# Unit-1
# Introduction

# Introduction to Algorithms

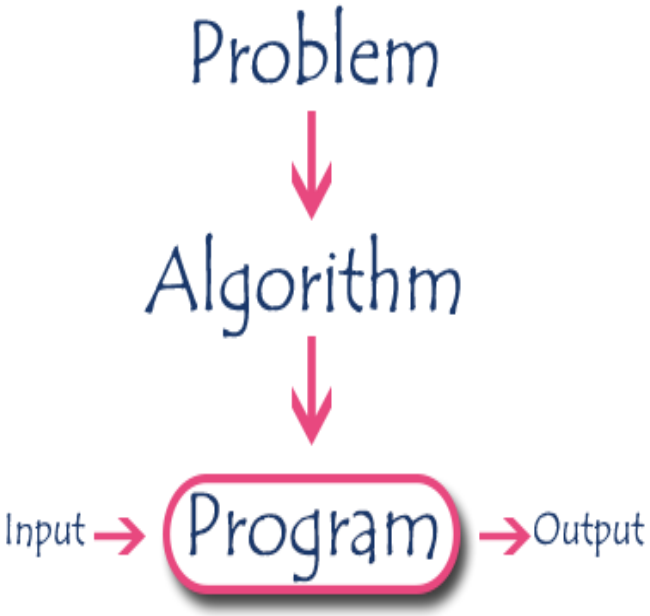
## What is an algorithm?

- An algorithm is a step by step procedure to solve a problem.

- In other words an algorithm is defined as a sequence of statements which are used to perform a task.

- Algorithms are used to convert our problem solution into step by step statements.

- These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as input, processes data according to the program instructions and finally produces a result as shown in the following picture.



## Algorithm Properties:

1. Every algorithm must take zero or more number of input values.

2. Each and every instruction should be in a simple language.

3. The number of steps should be finite.

4. The steps mentioned in an algorithm can be executable by the computer.

5. It should have an output.

## Example for an Algorithm

Let us consider the following problem for finding the largest number in a given list of numbers.

## Algorithm

**Step 1:** Define a variable **max=0**.

**Step 2:** Compare first number (say ‘a') in the list 'L' with 'max', **if (a>max),** set **max=a.**

**Step 3:** Repeat step 2 for all numbers in the list 'L'.

**Step 4:** Display the value of '**max**' as a result.

# Code using C Programming Language:

```
int find_larger_number(L)
 {
        int max = 0,      i;
        for(i=0;    i < Size;    i++)
        {
                if (L[i] > max)
                        max = L[i];
        }
                return max;
}
```

# What is Performance Analysis of an algorithm?

- If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train, by motorcycle and also by cycle. Depending on the availability and convenience, we choose the one which suits us.
- Similarly, in computer science, **to solve a problem there are multiple algorithms.** When we have more than one algorithm to solve a problem, we need to select the best one.
- Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

- We compare algorithms with each other which are solving the same problem, to select the best algorithm. **To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,**

  Generally, the performance of an algorithm depends on the following elements…

- Whether that algorithm is providing the exact solution for the problem?
- Whether it is easy to understand?
- Whether it is easy to implement?
- How much space (memory) it requires to solve the problem?
- How much time it takes to solve the problem? Etc.,

- When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.
- Based on this information, performance analysis of an algorithm can be defined as follows:

  **"Performance analysis of an algorithm is the process of calculating space and time required by that algorithm."**

Performance analysis of an algorithm is performed by using the following measures:

- Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
- Time required to complete the task of that algorithm (**Time Complexity**)

# Space Complexity

## What is Space complexity?

- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...
  1. To store program instructions.
  2. To store constant values.
  3. To store variable values.
  4. And for few other things like function calls, jumping statements etc.,.

Space complexity of an algorithm can be defined as follows...

**Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm**

- To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler).

- For example, the C Programming Language compiler requires the following…
  - ❖2 bytes to store Integer value.
  - ❖4 bytes to store Floating Point value.
  - ❖1 byte to store Character value.
  - ❖6 (OR) 8 bytes to store double value.

## Example 1

- Consider the following piece of code...

```
int square(int a)
{
        return a*a;
}
```

- In the above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.
- That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity*.

**If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity**

## Example 2

- Consider the following piece of code…

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

In the above piece of code it requires

- **n*2** bytes of memory to store array variable **a[ ]**
- 2 bytes of memory for integer parameter **n**
- 4 bytes of memory for local integer variables **sum** and **i** (2 bytes each)
- 2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

# Time Complexity

## What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

Time complexity of an algorithm can be defined as follows...

**The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.**

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine.
3. Read and Write speed of the machine.
4. The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
5. Input data

When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

Consider the following piece of code...

**Example 1**

```
int sum(int a, int b)
{
        return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

# Example 2

Consider the following piece of code...

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

- If we increase the **n** value then the time required also increases linearly.

**If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity**

## What is Data Structure?

- Whenever we want to work with a large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

- A data structure can be defined as follows:

Data structure is a method of organizing a large amount of data more efficiently so that any operation on that data becomes easy

Based on the organizing method of a data structure, data structures are divided into two types.

1. Linear Data Structures
2. Non - Linear Data Structures

# Linear Data Structures

**If a data structure is organizing the data in *sequential order*, then that data structure is called as Linear Data Structure.**

## Example

1. Arrays
2. List (Linked List)
3. Stack
4. Queue

# Non - Linear Data Structures

If a data structure is organizing the data in *random order*, then that data structure is called as Non-Linear Data Structure.

## Example

1. Tree
2. Graph

# What is a data type?

Two important things about data types:

1. Defines a certain domain of values.
2. Defines Operations allowed on those values.

Example:

```
int type
 -  Takes only integer values
 -  Operations: addition, subtraction,
    multiplication, bitwise operations etc.
```

# User defined data types

In contrast to primitive data types, there is a concept of user defined data types.

The operations and values of user defined data types are not specified in the language itself but is specified by the user.

Example: Structure, union and enumeration (enum)

By using structures, we are defining our own type by combining other data types.

```
struct point {
    int x;
    int y;
};
```

# ADT

- The term ADT refers to **Abstract Data Type** in Data Structures.
- An **abstract data type (ADT)** is characterized by:

  -a set of values

  -a set of operations
- **ADT** are like user defined data types which defines operations on values using function without specifying what is there inside the function and how the operations are performed.
- ADT is a conceptual model of information structure.
- ADTs are independent of data representation and implementation of operations.

EXAMPLE: Stack ADT

A stack consists of elements of same type arranged in a sequential order.

Operations:

initialize() - initializing it to be empty
Push() - Insert an element into the stack
Pop() - Delete an element from the stack
isEmpty() - checks if stack is empty
isFull() - checks if stack is full

# Array implementation of Data Structure

- An array is a data structure for storing more than one data item that has a similar data type.

**or**

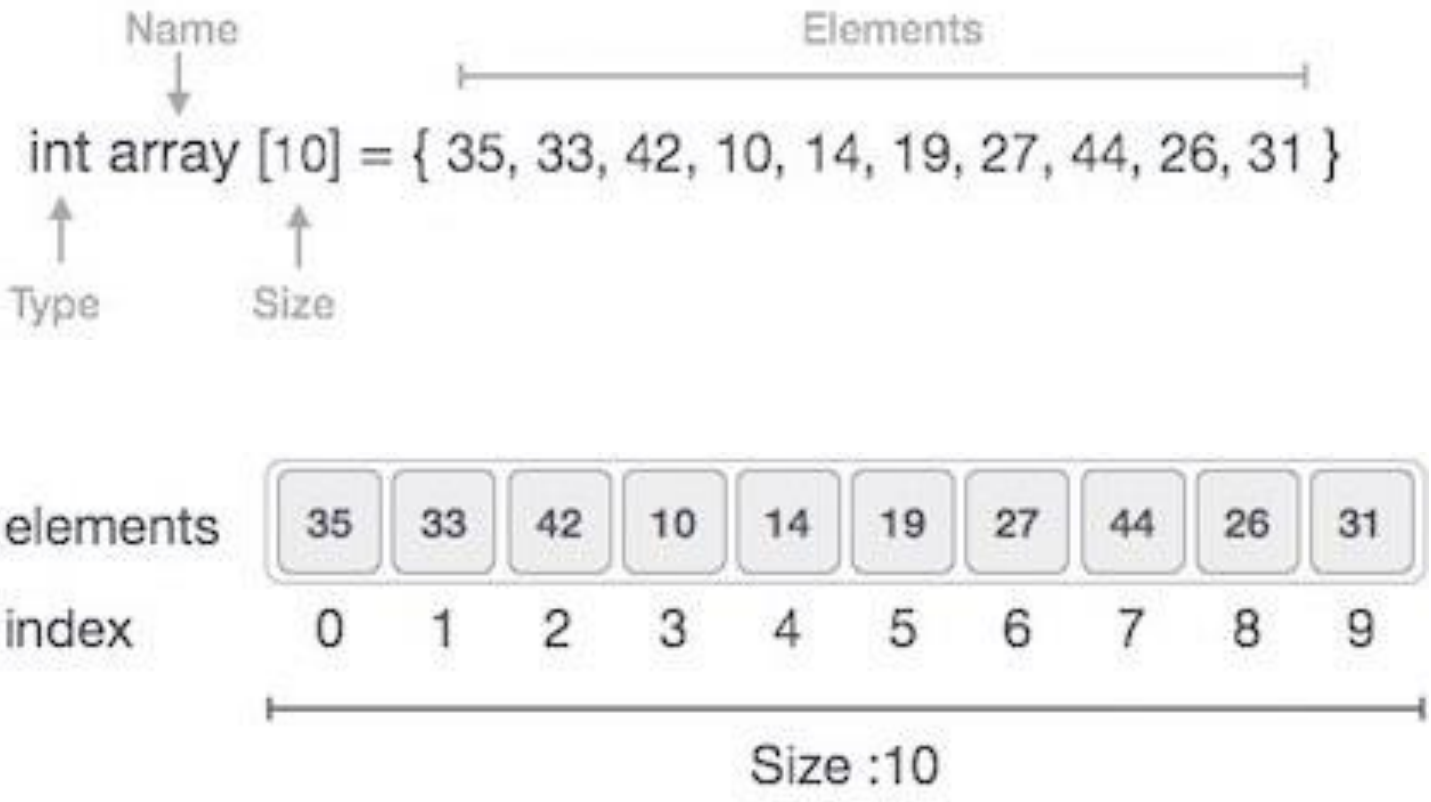
- Array is a container which can hold a fix number of items and these items should be of the same type.

- Most of the data structures make use of arrays to implement their algorithms.

- Following are the important terms to understand the concept of Array.
- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

# Array Representation

- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



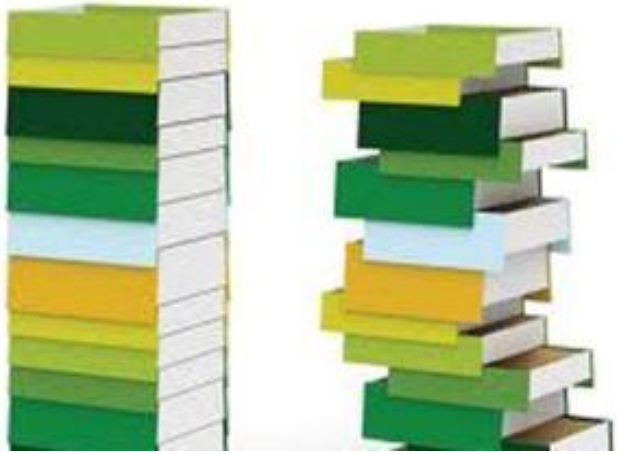
# Stack as an abstract data type and operation

**Definition:**

Stack is **First-In-Last-Out** [FILO] or **Last-In-First-Out** [LIFO] Structure.

**Explanation:**

- Stack is Data Structure used to store the data in such a way that element inserted into the stack will be removed at last.
- A stack is a list of elements in which an element may be inserted or deleted only at one end called Top of Stack [TOS].
- Just take real time example, suppose we have created stack of the book like this shown in the following fig –



A stack of elements of type T is a finite sequence of elements together with the operations

1. **CreateEmptyStack(S)**: create or make stack **S** be an empty stack
2. **Push(S,x):** Insert **x** at one end of the stack, called its **top**
3. **Top(S):** If stack **S** is not empty; then retrieve the element at its **top**
4. **Pop(S):** If stack **S** is not empty; then delete the element at its **top**
5. **IsFull(S):** Determine if **S** is full or not. Return **true** if **S** is full stack; return **false** otherwise
6. **IsEmpty(S)**: Determine if **S** is empty or not. Return **true** if **S** is an empty stack; return **false** otherwise.

# Queue as an abstract data type and operation

- Queue is a linear list of elements in which deletion of an element can take place at one end, called the **front** and insertion can take place at the other end, called the **rear**.

- The first element in a queue will be the first one to be removed from the list.

- Queues are also called **FIFO** (**First In First Out**) i.e. the data item stored first will be accessed first

# The queue as an ADT

- A queue **q** of type **T** is a finite sequence of elements with the operations
  - ***MakeEmpty(q):*** To make **q** as an empty queue
  - ***IsEmpty(q):*** To check whether the queue **q** is empty. Return true if **q** is empty, return false otherwise.
  - ***IsFull(q):*** To check whether the queue **q** is full. Return true in **q** is full, return false otherwise.
  - ***Enqueue(q, x):*** To insert an item **x** at the rear of the queue, if and only if **q** is not full.
  - ***Dequeue(q):*** To delete an item from the front of the queue **q** if and only if **q** is not empty.
  - ***Traverse (q):*** To read entire queue that is display the content of the queue.

# Enqueue Operation

- Queue maintains two data pointers, **front** and **rear**
- The following steps should be taken to **enqueue**(insert) data into queue –
  - Step 1 – Check if queue is full
  - Step 2 – if queue is full
    produce overflow error and exit
    else
    increment rear pointer to point next empty space
    and add data element to the queue location, where rear is pointing
  - Step 3 - return success

# Dequeue Operation

- Accessing data from queue is a process of two steps
  - Access the data from where **front** is pointing
  - And remove the data after access

- The following steps are taken to perform **dequeue** operation
  - Step 1 – Check if queue is empty
  - Step 2 – if queue is empty

    produce underflow error and exit

    else

    access data where front is pointing, increment front pointer to point next available data element
  - Step 3 – return success.

**Note: [slide 25 to 31 only for knowledge]**

**What is user-defined data type name? Explain with examples.**

**Ans:** C language allows user to define their own data types which are based on existing data types.

The general format of declaring data type is

**typedef existing _data_type user_defined_data_type;**

**Where:** typedef is keyboard.

existing_data_type is C data type.

user_defined_data_type is new data type name which user wants to define.

**Example:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
      typedef   int  surajhekka;
      surajhekka  a, b, sum;
      clrscr();
      printf("Input the value for a and b:");
      scanf("%d %d", &a,&b);
      sum=a+b;
      printf("The sum of a and b is %d",sum);
      getch();
}
```

**Output**

```
Input the value for a and b: 4          5
The sum of a and b is 9
```

# enum in C

- **Enumeration (enum)** is a user-defined datatype (same as structure). It consists of various elements of that type. There is no such specific use of enum, we use it just to make our codes neat and more readable. We can write C programs without using enumerations also.

- For example, Summer, Spring, Winter and Autumn are the names of four seasons. Thus, we can say that these are of types season. Therefore, this becomes an enumeration with name season and Summer, Spring, Winter and Autumn as its elements.

## Defining an Enum

- An enum is defined in the same way as structure with the keyword struct replaced by the keyword enum and the elements separated by 'comma' as follows.

- **enum enum_name**
  **{**
  **element1,**
  **element2,**
  **element3,**
  **element4,**
  **};**

Now let's define an enum of the above example of seasons.

```
enum Season
{
  Summer,
  Spring,
  Winter,
  Autumn,
};
```

## Declaration of Enum Variable

We also declare an enum variable in the same way as that of structures. We create an enum variable as follows.

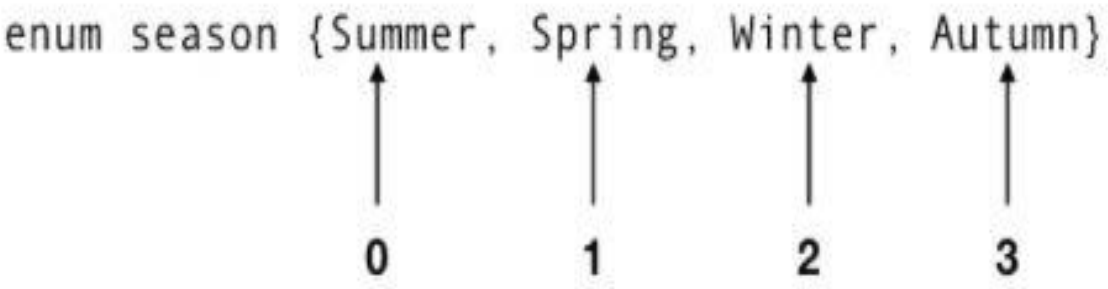
```
enum season
{
  Summer,
  Spring,
  Winter,
  Autumn
};
main()
{
  enum season s;
}
```

- So, here 's' is the variable of the enum named season. This variable will represent a season. We can also declare an enum variable as follows.

```
enum season
{
  Summer,
  Spring,
  Winter,
  Autumn
}s;
```

# Values of the Members of Enum

- All the elements of an enum have a value. By default, the value of the first element is 0, that of the second element is 1 and so on.




Let's see an example.

```c
#include <stdio.h>
enum season{ Summer, Spring, Winter, Autumn};
int main()
{
    enum season s;
    s = Spring;
    printf("%d\n",s);
    return 0;
}
```

# Let's see one more examples of enum.

```c
#include <stdio.h>
enum days{ sun, mon, tue = 5, wed, thurs, fri, sat};
int main()
{
    enum days day;
    day = thurs;
    printf("%d\n",day);
    return 0;
}
```

```c
#include <stdio.h>
enum days{ sun, mon, tue, wed, thurs, fri, sat};
int main()
{
    enum days day;
    day = thurs;
    printf("%d\n",day+2);
    return 0;
}
```