# Array Operations:

"Searching": to search/find/check a particular key ele in a given collection/list (either in an array or linked list or in other data sturcture as well) of elements.

- Two searching algorithms can be applied on an array:
1. Linear Search
2. Binary Search

1. Linear Search:
- in this algo, we start comparing key ele with first ele in a collection/list of ele's and we compare key ele with each ele in a collection/list ele's sequentially either till match is found or maximum till the last ele.

```
Algorithm LinearSearch(A, n, key)
{
        for( index = 1 ; index <= n ; index++ )
        {
                if( key == A[index] )
                        return true;
        }

        return false;
}
```

- In Linear Search:
* **"Best case":** occures if key is found at very first position, in this case only 1 comparison takes place --> running time of an algo in this case = O(1).
    Best case : Ω(1)

* **"Worst case":** occures either if key is found at last position or key does not exists, in this case "n" no. comparison takes place, wheras n is size of an array
--> running time of an algo in this case = O(n).
    Worst case: O(n)

* **"Average case":** occures if key is found at in between position, in this case
n/2 no. comparison takes place --> running time of an algo in this case = O(n/2).
    average case: θ(n)

# "Asymptotic Analysis": it is a "mathematical" way to calculate time complexity and space complexity of an algorithm without implementing it in any programming lanaguage.

Asymptotic Notations:
1. Big Oh(O): this notation can be used to represent worst case time complexity of an algo.
2. Big Omega( ): this notation can be used to represent best case time complexity of an algo
3. Big Theta ( ): this notation can be used to represent an average case time time complexity of an algo.

- if running time of an algo is having any additive/subtractive/multiplicative/divisive constant it can be neglected.
e.g.

    O(n+2) => O(n)
    O(n-3) => O(n)
    O(2*n) => O(n)
    O(n/2) => O(n)

2. Binary Search:
- this algo follows "divide-and-conquer" statergy
- if we want to apply binary search on array, prerequisite is that array ele's must be in sorted manner

```
Algorithm BinarySearch(A, key, n)//whereas A is an array of size "n"
{
    left=1
    right=n


    while( left <= right )//till subarray is valid
    {
        mid=(left+right)/2;

        if( key == A[mid] )
            return true;

        if( key < A[mid] )
                right = mid-1;
            else
                left = mid+1;
        }

        return false;
}
```

if is key is found in very first iteration at mid position -- best case
either key is found at leaf position or key does not exists -- worst case
if key is exists in between -- average case

- In Binary Search:
*** "Best case":** occures if key is found at mid pos in very first iteration, in this case only 1 comparison takes place --> running time of an algo in this case = O(1).

    Best case : Ω(1)

*** "Worst case":** running time of an algo in this case = O(log n).

    Worst case: O(log n)

*** "Average case": θ(log n)**

- while comparing two algo's, we always decides effieciency of an algo depends on worst case time complexity.
worst case time complexity of linear search = O(n)
worst case time complexity of binary search = O(log n)

  O(log n) < O(n)
  binary search algo is efficient than linear search
----

1. Selection Sort:
- in this algo, in the first iteration first position gets selected and ele which is at selected position gets compared with remaining all positions elements, if we found any ele smaller than element which is at selected pos then we will swap them with it, so in first iteration the smallest ele gets setteled/fixed at first position.
in second iteration, second position gets selected, and gets compared with remaining all its next positions ele's, if we found any ele smaller than element which is at selected pos then we will swap them with it, so in second iteration second smallest ele gets settled/fixed at second position, and so on.... in max (n-1) no. of iterations all array ele's gets arranged in a sorted manner.

```
Algorithm SelectionSort(A, n)
{
    for( sel_pos = 1 ; sel_pos < n ; sel_pos++ )//O(n)
    {
        for( pos = sel_pos + 1 ; pos <= n  ; pos++ )//O(n*n)
        {
            if( A[ sel_pos ] > A[ pos ] )
                SWAP(A[sel_pos], A[pos]);
        }
    }
}
```

Worst Case Time Complexity = $O(n^2)$.
Best Case Time Complexity = $\Omega(n^2)$.
Average Case Time Complexity = $\theta(n^2)$.