



# Arrays

# Arrays

- ❖ An array is defined as a set of finite number of homogeneous elements or same data items.
- ❖ It means an array can contain one type of data only, either all integer, all float-point number or all character.
- ❖ Simply, declaration of array is as follows:

```
int arr[10]
```

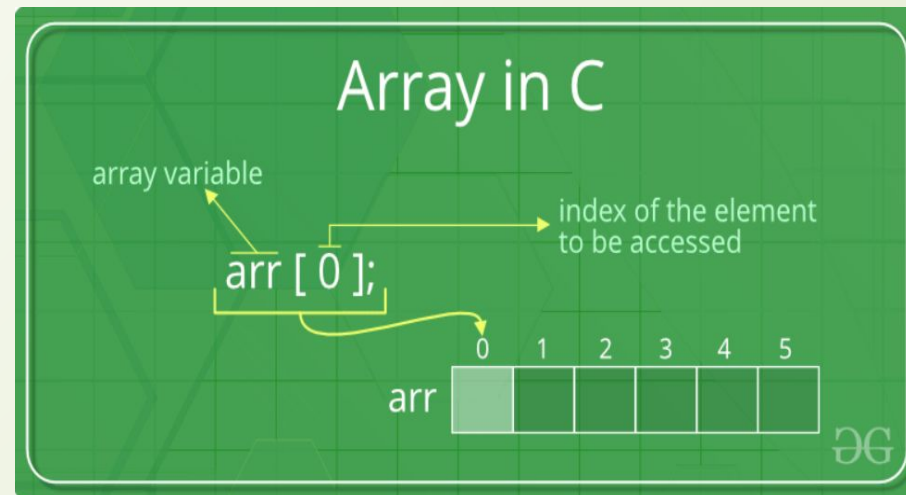
where *int* specifies the data type or type of elements arrays stores.

- ❖ “*arr*” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

# Arrays

❓ Following are some of the concepts to be remembered about arrays:

- ❓ The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
- ❓ The first element of the array has index zero[0]. It means the first element and last element will be specified as: arr[0] & arr[9] respectively.



- ❓ The elements of array will always be stored in the consecutive (continues) memory location.
- ❓ The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

$$(\text{Upperbound} - \text{lowerbound}) + 1$$

# Categories of Arrays

- ❖ The individual elements of an array are accessed using an index or indices to the array. Depending on the number of indices required to access an individual element of an array, arrays can be classified as:
- ❖ One-Dimensional Array or Linear Array – that requires only one index to access an individual element of an array.
- ❖ Two-Dimensional Arrays – that requires two indices to access an individual element of the array.
  - ❖ Equivalent to **Matrix** (Mathematics)
  - ❖ Equivalent to **Table** (Business)
- ❖ Multi-Dimensional Arrays – for which we need two or more indices

# Linear Arrays

- ❖ A Linear Array is a list of a finite number, say  $n$ , of homogeneous data elements such that:
- ❖ The elements of the array are referenced by an index set consisting of  $n$  consecutive integer numbers
- ❖ The elements of the array are stored in consecutive memory locations.
- ❖ The number  $n$  of the elements is called the size of the linear array. In general, if **lb** is the smallest index, called as Lower Bound and **ub** is the largest index, called as upper bound, then the size of the linear array is given as

$$\text{Size} = \text{ub} - \text{lb} + 1$$

# Two Dimensional Array

- ❖ A two dimensional array is a list of a finite number, say  $m*n$ , of homogeneous data elements such that
- ❖ The elements of the array are referenced by two index sets consisting of  $m$  and  $n$  consecutive integers.
- ❖ The elements of the array are stored in consecutive memory locations.
- ❖ The size of two dimensional array is denoted by  $m \times n$  and is pronounced as  $m$  by  $n$ .

# Representing Linear Array in Memory

- ❖ Let  $a$  be a linear array with  $n$  elements. As arrays are stored in consecutive memory location, the system need not keep track of the address of every element of  $a$ , but needs to keep track of first element only. The address of the first element is also known as the base address of the array and is denoted by

**$\text{base}(a)$**

- ❖ Using this base address, the computer computes the address of the  $k$ th element,  $\text{loc}(a[k])$ , using the formula

$$\text{loc}(a[k]) = \text{base}(a) + w(k-1)$$

where  $w$  is the number of bytes per storage location for one element of the array.

- ❖ It is clear from the formula, that the time taken to compute  $\text{loc}(a[k])$  is essentially the same for any value of  $k$ .
- ❖ Hence, Computer takes same time to access any element of the array.

# Operations on Linear Arrays

❖ Some common operation performed on array are:

- ❖ Creation of an array
- ❖ Traversing an array
- ❖ Insertion of new element
- ❖ Deletion of required element
- ❖ Modification of an element
- ❖ Merging of arrays



# Traversal Operation

- ❖ Traversing is the process of visiting each element of the array exactly once.
- ❖ Traversal is done by starting with the first element of the array and reaching to the last.
- ❖ Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A with given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of A exactly once.
- ❖ Consider the following array with LB = 0 and UB = 9

0	1	2	3	4	5	6	7	8	9	Indices of elements
5	20	60	10	77	29	30	2	11	87	Array Elements

Array of 10 integer elements

# Algorithm- Traversal in a Linear Array

❖ To apply process on this array of ten elements a counter variable is needed to store the address of current element. After processing each element the counter is incremented and same process is applied to the next element. This is repeated until the counter reaches the last index of the array.

## ❖ **TraverseLinearArray (a, n)**

Here  $a$  is a linear array of size  $n$ . This algorithm traverses the array and applies the operation process to each element of the array.

1. Set  $i = 0$  // Initialize Counter
2. Repeat steps 3 and 4 while  $i \leq (n-1)$
3. Apply process ( $a[i]$ ) // visit element
4. Set  $i = i + 1$  // increment counter
- Endwhile
5. Exit

# Complexity of Traversal in a Linear Array

Traversal operation results in visiting every element of the linear array once. In the algorithm the following is the way the steps are counted.

1. Step 1 is executed once, so it contributes 1 to complexity function  $f(n)$
2. Step 2 is a loop control step that executes step 3 and step 4  $n$  times (once for each element of array  $a$  having  $n$  elements)

So the  $f(n)$  can be defined as

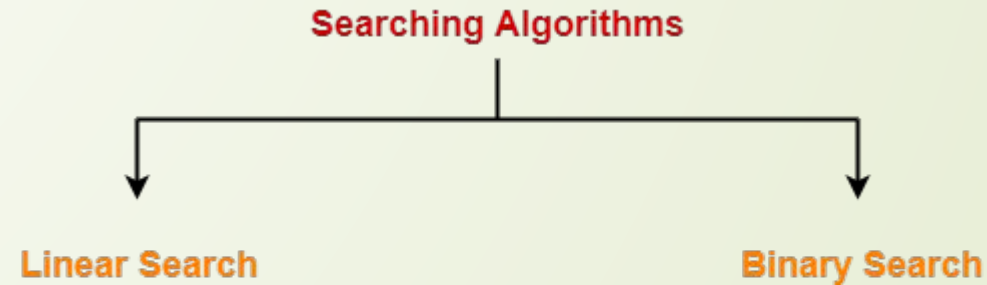
$$f(n) = 2*n + 1 \text{ or}$$

$$f(n) = 2n + 1$$

The highest order term in this function defined in the terms of input size of the algorithm is  $n$ , so we can say that the complexity of this algorithm is  **$O(n)$** . It also means that the time of traversal of linear array grows linearly with the size of array DATA.

# Search Operation

- ❖ Searching is the process of finding the location of given element in the linear array. The Search is said to be successful if the given element is found i.e. the element does exist in the array; otherwise unsuccessful.
- ❖ The searching of an element in the given array may be carried out in the following two ways-



**Linear Search**

**Binary Search**

- ❖ The algorithm that one chooses generally depends on organization of the array elements.
- ❖ If the elements are in random order, then linear search technique is used and if the array elements are sorted, then it is preferable to use binary Search

# Linear Search

## What is a Linear Search?

- ❖ A linear search, also known as a sequential search, is a method of finding an element within a list. Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

## A simple approach to implement a linear search is:

- Begin with the leftmost element of `arr[]` and one by one compare `x` with each element.
- If `x` matches with an element then return the index.
- If `x` does not match with any of the elements then return -1.

# Algorithm: Linear Search

## ❖ LinearSearch ( a, n, item, loc)

Here  $a$  is a linear array of size  $n$ . This algorithm finds the location,  $loc$ , of the element  $item$  in linear array  $a$ . If search ends in success it sets  $loc$  to the index of the element; otherwise it sets  $loc$  to -1.

1. Set  $i = 0$  // Initialize Counter
2. Repeat steps 3 and 4 while ( $i \leq (n-1)$ )
3.   If ( $a[i] = item$ ) then  
        Set  $loc = i$       // element found at location at  $i$   
        Exit  
    EndIf
4.   Set  $i = i + 1$  // increment counter  
    EndWhile
5. Set  $Loc = -1$  //element not found
6. Exit



# Linear Search


## Advantages

- ❖ The linear Search is simple – it is easy to understand and implement
- ❖ It does not require the data in the array to be stored in any particular order.

## Disadvantages

- ❖ It has poor efficiency because it takes lots of comparisons to find a particular record in big files.
- ❖ The performance of the algorithm scales linearly with the size of the input
- ❖ Linear search is slower than other searching algorithms.

# Example: Linear Search



5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30



# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30

# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30

# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30

# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30

# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30

# Example: Linear Search

5	20	60	10	77	29	30	2	11	87
---	----	----	----	----	----	----	---	----	----

↑  
Element

Searching item = 30 Found!!

# Performance Analysis of Linear Search

## ❖ **Best Case:**

the item may occur at first position. In this case the search operation terminates in success with just one comparison.

## ❖ **Worst case:**

either the item is present at the last position or it is not present in the list. In former case the search terminates with the success with  $n$  comparisons. However in the later case, the search terminates in a failure with  $n$  comparisons.

❖ Thus, in worst case the complexity of linear search is  **$O(n)$** .

# Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.
- Binary Search Algorithm can be applied only on **Sorted arrays**.
- So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
- To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.



# Binary Search Algorithm

- Binary Search Algorithm searches an element by comparing it with the middle most element of the array. Then, following three cases are possible-

- **Case-01**

If the element being searched is found to be the middle most element, its index is returned.

- **Case-02**

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

- **Case-03**

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

- This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

## BinarySearch (a, n, item, loc)

Here, a is a linear array of size n. This algorithm finds the location of the element item in sorted linear array a. If search ends in success it sets loc to the index of the element; otherwise it sets loc to -1. Here, variables beg and end keep track of the first element and the last element of the array to be searched and variable mid is used as index of the middle element of the array under consideration.

1. Begin
2. Set beg = 0
3. Set end = n-1
4. Set mid = (beg + end) / 2
5. Repeat steps 6 and 7  
    while ( (beg <= end) and (a[mid] ≠ item) )
6. If (item < a[mid]) then  
    Set end = mid - 1  
Else  
    Set beg = mid + 1  
Endif
7. Set mid = (beg + end) / 2  
    Endwhile
8. If (beg > end) then  
    Set loc = -1      // element not found  
Else  
    Set loc = mid      // element found at location mid  
Endif
9. Exit

# Example:

- ❖ Consider the following sorted linear array.
  - Element 15 has to be searched in it using Binary Search Algorithm.

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Binary Search Example

❖ Binary Search Algorithm works in the following steps-

❖ **Step-01:**

- To begin with, we take  $beg=0$  and  $end=6$ . We compute location of the middle element as-  
$$mid = (beg + end) / 2 = (0 + 6) / 2 = 3$$
- Here,  $a[mid] = a[3] = 20 \neq 15$  and  $beg < end$ . So, we start next iteration.

❖ **Step-02:**

- Since  $a[mid] = 20 > 15$ , so we take  $end = mid - 1 = 3 - 1 = 2$  whereas  $beg$  remains unchanged. We compute location of the middle element as-  
$$mid = (beg + end) / 2 = (0 + 2) / 2 = 1$$
- Here,  $a[mid] = a[1] = 10 \neq 15$  and  $beg < end$ . So, we start next iteration.

❖ **Step-03:**

- Since  $a[mid] = 10 < 15$ , so we take  $beg = mid + 1 = 1 + 1 = 2$  whereas  $end$  remains unchanged. We compute location of the middle element as-  
$$mid = (beg + end) / 2 = (2 + 2) / 2 = 2$$
- Here,  $a[mid] = a[2] = 15$  which matches to the element being searched. So, our search terminates in success and index 2 is returned.

# Time Complexity Analysis

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for  $n$  elements in the array, there are  $\log_2 n$  iterations or recursive calls.

❖ Thus,

❖ **Time Complexity of Binary Search Algorithm is  $O(\log_2 n)$ .**

❖ Here,  $n$  is the number of elements in the sorted linear array.

❖ This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

# Binary Search Algorithm Advantages

- ◆ The advantages of binary search algorithm are-
  - It eliminates half of the list from further searching by using the result of each comparison.
  - It indicates whether the element being searched is before or after the current position in the list.
  - This information is used to narrow the search.
  - For large lists of data, it works significantly better than linear search.

# Binary Search Algorithm Disadvantages

- ❖ The disadvantages of binary search algorithm are-
  - It employs recursive approach which requires more stack space.
  - Programming binary search algorithm is error prone and difficult.
  - The interaction of binary search with memory hierarchy i.e. caching is poor. (because of its random access nature)

# Insertion Operation

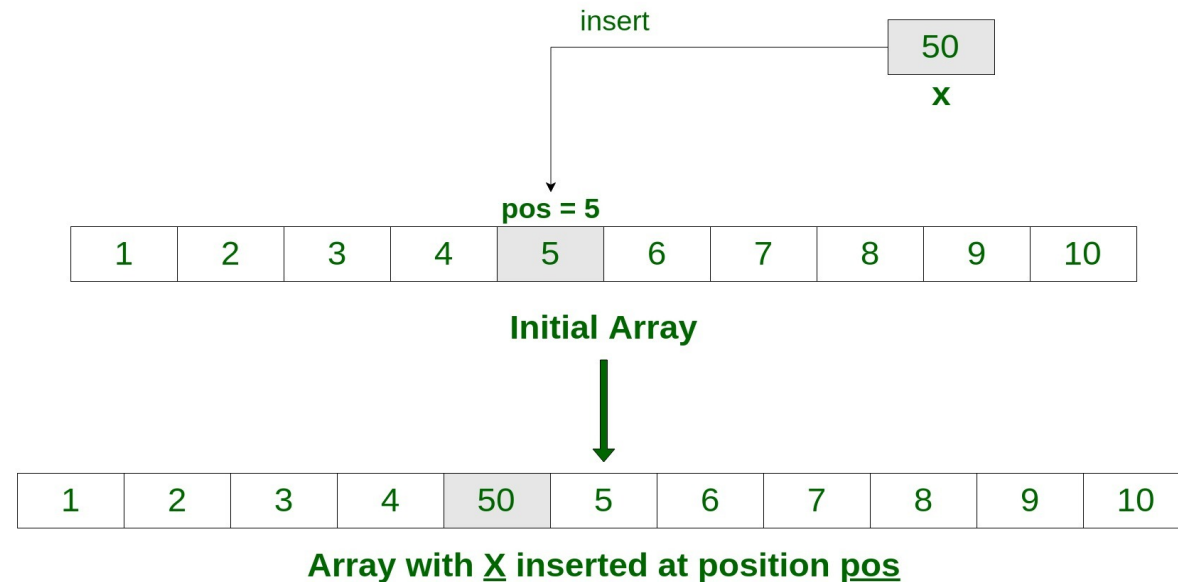
- ❖ Insertion refers to the operation of adding an element to existing list of elements.
- ❖ After insertion the size of the linear array is increased by factor of one.
- ❖ Insertion is possible only if memory space allocated is large enough to accommodate the additional element.
- ❖ Inserting an element at the end of the array can be done very easily. However, to insert an element at any other location, the elements are to be moved downward to new locations to accommodate the new element and keep the order of other elements.



# How to insert an element in an array

Given an array **arr** of size **n**, this article tells how to insert an element **x** in this array **arr** at a specific position **pos**.

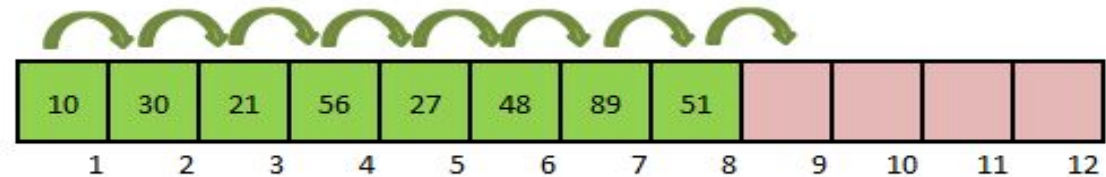
## Insert an element at a specific position in an Array.



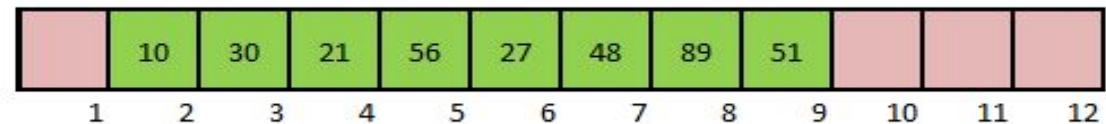
# Insertion at the Beginning of the array

- ❖ In this case we have to move all the elements one position backwards to make a hole at the beginning of array. Though the insertion process is not difficult but freeing the first location for new element involves movement of all the existing elements. This is the worst case scenario in insertion in a linear array.
- ❖ In the below example, array elements from index 1 to index 8 have to be moved one position backwards so that the new element 44 can be stored at index 1

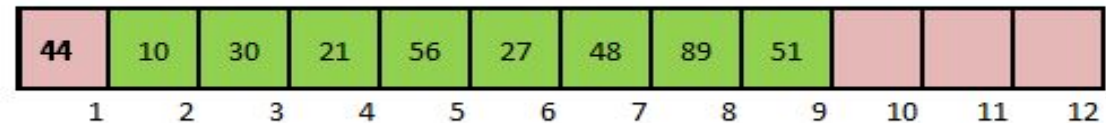
❖ Initially



❖ After the backward movement

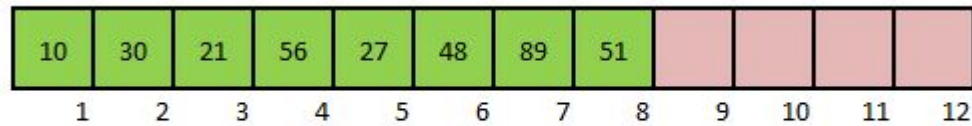


❖ After Insertion



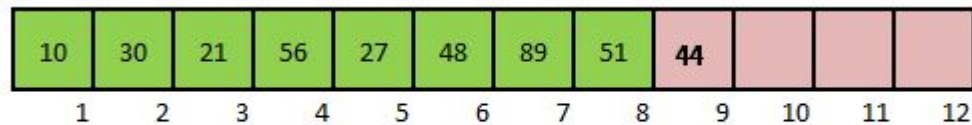
# Insertion at the end of the array

- ? In this case we don't have to move any elements since the action here will be just to append the element at the location after the last element. This is the best case scenario.
- ? In the example array no elements are moved. The new element is stored in index 9.
- ? Initially



10	30	21	56	27	48	89	51				
1	2	3	4	5	6	7	8	9	10	11	12

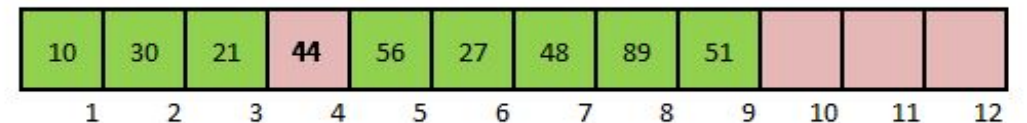
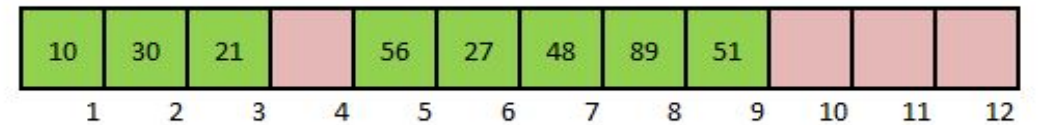
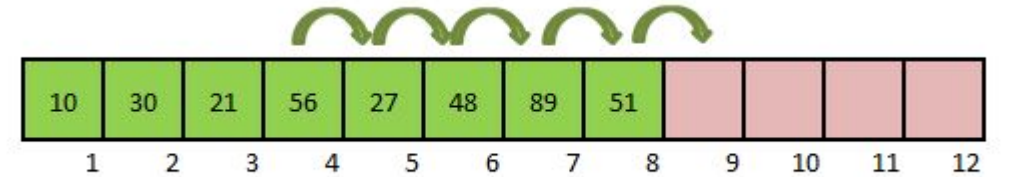
- ? After Insertion



10	30	21	56	27	48	89	51	44			
1	2	3	4	5	6	7	8	9	10	11	12

# Insertion at the give position J

- ❖ Let J be any location in the array of one element already existing. We have to add the new element at J position. To do this, starting from J, every element is moved one place backwards so that a hole is created at J and new element can be inserted here. This is the average case scenario.
- ❖ In the example array ,elements from index J (4) to index 8 have to moved one position backwards so that a new element can be stored at index J(4)
- ❖ Initially
- ❖ After Movement of Elements
- ❖ After Insertion



# Performance Analysis

- ❖ The best possible case occurs when the item is inserted at the last position. In this case, no element is moved down.
- ❖ The worst case occurs when the element is to be inserted at the first position . In this case, all the elements (n in number) are moved down. Therefore, in worst case, n number of movements will occur.
- ❖ Hence, Complexity of insertion operation is  **$O(n)$** .

# Deletion in Array

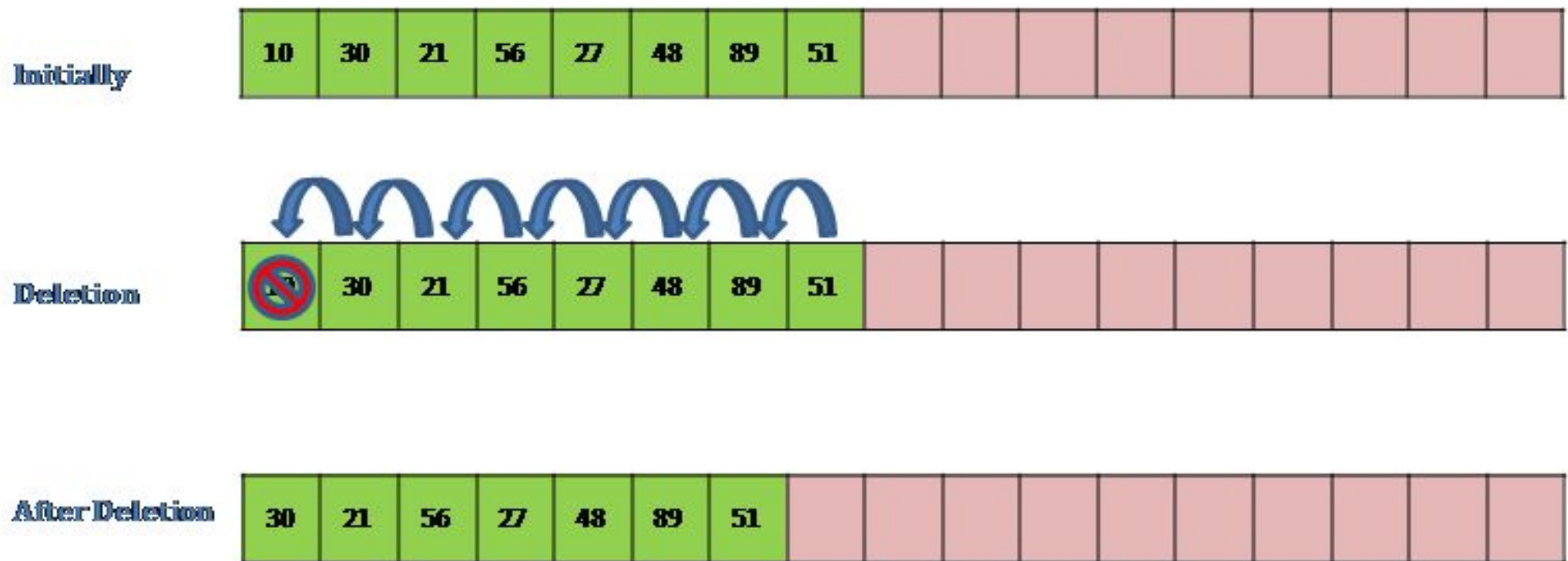
- ❖ Deletion in array means removing an element and replacing it with the next element or element present at next index.
- ❖ After deletion the size of the linear array is decreased by factor of one.
- ❖ It involves three cases:

**At Beginning,  
Given Location and  
End of Linear Array**



# Deletion at the beginning of the array

- ❓ In this case we have to move all the elements one position forward to fill the position of the element at the beginning of array. Though the deletion process is not difficult but moving all elements one position forward involve movement of all the existing elements except the one being deleted. This is the worst case scenario in deletion in a linear array.
- ❓ In the example array elements from index 1 to index 8 have to moved one position forwards so that the first element is replaced by second, second by third and so on









# Performance Analysis

- ❖ The best possible case occurs when the item is deleted from the last position. In this case, no element is moved up.
- ❖ The worst case occurs when the element is deleted from the first position. In this case, almost all the elements ( $n-1$  in number) are moved up. Therefore  $n-1$  movements will occur.
- ❖ Hence, the complexity of Deletion operation is  **$O(n)$**

# Sort Operation

- ❖ Sorting is the process of arranging the elements of the array in some logical order. The logical order may be ascending or descending in case of numeric values or dictionary order in case of alphanumeric values.
- ❖ Most common sorting algorithm is **Bubble sort**.

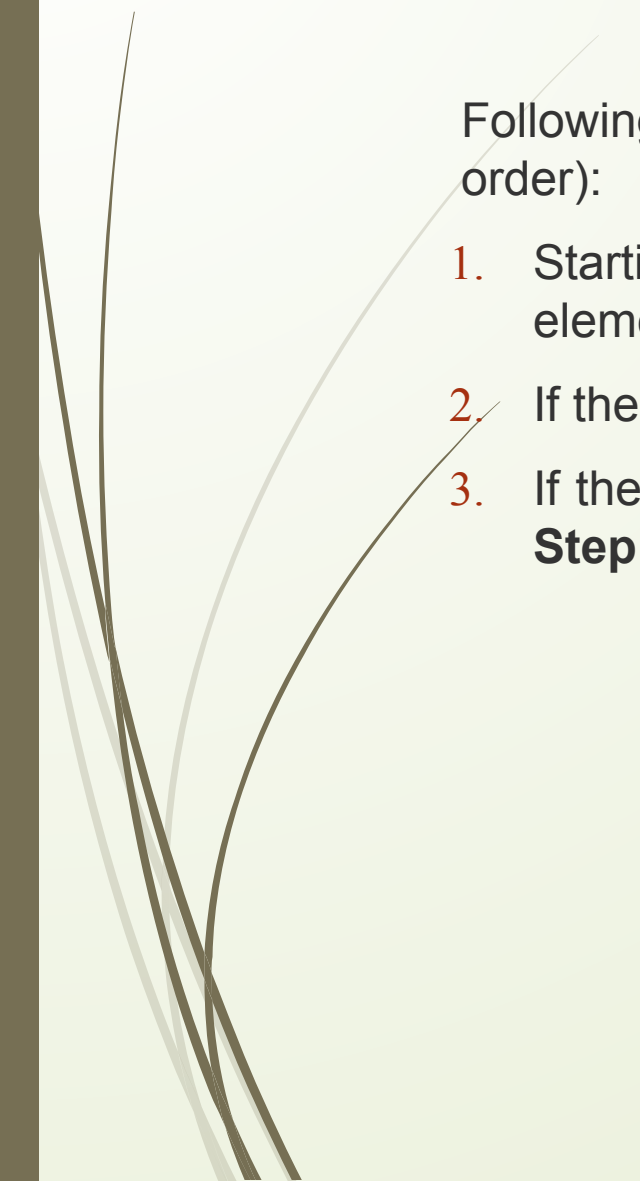
# Bubble Sort

- ❑ **Bubble Sort** is a simple algorithm which is used to sort a given set of  $N$  elements provided in form of an array with  $N$  number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
- ❑ If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.
- ❑ If we have total  $N$  elements, then we need to repeat this process for  $N-1$  times.
- ❑ It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.
- ❑ Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.



# Implementing Bubble Sort Algorithm

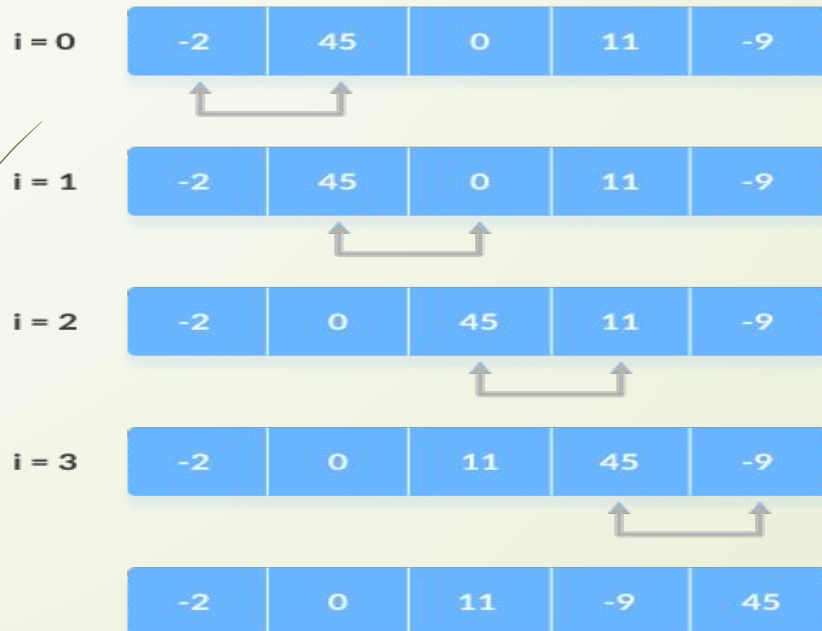
Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element (index = 0), compare the current element with the next element of the array.
  2. If the current element is greater than the next element of the array, swap them.
  3. If the current element is less than the next element, move to the next element. **Repeat Step 1.**
- 

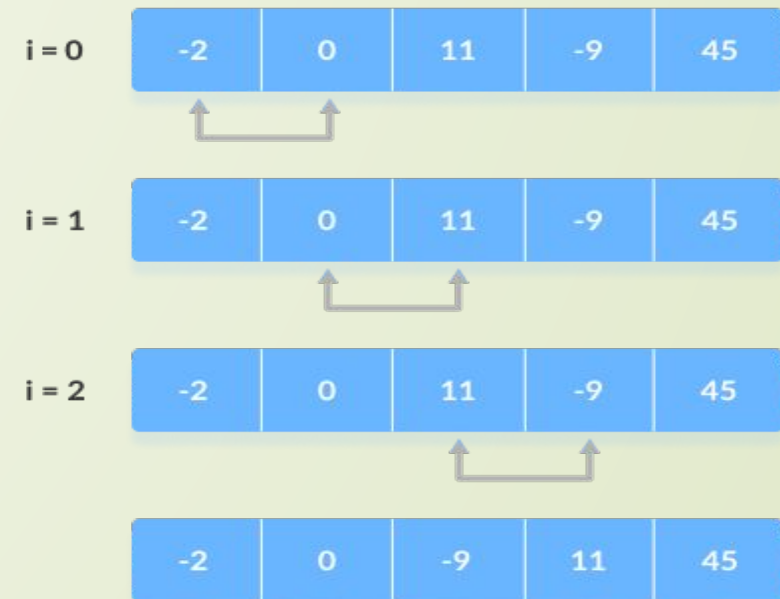
# Example:

Consider an Array of 5 elements: -2, 45, 0, 11, -9

step = 0



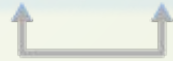
step = 1



step = 2

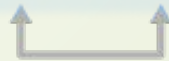
i = 0

-2	0	-9	11	45
----	---	----	----	----



i = 1

-2	0	-9	11	45
----	---	----	----	----

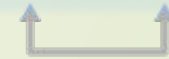


-2	-9	0	11	45
----	----	---	----	----

step = 3

i = 0

-2	-9	0	11	45
----	----	---	----	----



-9	-2	0	11	45
----	----	---	----	----

# Algorithm Bubble Sort

## BubbleSort ( a, n)

Here **a** is a linear array with **n** elements in memory. This algorithm sorts the elements in the ascending order. It uses a temporary variable **temp** to facilitate the exchange of two values and variable **i** is used as loop control variable.

1. Repeat steps 2 and 3 For k = 1 to (n-1) in steps of +1
2. Repeat step 3 For j = 0 to (n-k-1) in steps of +1
3. If (a[j] > a[j+1] ) then
  - Set temp = a[j]
  - Set a[j] = a[j+1]
  - Set a[j+1] = tempEndIf
- EndFor
- EndFor
4. Exit



# Performance Analysis

- ❖ In Bubble sort, array will be sorted after  $(n-1)$  passes.
- ❖ The first pass requires  $(n-1)$  comparisons, the second pass requires  $(n-2)$ , .....,  $k$ th pass requires  $(n-k)$  comparisons. Therefore, total comparisons are:

$$\begin{aligned}f(n) &= (n-1) + (n-2) + (n-3) + \dots + (n-k) + \dots + 3 + 2 + 1 \\&= n(n-1)/2 \\&= O(n^2)\end{aligned}$$