Searching

Searching technique refers to finding a key element among the list of elements.

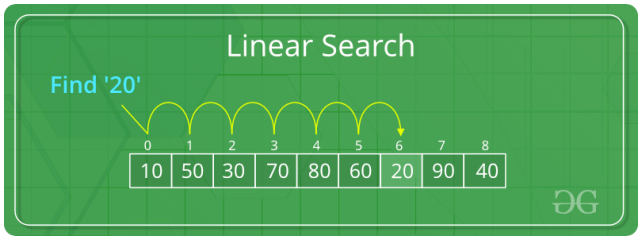
- If the given element is present in the list, then the searching process is said to be successful.
- If the given element is not present in the list, then the searching process is said to be unsuccessful.

C language provides two types of searching techniques. They are as follows –

- Linear search
- Binary search

Linear Search(Sequential Search)

- **Linear Search** is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.
- Searching for the key element is done in a linear fashion.
- It is the simplest searching technique.
- It does not expect the list to be sorted.
- Limitation − It consumes more time and reduce the power of system.



Linear Search Algorithm

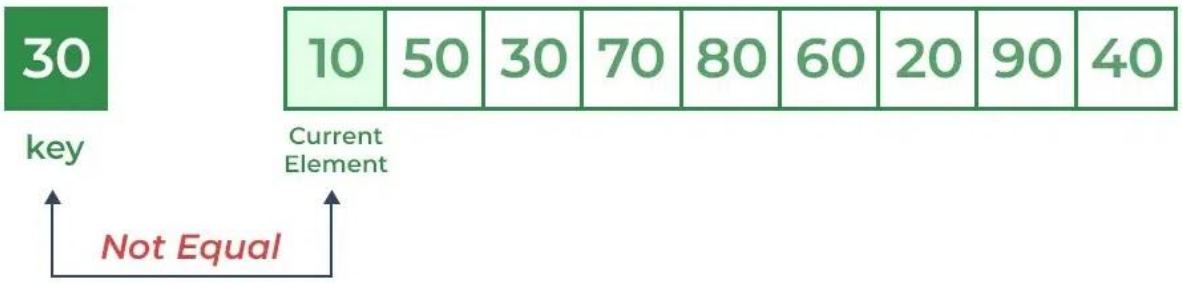
How Does Linear Search Algorithm Work?

In Linear Search Algorithm,

- Every element is considered as a potential match for the key and checked for the same.
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields “No match found”.

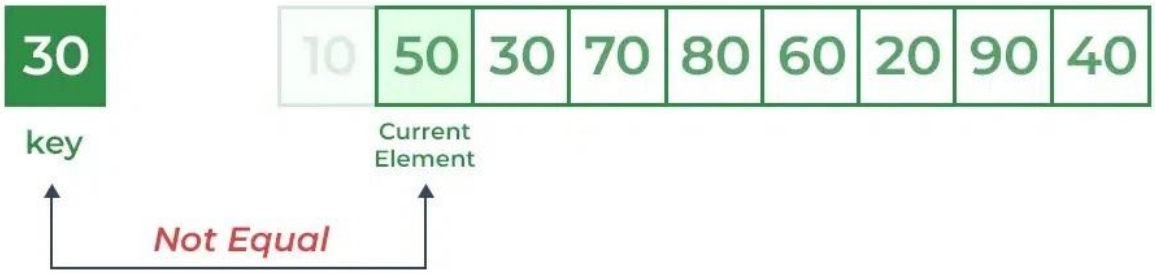
For example: Consider the array arr[] = {10, 50, 30, 70, 80, 20, 90, 40} and key = 30

Step 1: Start from the first element (index 0) and compare key with each element (arr[i]). Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.



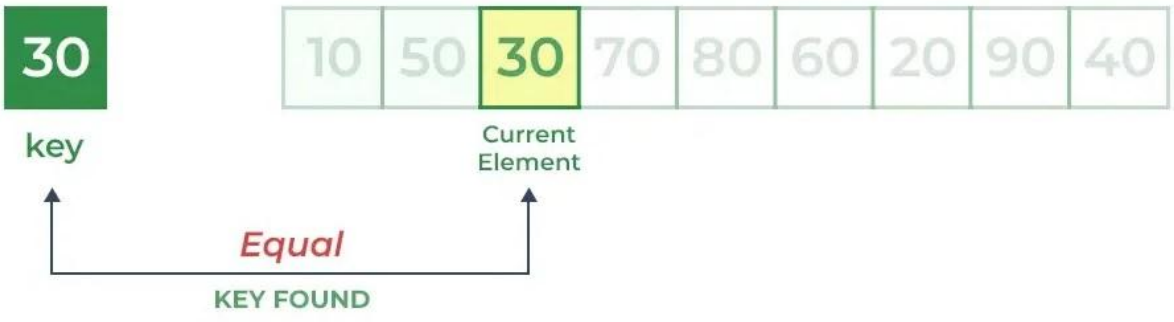
Compare key with arr[0]

Step 2: Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.



Compare key with arr[1]

Step 3: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



Compare key with arr[2]

Program:

```
#include<conio.h>
#include<stdio.h>
void main()
{
int a[50],i,n,key,found=0;
clrscr();
printf("Enter the size of an array:");
scanf("%d",&n);
printf("Enter The Elements into an arrays:");
for(i=0;i<n;i++){
scanf("%d",&a[i]);
}
printf("Enter The Value to be search:");
scanf("%d",&key);
for(i=0;i<=n-1;i++)
{
if(a[i]==key)
{
printf("After Linear Search The Key found at location%d",i);
found=1;
break;
}
}
if(!found){
printf("key not found");
}
getch();

}
```

Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

Time Complexity:

- Best Case: In the best case, the key might be present at the first index. So the best case complexity is O(1)
- Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- Average Case: O(N)

Binary Search

Binary Search is defined as a searching algorithm used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Conditions for when to apply Binary Search in a Data Structure:
To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

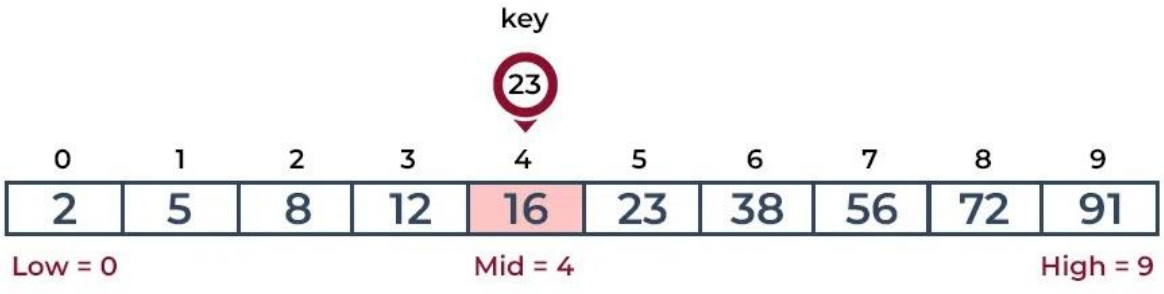
In this algorithm,

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
    - If the key is smaller than the middle element, then the left side is used for next search.
    - If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

How Binary Search work?

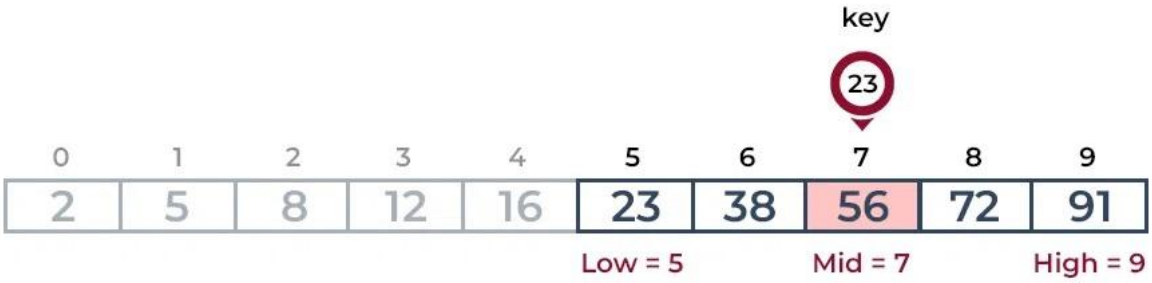
- Consider an array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the target = 23.
- First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

- Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



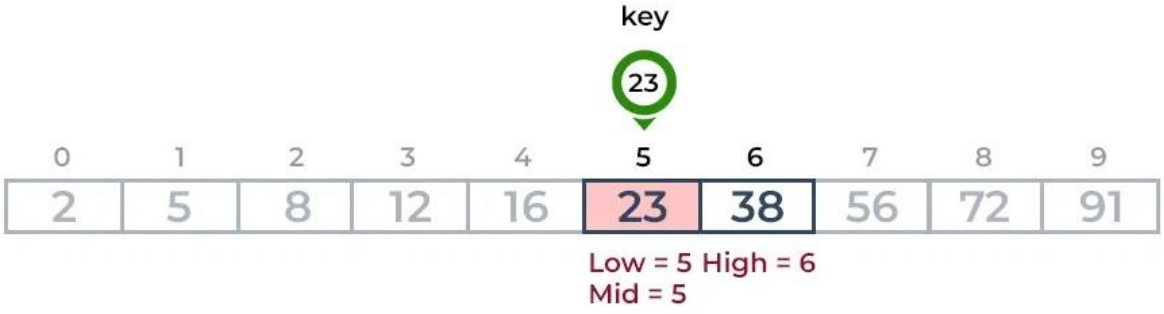
Binary Search Algorithm : Compare key with 16

- *Key is less than the current mid 56. The search space moves to the left.*



Binary Search Algorithm : Compare key with 56

Second Step: *If the key matches the value of the mid element, the element is found and stop search.*



Binary Search Algorithm : Key matches with mid

Program

```
#include<conio.h>
#include<stdio.h>
void main()
{
int a[50],i,n,low,high,mid,key,found=0;
clrscr();
printf("Enter the size of an array:");
scanf("%d",&n);
printf("Enter The Elements into an arrays in assending order:");
for(i=0;i<n;i++){
scanf("%d",&a[i]);
}
printf("Enter The Value to be search:");
scanf("%d",&key);
low=0;
high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(key<a[mid])
{
high=mid-1;
}
else if(key>a[mid])
```

```
{
low=mid+1;
}
else if(key==a[mid])
{
printf("After Binary Search The Key found at location %d",mid);
found=1;
break;
}
} if(!found)
{
printf("After Binary Search The Key is not found:");
}
getch();
}
```

Sorting

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

- Bubble sort (or) Exchange Sort.
- Selection sort.
- Insertion sort (or) Linear sort.
- Quick sort (or) Partition exchange sort.
- Merge Sort (or) External sort.

Table of Time Complexity

Name	Best Case	Average Case	Worst Case	Method Used
Bubble Sort	n	n^2	n^2	Exchanging
Selection Sort	n^2	n^2	n^2	Selection
Insertion Sort	n	n^2	n^2	Insertion

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

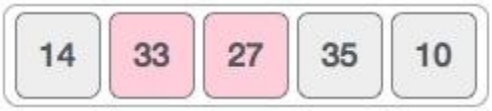
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

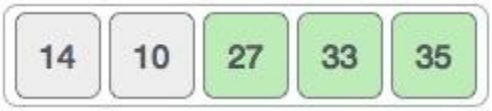
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

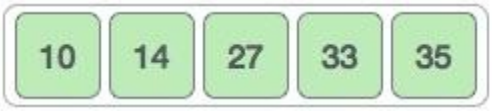
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Program

```
#include<stdio.h>

#include<conio.h>

void main()

{
```

```
int a[50],i,n,j,temp;
clrscr();
printf("enter the size of an array");
scanf("%d",&n);
printf("enter the values of array");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("The array Before bubble sorting:");
for(i=0;i<=n-1;i++)
{
printf("%d\n",a[i]);
}
for(i=0;i<=n;i++)
{
for(j=0;j<=n-1-i;j++)
{
if(a[j]>a[j+i])
{
temp=a[j];
a[j]=a[j+i];
a[j+i]=temp;
```

```
}}}

printf("The Array After Bubble Sorting:");

for(i=0;i<n;i++)

{

printf("%d\n",a[i]);

}

getch();

}
```

The **selection sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted subarray and putting it at the beginning of the sorted subarray. In this article, we will learn to write a C program to implement selection sort.

Approach

- Initialize minimum value **min_id x** to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than **min_id x** is found then swap both values.
- Then, increment **min_id x** to point to the next element.
- Repeat until the array is sorted.

Selection Sort Algorithm in C

```
selectionSort(array, size)
  loop i from 0 to size - 2
    set minIndex as i
    loop j from first unsorted to size - 1
      check if array[j] < array[minIndex]
        set minIndex as j
    swap array[i] with array[minIndex]
  end for
end selectionSort
```

Working of Selection Sort in C

In each pass, the minimum element is identified from the unsorted part of the array and placed at the beginning of the sorted part of the array. The sorted portion of the array gradually grows with each pass until the entire array becomes sorted.

Let’s take the example of the above array and see what happens in each pass using the below

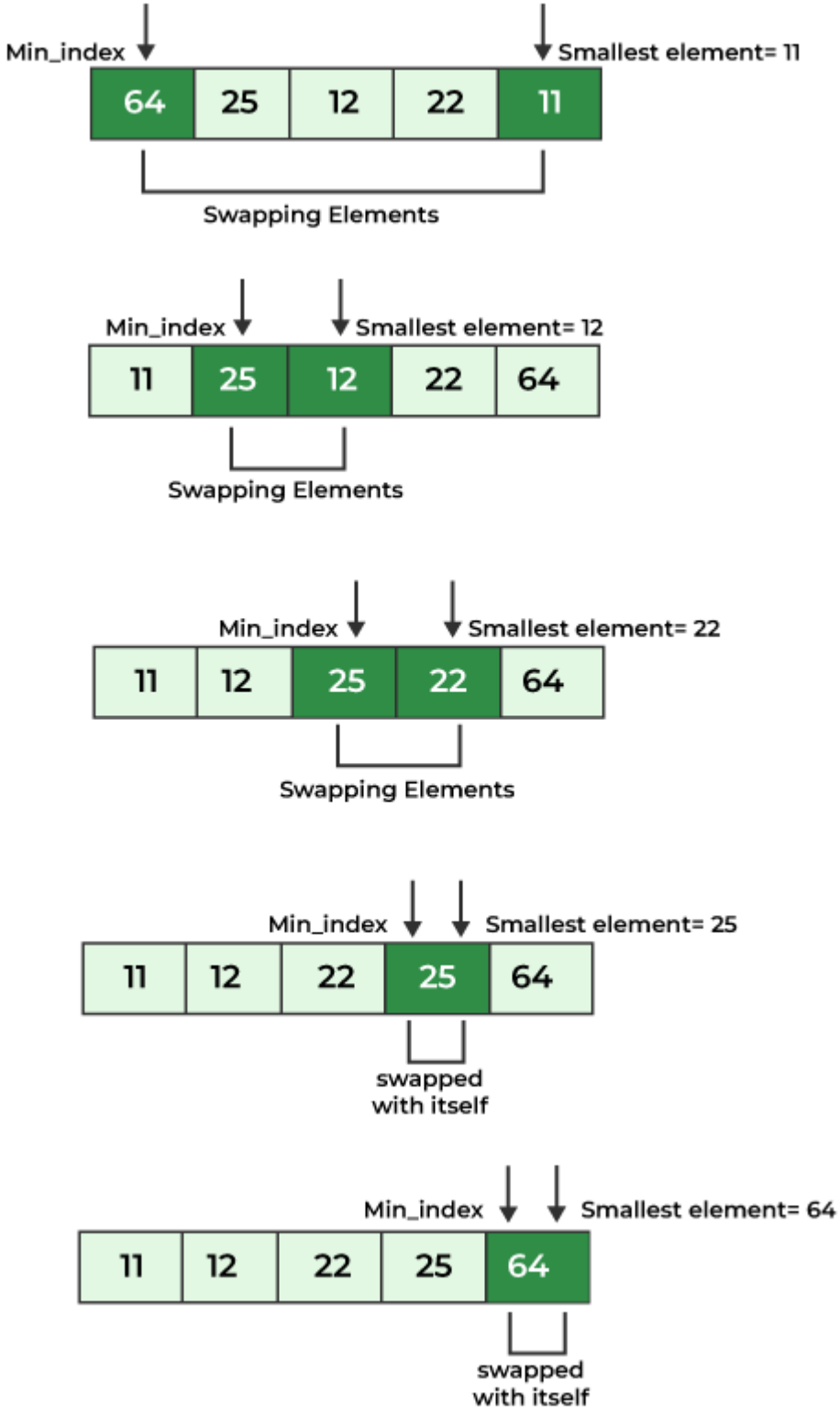


illustration.
The resultant array is the sorted array.

Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[50],i,j,min,n,temp;
clrscr();
printf("enter the size of an array");
scanf("%d",&n);
printf("enter the values of array");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("The array Before Selection sorting:");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
for(i=0;i<n-1;i++)
{
min=i;
for(j=i+1;j<n;j++)
{
if(a[j]<a[min])
{
min=j;
}
}
temp=a[i];
a[i]=a[min];
a[min]=temp;
}
printf("The Array After Selection Sorting:");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
getch();
}
```

Insertion sort is an algorithm used to sort a collection of elements in ascending or descending order. The basic idea behind the algorithm is to divide the list into two parts: a sorted part and an unsorted part.

Initially, the sorted part contains only the first element of the list, while the rest of the list is in the unsorted part. The algorithm then iterates through each element in the unsorted part, picking one at a time, and inserts it into its correct position in the sorted part.

To do this, the algorithm compares the current element with each element in the sorted part, starting from the rightmost element. It continues to move to the left until it finds an element that is smaller (if sorting in ascending order) or larger (if sorting in descending order) than the current element.

Once the correct position has been found, the algorithm shifts all the elements to the right of that position to make room for the current element, and then inserts the current element into its correct position.

This process continues until all the elements in the unsorted part have been inserted into their correct positions in the sorted part, resulting in a fully sorted list.

One of the advantages of insertion sort is that it is an in-place sorting algorithm, which means that it does not require any additional storage space other than the original list. Additionally, it has a time complexity of O(n^2), which makes it suitable for small datasets, but not for large ones.

Overall, insertion sort is a simple, yet effective sorting algorithm that can be used for small datasets or as a part of more complex algorithms.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Insertion Sort Algorithm

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[N] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12	**11**	**13**	**5**	**6**

First Pass:

- *Initially, the first two elements of the array are compared in insertion sort.*

12	**11**	13	5	6

- *Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.*
- *So, for now 11 is stored in a sorted sub-array.*

11	**12**	13	5	6

Second Pass:

- *Now, move to the next two elements and compare them*

11	**12**	**13**	5	6

- *Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11*

Third Pass:

- *Now, two elements are present in the sorted sub-array which are **11** and **12***
- *Moving forward to the next two elements which are 13 and 5*

11	12	**13**	**5**	6

- *Both 5 and 13 are not present at their correct place so swap them*

11	12	**5**	**13**	6

- *After swapping, elements 12 and 5 are not sorted, thus swap again*

11	**5**	**12**	13	6

- *Here, again 11 and 5 are not sorted, hence swap again*

5	**11**	12	13	6

- *here, it is at its correct position*

Fourth Pass:

- *Now, the elements which are present in the sorted sub-array are **5, 11** and **12***
- *Moving to the next two elements 13 and 6*

5	11	12	**13**	**6**

- *Clearly, they are not sorted, thus perform swap between both*

5	11	12	**6**	**13**

- *Now, 6 is smaller than 12, hence, swap again*

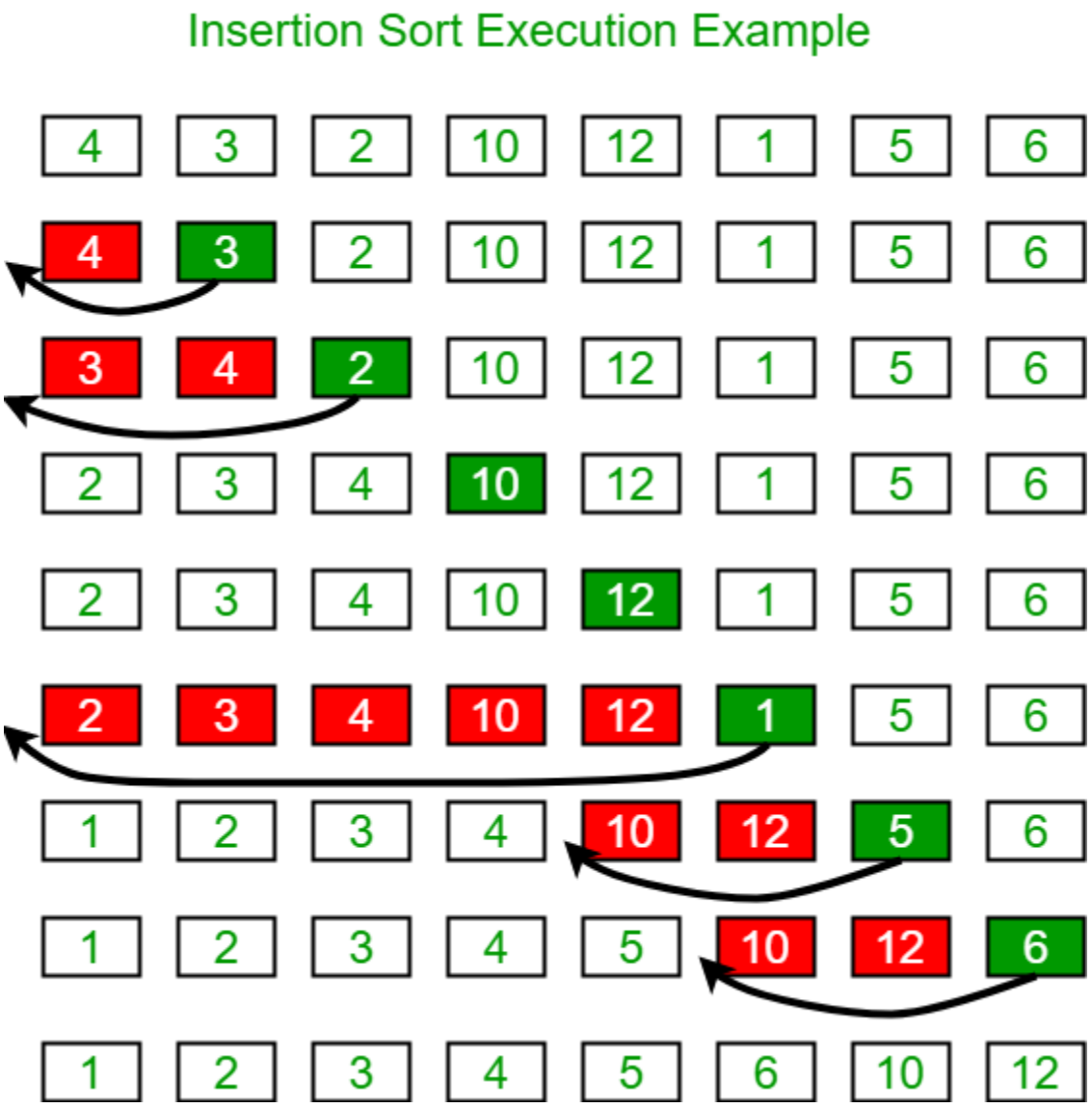
5	11	**6**	**12**	13

- *Here, also swapping makes 11 and 6 unsorted hence, swap again*

5	**6**	**11**	12	13

- *Finally, the array is completely sorted.*

Illustrations:



Program

```
#include<stdio.h>

#include<conio.h>

void main()

{

int a[50],i,n,j,temp;

clrscr();
```

```
printf("enter the size of an array");
scanf("%d",&n);
printf("enter the values of array");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("The array Before insertion sorting:");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
for(i=1;i<n;i++)
{
temp=a[i];
j=i-1;
while(j>=0 && a[j]>temp)
{
a[j+1]=a[j];
j=j-1;
}
a[j+1]=temp;
  }

printf("The Array After insertion Sorting:");
```

```
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
getch();
}
```