 # Batch Name : OM32
 # Module Name : Data Structures
==

 # DS DAY-01:
+ Introduction to an DS:
- if we want to store marks of 100 students

```
int m1, m2, m3, m4, m5, ......., m100;//sizeof(int)*100 =
400 bytes
```

if we want to sort marks of 100 students =>

```
int marks[ 100 ];//sizeof(int)*100 = 400 bytes
```

+ "array" => an array is a basic/linear data structure which is a collection of logically related similar type of elements gets stored into the memory at contiguos locations.

```
int arr[ 5 ];


arr : int []
arr[ 0 ] : int
arr[ 1 ] : int
.
.
.
```

primitive data types: char, int, float, double, void
non-primitive data types: array, structure, pointer, enum

- we want to store info of 100 students

```
rollno   : int
name: char []/string
marks    : float
```

+ "structure" => it is a basic/linear data structure, which is a collection of logically related similar and disimmilar type of data elements gets stored into the memory collectively as as single record/entity.

```
struct student
{
	int rollno;
	char name[ 32 ];
	float marks;
};
```

C => Array
C++ => Array
Java => Array
Python => Array

=> data structures is a programming concept
=> to learn data structures is not learn any programming language, it is nothing but to learn an algorithms, data structure algorithms can be implemented in any programming language.
=> in this course we will use C programming language.
Prerequisite: C

Q. What is a Program?
Q. What is an algorithm?
Q. What is a Pseudocode?

- to traverse an array => to visit each array element sequentially from first element max till last element.

+ "algorithm" => to do sum of array elements => any human user
step-1: intially take sum var as 0

step-2: traverse an array and add each array element sequentially into the sum variable
step-3: return final sum

+ "pseudocode" => to do sum of array elements => programmer user

```
Algorithm ArraySum(A, n){//whereas A is an array of size
"n"
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }

     return sum;
}
```

- pseudocode is a special form of an algorithm in which finite set of instructions can be written in human understandable langauge with some programming constraints.

+ "program" => to do sum of array elements => machine

```
int array_sum(int arr[], int size){
     int sum = 0;
     int index;

     for( index = 0 ; index < size ; index++ )
          sum += arr[ index ];

     return sum;
}
```

flowchart => it is a digramatic representation of an algorithm.

=> an algorithm is a solution of a given problem.
=> an algorithm = solution
- "one problem may has many solutions", and in this case there is a need to decide an efficient solution.

e.g. searching => to find/search a key element in a given collection/list of data elements.

1. linear search
2. binary search

e.g. **sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.**

1. selection sort
2. bubble sort
3. insertion sort
4. quick sort
5. merge sort

etc...

- to decide effciency of an algorithms, we need to do their analysis
- there are two measures of an analysis of an algorithms:

1. time complexity
2. space complexity

linear search =>

```
step-1: accept key from user
step-2:
    for( index = 1 ; index <= size ; index++ ){
        //if matches with any array element
        if( key == arr[ index ] )
            return true;
    }

    //if key do not matches with any array element
    return false;
```

if key is found in an array at very first pos

if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 50 => no. of comparisons = 1
.

.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in this case = 1 => best case
running time of an algo in best case = O(1).

+ worst case:

if either key is found in an array at last pos or key do not found
if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
if size of an array = 50 => no. of comparisons = 50
.
.
if size of an array = n => no. of comparisons = n

no. of comparisons = depends on size of an array
for any input size array no. of comparisons in this case = n => worst case
running time of an algo in worst case = O(n).

+ asymptotic rules: (descrete maths)
"rule-1" : if running time of an algo is having any additive/substractive/divisive/multiplicative constant then it can be neglected.
e.g.
    O(n+3) => O(n)
    O(n-5) => O(n)
    O(2*n) => O(n)
    O(n/2) => O(n)

```
typedef unsigned long int size_t;
```

2. binary search:

by means of calculating mid pos big size array gets divided logically into two subarray's => left subarray & right sub array
left subarray => left to mid-1
right subarray => mid+1 to right

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

if(left == right) => subarray contains only 1 ele and it is valid
if(left <= right) => subarray is valid
in other words :
if(left > right) => subarray is invalid

DS DAY-02:

if size of an array = 1000
iteration-1: search space = n => 1000

[0 ...... 999]
[0.... 499] [501 ..... 999]

iteration-2: search space = n/2 = 500
[0.... 499]
[0...249] [251 ....499]

iteration-3: search space = n/2 / 2 => n / 4 = 250
[0...124] [126 ...249]

iteration-4: search space = n/4 / 2 => n / 8 = 125

after every iteration search space is getting reduced by half
n => n/2 => n/4 => n/8 ....
n => n / 2^0
n/2 => n / 2^1
n/4 => n / 2^2
n/8 => n / 2^3
- search space is getting reduced exponetially

- binary search is also called as logarithmic search, and hence we get time complexity of binary search in terms of log.

Rule => if any algo follows divide-and-conquer approach then we get time complexity of that algo in terms of log.

Best Case => if key is found at root position in only 1 comparison => O(1) => $\Omega(1)$.

Worst Case => if either key is found at leaf position or key is not found => O(log n) => O(log n)

Average case => if key is found at non-leaf position
=> O(log n) => Θ(log n).

+ Sorting Algorithms:
1. Selection Sort:
iteration-1: no. Of comparisons = n-1
iteration-2: no. Of comparisons = n-2
iteration-3: no. Of comparisons = n-3
.
.
.
iteration-(n-1): no. Of comparisons = 1

total no. of comparisons = (n-1)+(n-2)+(n-3)+....+ 1
arithmetic progression formula:
=> n(n-1) / 2
=> $(n^2 - n) / 2$
=> $O((n^2 - n) / 2)$
=> $O(n^2 - n)$ ...by neglecting divisive constant
=> $O(n^2)$ .... by using rule of polynomial only leading term is considered

rule => if running time of an algo is having a polynomial, then in its time complexitiy of leading term will be considered.
e.g.
$O(n^3 + n^2 + n - 3) => O(n^3)$
$O(n^2 + 5) => O(n^2)$

2. Bubble Sort:
iteration-1: no. Of comparisons = n-1
iteration-2: no. Of comparisons = n-2
iteration-3: no. Of comparisons = n-3
.
.
.
iteration-(n-1): no. Of comparisons = 1

total no. of comparisons = (n-1)+(n-2)+(n-3)+....+ 1
by arithmetic progression formula:
=> n(n-1) / 2
=> $(n^2 - n) / 2$
=> $O((n^2 - n) / 2)$
=> $O(n^2 - n)$ ...by neglecting divisive constant
=> $O(n^2)$ .... by using rule of polynomial only leading term is considered

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2
for itr=3 => pos=0,1
.
.
.

```
for( pos=0 ; pos < 6-1-itr ; pos++ )
```

10 20 30 40 50 60

iteration-1:

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

- if all pairs in an array are already inorder => there is no need of swapping => array is already sorted

- by basic algo/by design bubble is not efficient for already sorted input array, but it can be implemented efficiently by using logic of flag.
- best case occurs in bubble if array ele's are already sorted.
In this case only 1 iteration takes place and no. Of comparisons = n-1
T(n) = O(n-1)
T(n) = O(n) => Ω(1).

 # DS DAY-03:

3. Insertion Sort:
Best Case - if array is already sorted

Input Array : 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-5:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in best case in each iteration only 1 comparison takes place and in max (n-1) no. Of iterations array elements gets arranged in a sorted manner.
Total no. of comparisons = 1*(n-1) = n-1
T(n) = O(n-1) = O(n) = Ω(n).

4. Merge Sort:

by means of calculating mid pos big size array gets divided logically into two subarray's => left subarray and right subarray
left subarray => left to mid
right subarray => mid+1 to right

for left subarray => value of left remains same, and right = mid
for right subarray => value of right remains same, and left = mid+1

5. Quick Sort:
- we can apply partitioning on any partition only if size of an array/partition is greater than 1 i.e. only if left > right.

if(left == right) => partition contains only 1
if(left < right) => partition contains more 1 ele's
if(left > right) => partition is invalid

worst case occurs in quick sort if array is already sorted or array elements are exists exactly in a reverse order.

```
Pass-1:
[ 10 20 30 40 50 60 ]
[ LP ] 10 [ 20 30 40 50 60 ]

Pass-2:
[ 20 30 40 50 60 ]
[ LP ] 20 [ 30 40 50 60 ]

Pass-3:
[ 30 40 50 60 ]
[ LP ]  30 [ 40 50 60 ]

Pass-4:
[ 40 50 60 ]
[ LP ]  40 [ 50 60 ]

Pass-5:
[ 50 60 ]
[ LP ] 50 [ 60 ]
```