking down a complex problem into smaller, more manageable sub-problems and solving each sub problem individually. Designing a system starting from the highest level of abstraction and moving towards the lower levels.

- **Bottom-up approach:** The bottom-up approach is also known as the reverse of top-down approaches. In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program. Building a system by starting with the individual components and gradually integrating them to form a larger system. Solving sub-problems first and then using the solutions to build up to a solution of a larger problem. Both approaches have their own advantages and disadvantages and the choice between them often depends on the specific problem being solved.

1. **Designing Algorithm -** Algorithm is a branch in Computer Science that consists of **designing and analyzing** computer algorithms:

The **"design"** pertains to :

(i) The **description of algorithm** at an abstract level by means of a pseudo language
(ii) **Proof of correctness** that is, the algorithm solves the given problem in all cases.

Program Design - There are various ways by which we can specify an program design. Those are -

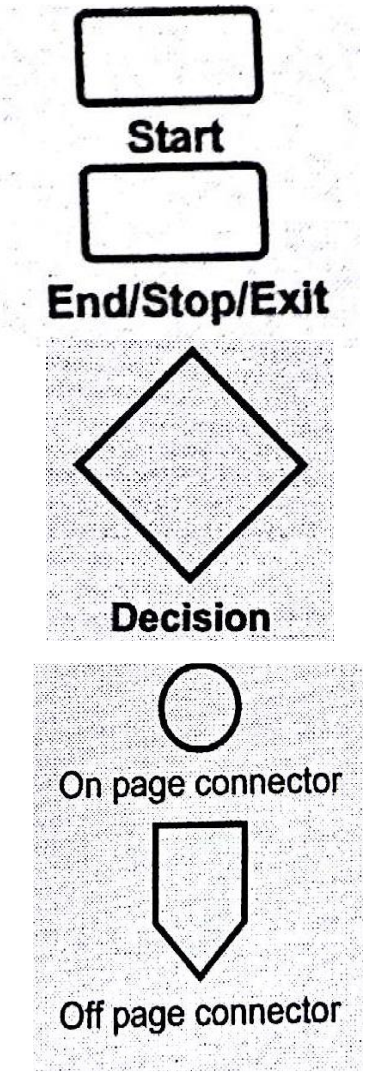
1. Use of Pseudocode 2. Representation of Flowchart

Pseudocode Definition: Pseudo code is nothing but an **informal way of writing** a Program. It is a **combination of algorithm** written in English and **some**

programming language. There is **no restriction** of following **syntax** of programming language. Pseudo code **cannot be compiled;** it is just a previous step of developing a code in algorithm.

Flowchart - Flowcharts are the **graphical representation** of the algorithms. The algorithms and flowcharts are the **final steps in organizing** the solutions. Using the algorithms and flowcharts the programmers can **find out the bugs** in the programming logic and then can go for coding. Flowcharts can **show errors** in the logic and set of data can be easily tested using flowcharts.

Symbols used in Flowchart



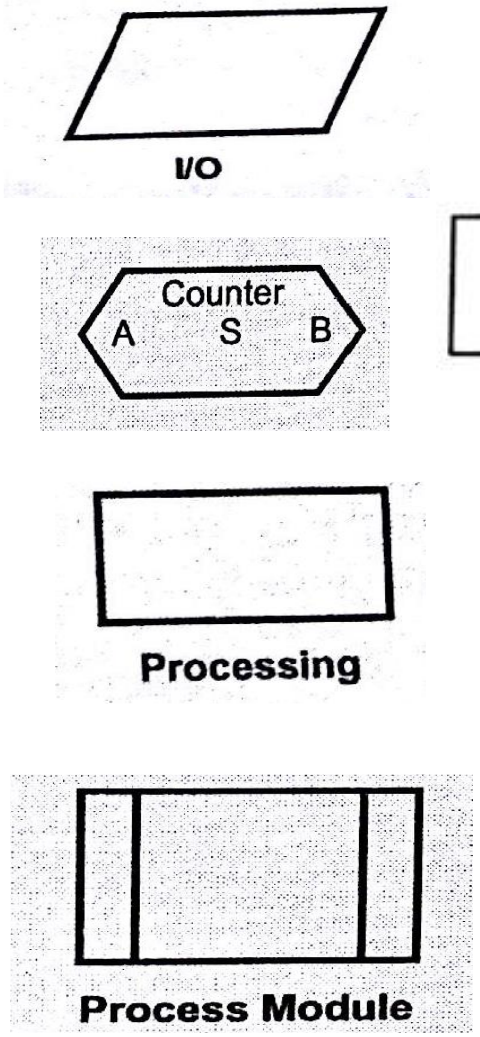


Figure 9 symbol of flowchart

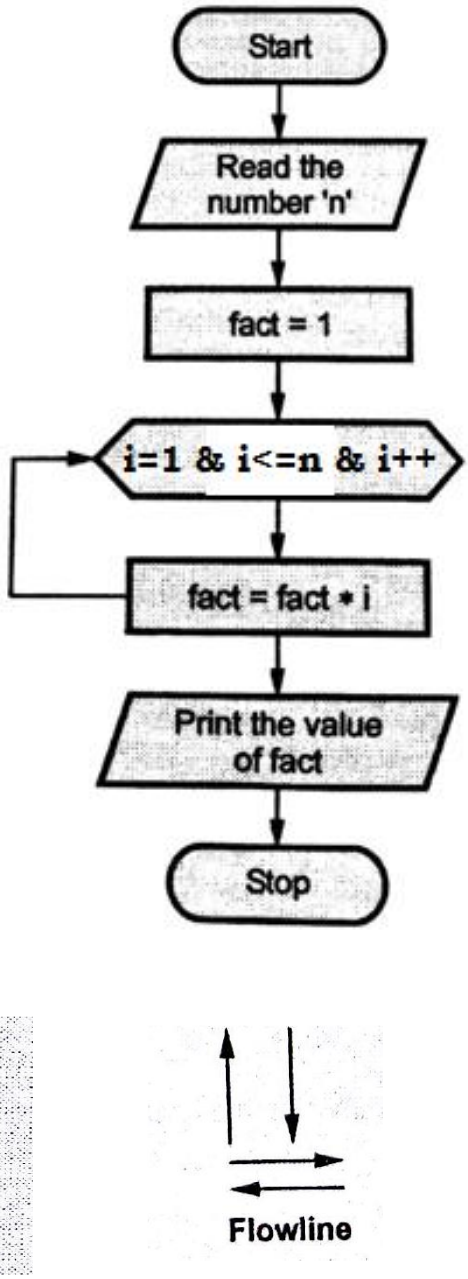


Figure 10 Flowchart of factorial

2. Algorithm Analysis - The **"analysis"** deals with **performance evaluation** (complexity analysis). The algorithm can be **analyzed in two levels**, i.e., first is **before creating** the algorithm, and second is **after creating** the algorithm. The following are the two analysis of an algorithm:

1. **Priori Analysis:** Here, priori analysis is the **theoretical analysis** of an algorithm which is **done before implementing** the algorithm. Various f**actors**

can be considered before implementing the algorithm like **processor speed, memory size** etc. which has no effect on the implementation part.

2. **Posterior Analysis:** Here, posterior analysis is a **practical analysis** of an algorithm. The practical analysis is **achieved by implementing** the algorithm using any programming language. This analysis basically evaluate that how **much running time and space** taken by the algorithm.

Performance Analysis

The **efficiency of algorithm** can be decided by measuring the performance of algorithm. We can measure the performance of an algorithm by counting two factors

1. **Amount of time required** by the algorithm to execute. This is known as **time complexity.**
2. **Amount of space required** by the algorithm to execute. This is known as **space complexity.**

Suppose we want to find out the time taken by following program statement

x = x+1

Determining the amount of time required by the above statement in terms of clock time is **not possible** because it is **always dynamic**. Following are always dynamic

1. The machine that is used to execute the programming statement
2. Machine language instruction set
3. Time required by each machine instruction.
4. The translation of compiler will make for this statement to machine language.
5. The kind of operating system (multi—programming or time sharing)

The above information varies from machine to machine. Hence it is not possible to find out the exact figure. Hence the performance of the machine is measured in terms of frequency count.

Frequency Count and its Importance –

Definition: The frequency count is a count that denotes **how many times particular statement is executed**. For Example Consider following code for counting the frequency count

```
void fun()
{
  int a=10;
  a++;        .....................1
  printf ("%d, a");        .....................1
}
```

The frequency count of above program is 2.

Algorithm Complexity

The term algorithm complexity measures **how many steps are required** by the algorithm to solve the given problem. It evaluates the order of **count of operations executed** by an algorithm as a function of input data size. To assess the complexity, the order (approximation) of the **count of operation is always considered** instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The **complexity of the asymptotic** computation O(f) determines in which **order the resources** such as CPU time, memory, etc. **are consumed** by the algorithm that is articulated as a function of the size of the input data.

The **complexity can be** found in any form such as **constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential,** etc. It is nothing but the order of constant, logarithmic, linear and so on, the **number of steps** encountered for the completion of a particular algorithm. To make it **even more precise**, we often call the complexity of an algorithm as "**running time".**

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n). The complexity of an algorithm can be divided into two types. The **time complexity** and the **space complexity**.

Space Complexity

Space complexity is defining as the **process of defining a formula** for prediction of **how much memory space is required** for the successful execution of the algorithm. The memory space is generally considered as the primary memory. To compute the space complexity we use **two factors: constant and instance** characteristics. The space requirement S(p) can be given as

S(P)=C+Sp

where **C** is a **constant** i.e. **fixed part** and it denotes the **space of inputs** and **outputs.** This space is an amount of space taken by **instruction, variables and identifiers**. And **Sp** is a **space** dependent upon **instance characteristics**. This is a **variable part** whose space requirement depends on particular problem instance.

There are **two types of components** that contribute to the space complexity

1. **Fixed part and**
2. **variable part**.

The **fixed part includes space** for:-

1. Instructions 2. Variables 3. Array 4. Space for constants.

The **variable part includes space** for:-

1. The variables **whose size is dependent** upon the **particular problem** instance being solved. The control statements (such as **for, do, While, choice**) are used to solve such instance
2. Recursion stack for handling **recursive call.**

Time Complexity –

The time complexity is defined as the **process of determining a formula** for **total time required** towards the **execution** of that algorithm is called the time complexity of that algorithm. This calculation is **totally independent** of implementation and programming language. For determining the time complexity of particular algorithm **following steps** are carried out.

1. **Identify the basic operation** of the algorithm
2. **Obtain the frequency count** for this basic operation.
3. **Consider the order of magnitude** of the frequency count and express it in terms of big oh notation.

Typical Complexities of an Algorithm

1. **Constant Complexity :** It imposes a complexity of **O(1)**. It undergoes an execution of a **constant number of steps** like 1, 5, 10, etc. for solving a given problem. The count of **operations is independent of the input** data size.

2. **Logarithmic Complexity :** It imposes a complexity of **O(log(N))**. It undergoes the **execution of the order of log(N)** steps. To perform operations on **N elements**, it often takes the **logarithmic base as 2.** For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.

3. **Linear Complexity :** It imposes a complexity of **O(N)**. It encompasses the **same number of steps** as that of the **total number of elements** to implement an operation on N elements. For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.

   It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem. For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

4. **Quadratic Complexity:** It imposes a complexity of **$O(n^2)$**. For **N input data size**, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem. If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.

5. **Cubic Complexity:** It imposes a complexity of **$O(n^3)$**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem. For example, if there exist 100 elements, it is going to execute 1,000,000 steps.

6. **Exponential Complexity:** It imposes a complexity of **$O(2^n)$, $O(N!)$, $O(n^k)$, ...** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size. For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Algorithm analysis is an important part of **computational complexity theory**, which provides theoretical estimation for the **required resources** of an algorithm to solve a specific computational problem. Analysis of algorithms is the **determination** of the **amount of time and space** resources required to execute it. Complexity **affects performance** but **not vice-versa.** Be careful to differentiate between:

- **Performance:** How much **time/memory/disk/etc. is used** when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
- **Complexity:** How **do the resource requirements** of a program or algorithm **scale**, i.e. what happens as the size of the problem being solved by the code gets larger.

Why Analysis of Algorithms is important?

- To **predict the behaviour** of an algorithm without implementing it on a specific computer.
- It is much more **convenient to have simple measures for the efficiency** of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is **impossible to predict the exact behaviour** of an algorithm. There are too many influencing factors.
- The analysis is thus **only an approximation**; it is **not perfect.**
- More importantly, by analysing different algorithms, **we can compare** them to **determine the best one** for our purpose.

Types of Algorithm Analysis:

- **Best case:** Define the input for which algorithm **takes less time or minimum time.** In the best case calculate the lower bound of an algorithm.

- **Worst Case**: Define the input for which algorithm takes a **long time or maximum time**. In the worst calculate the upper bound of an algorithm.

- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs. And then we divide it by the total number of inputs. **Average case** = all random case time / total no of case

How to calculate time complexity

Example 1: Algorithm SUM(A, n)

```
Start
s=0------------------------1
For i=0;i<n;i++---------n+1
  s=s+i------------------n
Return s-------------------1
End
```

Time Complexity

T(n)=1+n+1+n+1
T(n)=2n+3
T(n)=O(n)

Example 2: Algorithm ADDMAT(A, B, n)

```
Start
For i=0;i<n;i++  ----------------n+1
  For j=0;j<n;j++ -----------n * (n+1)
          C[i][j]=A[i][j]+B[i][j]--------n(for I loop)*n(for j loop)
Return C[i][j]-----------------------------------------------------1
End
```

Time Complexity

T(n)= n+1+n*(n+1)+n*n+1+1
T(n)=n+1+n^2+n+n^2+2
T(n)=2n^2+2n+2
T(n)=O(n^2)

Example 3: Algorithm MULMAT(A, B, n)

```
Start
For i=0;i<n;i++           .............n+1
  For j=0;j<n;j++      .............n*(n+1)
          C[i][j]=0                 ................n*n
For k=0;k<n;k++        ......................n+1
  C[i][j]=C[i][j]+A[j][k]*B[K][j]       ...........n
Return C[i][j]                          ..............1
End
```

Time Complexity

T(n)= n+1+n*(n+1)+n*n+n+1+n+1
T(n)= n+1+n^2+n+n^2+n+1+n+1
T(n)= 2n^2+4n+4
T(n)= O(n^2)

Asymptotic notations

To **choose the best algorithm**, we need to **check efficiency** of each algorithm. The efficiency can be measured by **computing time complexity** of each algorithm. Asymptotic notation is a **shorthand way to represent** the time complexity. Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time". **Various notations** such as Ω-Omega, θ-theta and O-Big O used are **called asymptotic notations.**

1. **Big oh Notation -** The Big oh notation is **denoted by "O"**. It is a method of representing the **upper bound of algorithm's** running time. Using big oh notation we can give **longest amount of time** taken by the algorithm to complete.

Definition - Let F(n) and g(n) be two non-negative functions. Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly, c is some constant such that $c > 0$. We can write **F(n)<= c*g(n)**

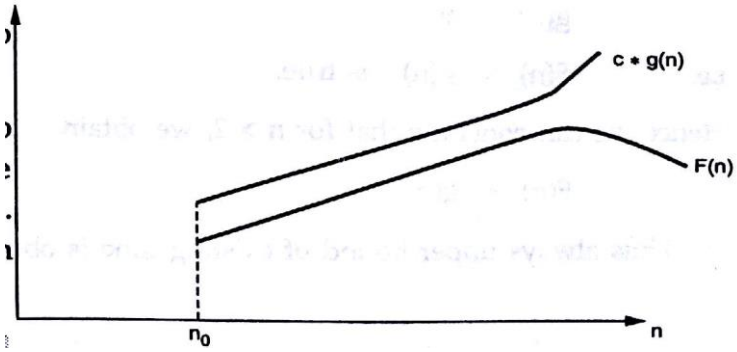


Figure 11 Big OH notation

Then F(n) is big oh of g(n). It is also denoted as F(n) ϵ O (g(n)). In other words F(n) is less than g(n) if g(n) is multiple of some constant c.

Example : Consider function F(n) = 2n + 2 and g(n) = n^2 . Then we have to find some constant c, so that F(n)<= c * g(n). As F(n) = 2n + 2 and g(n) = n^2 then we find c

for n = 1 then
F(n) = 2n+ 2
= 2(1) + 2
F(n) = 4
and g(n) = n^2
= $(1)^2$
g(n) = 1
i.e. F(n) > g(n) '

If n = 2 then
F(n) = 2(2) + 2
= 6
g(n) = $(2)^2$
g(n) = 4
i.e. F(n) > g(n)

If n = 3 then
F(n) = 2(3) + 2
= 8

g(n) = (3)²
g(n) = 9

i.e. F(n) < g(n) is true.

Hence we can conclude that for n > 2, we obtain .
F(n) < gm)

Thus always upper bound of existing time is obtained by big oh notation.

2. Omega Notation - Omega notation is **denoted by "Ω".** This notation is used to represent the **lower bound of algorithm's** running time. Using omega notation we can denote **shortest amount of time** taken by algorithm.

Definition - A function F(n) is said to be in **Ω**(g(n)) if F(n) is bounded below by some positive constant multiple of g(n) such that

F(n) => C * g(n) For all n >n_0

It is denoted as F(n) ϵ Ω (g(n)). Following graph illustrates the curve for Ω notation.

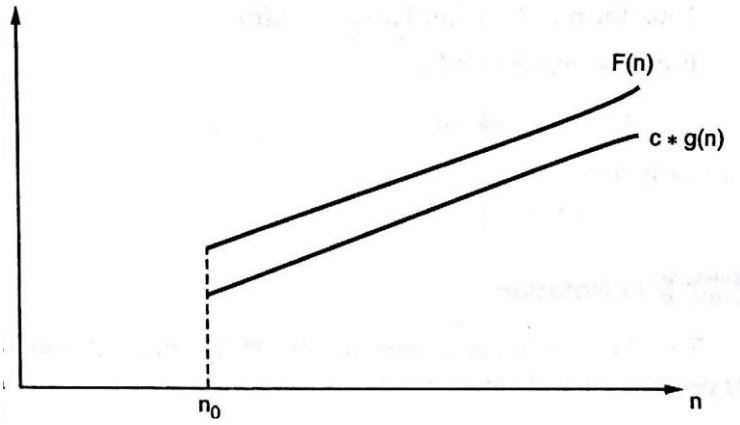


Figure 12 Omega notation F(n) ϵ Ω (g(n))

Example: Consider **F(n) = 2n² + 5 and g(n) = 7n**

Then if n = 0
F(n) = 2(0)² + 5
g(n) = 7 (0)
= 0
i.e. F(n) > g(n)

But if n = 1
F(n) = 2(1)² + 5
. = 7
g(n) = 7(1)
= 7
i.e. F(n) = g(n)

If n = 3 then,
F(n) = 2(3)² + 5
= 18 + 5
= 23
g(n) = 7(3)
= 21
i.e. F(n) > g(n)

Thus for n > 3 we get F(n) > c * g(n). It can be represented as $2n^2 + 5$ ϵ Ω (n)

3. Θ Notation - The theta notation is **denoted by Θ**. By this method the running **time is between upper bound and lower bound.**

Definition - Let F(n) and g(n) be two non negative functions. There are two positive constants namely c1 and c2 such that: c1 g(n) <= F(n) <= c2 g(n). Then we can say that : **F(n) ϵ θ (g(n))**

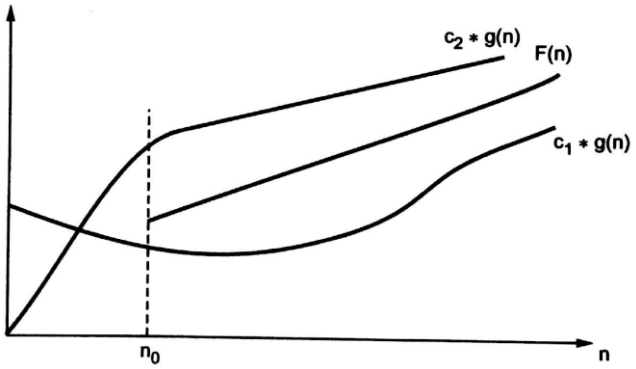


Figure 13 Theta notation F(n) ϵ θ (g(n))

Example: If F(n) = 2n + 8 and g(n) = 7n.
where n => 2
Similarly F(n) = 2n + 8
g(n) = 7n
i.e. 5n < 2n + 8 < 7n
For n=>2
Here c1 = 5 and c2 = 7 with n_0 = 2

The theta notation is more precise with both big oh and omega notation.

Order of Growth

If we have two algorithms that perform same task and the first one has a computing time of O(n) and the second of $O(n^2)$, then we will usually Prefer the first one. The reason for this is that as n increases the time required for the execution of second algorithm far more than the time required for the execution of first.

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648

Figure 14 order growth table

Types of Algorithm Analysis:

- **Best case:** Define the input for which algorithm **takes less time or minimum time.** In the best case calculate the lower bound of an algorithm.
- **Worst Case**: Define the input for which algorithm takes a **long time or maximum time**. In the worst calculate the upper bound of an algorithm.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs. And then we divide it by the total number of inputs. **Average case** = all random case time / total no of case.

Best, Worst, and Average ease Efficiency - Let us assume a list of n number of values stored in an array. Suppose if we want to search a particular element in this list, the algorithm that search the key element in the list among n elements, by comparing the key element with each element in the list sequentially.

The **best case** would be if the **first element in the list matches** with the key element to be searched in a list of elements. The efficiency in that case would be expressed as **O(1)** because only one comparison is enough. Minimum number of comparisons = **1**

Similarly, the **worst case** in this scenario would be if the **complete list is searched** and the **element is found only at the end** of the list or is not found in the list. The efficiency of an algorithm in that case would be expressed as **O(n)** because **n** comparisons required to complete the search. Maximum number of comparisons = **n**

The **average case** efficiency of an algorithm can be obtained by **finding the average number of comparisons** as given below:

If the element not found then maximum number of comparison = **n** Therefore, average number of comparisons = **(n + 1)/2** Hence the **average case** efficiency will be expressed as **O (n).**

Time and space trade-off

Time space trade-off is basically a situation where either a space efficiency (memory utilization) can be achieved at the cost of time or a time efficiency (performance efficiency) can be achieved at the cost of memory.

Example 1 : Consider the programs like compilers in which symbol table is used to handle the variables and constants. Now if entire symbol table is stored in the program then the time required for searching or storing the variable in the symbol table will be reduced but memory requirement will be more. On the other hand, if we do not store the symbol table in the program and simply compute the table entries then memory will be reduced but the processing time will be more.

Example 2 : Suppose, in a file, if we store the uncompressed data then reading the data will be an efficient job but if the compressed data is stored then to read such data more time will be required.

Example 3 : This is an example of reversing the order of the elements. That is, the elements are stored in an ascending order and we want them in the descending order. This can be done in two ways -

i. We will use another array b[ ] in which the elements in descending order can be arranged by reading the array an in reverse direction. This approach will actually increase the memory but time will be reduced.

ii. We will apply some extra logic for the same array all to arrange the elements in descending order. This approach will actually reduce the memory but time of execution will get increased.

Thus time space trade-off is a situation of compensating one performance measure at the cost of another.