# UNIT-VI

## Sorting Techniques; Insertion sort, selection sort, merge sort, heap sort, searching Techniques: linear search, binary search and hashing

# UNIT- VI

## Searching:

Searching in data structure refers to the process of finding location of an element in a list. This is one of the important parts of many data structures algorithms, as one operation can be performed on an element if and only if we find it. Various algorithms have been defined to find whether an element is present in the collection of items or not. The efficiency of searching an element increases the efficiency of any algorithm.

### Searching Techniques

To search an element in a given array, it can be done in following ways:

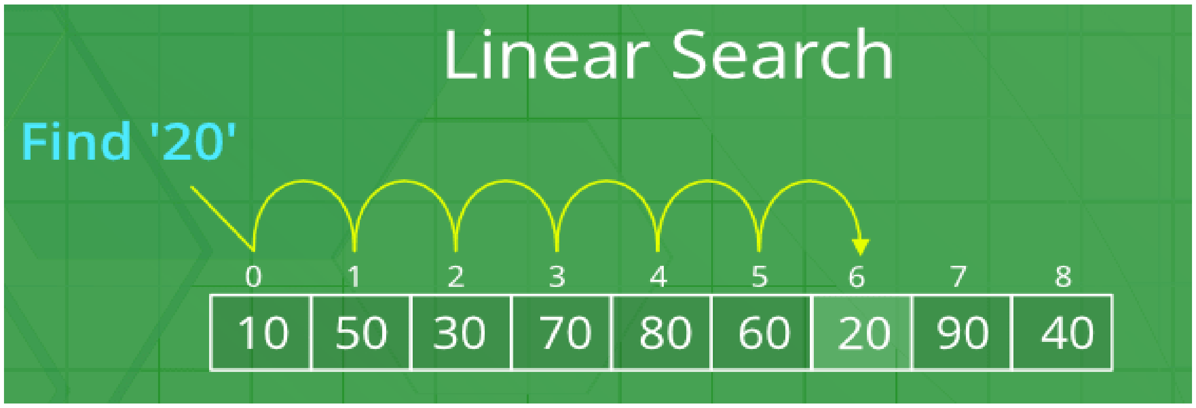
1. Linear Search
2. Binary Search

### Linear Search:

Linear search in C is to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

Following are the steps of implementation that we will be following:

1. Traverse the array using a loop.
2. In every iteration, compare the target value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return.



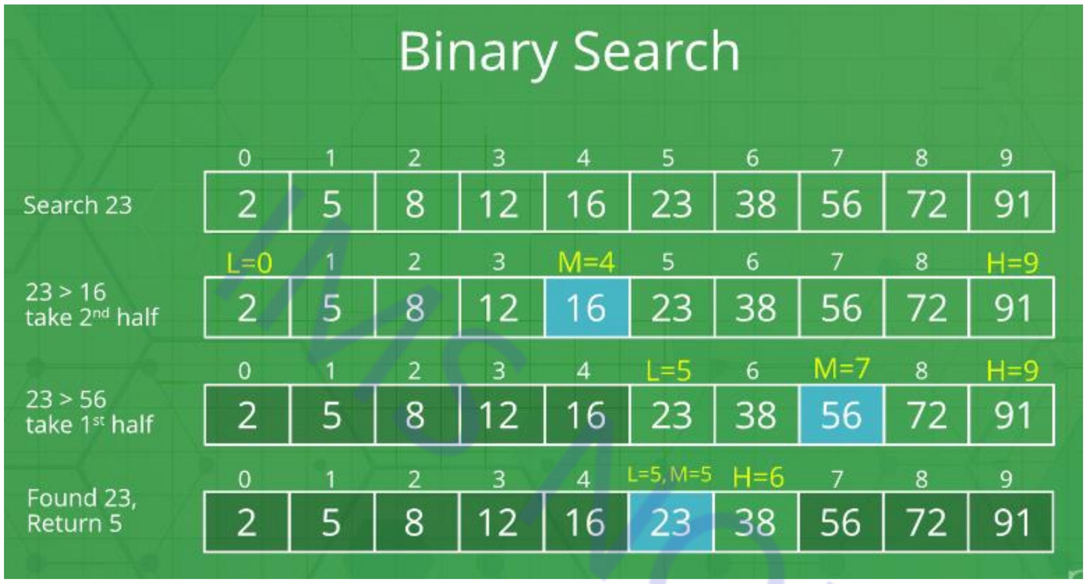
*The time complexity of above algorithm is O(n).

## Linear search program in C

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);
  return 0;
}
```

## Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



*The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

## Algorithm:

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

```
○ Step 1: [INITIALIZE] SET BEG = lower_bound
  END = upper_bound, POS = - 1
○ Step 2: Repeat Steps 3 and 4 while BEG <=END
○ Step 3: SET MID = (BEG + END)/2
○ Step 4: IF A[MID] = VAL
  SET POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF A[MID] > VAL
```

```
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
```

- **Step 5:** IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- **Step 6:** EXIT

## Binary search program in C

```
#include <stdio.h>
int main()
{
  int c, beg, end, mid, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
  beg = 0;
  end = n - 1;
  mid = (beg + end)/2;
  while (beg <= end)
        {
         if (array[mid] < search)
                beg = mid + 1;
         else if (array[mid] == search)
                {
                printf("%d found at location %d.\n", search, mid+1);
                break;
                }
        else
                end = mid - 1;
        mid = (beg + end)/2;
        }
  if (beg > mid)
    printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

## Sorting:

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

### Sorting Techniques

Sorting technique depends on the situation. It depends on two parameters.
1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.

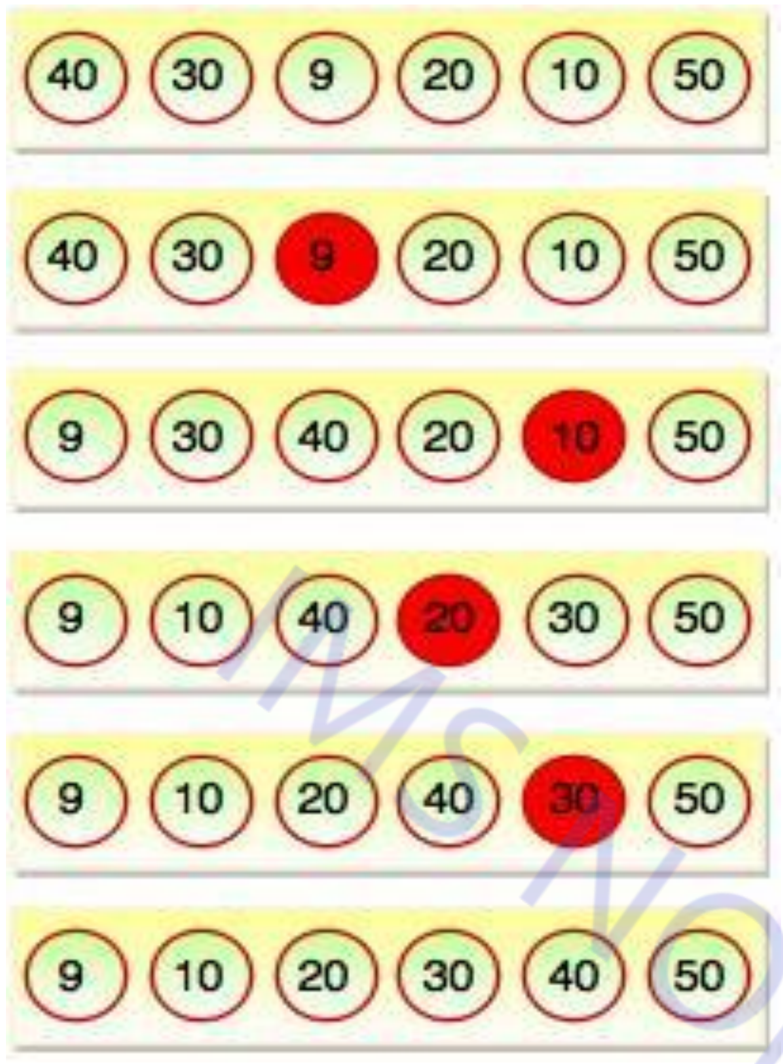
Sorting techniques are differentiated by their efficiency and space requirements.

### Sorting can be performed using several techniques or methods, as follows:

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

**Selection Sort:**

Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.



In the above diagram, the smallest element is found in first pass that is 9 and it is placed at the first position. In second pass, smallest element is searched from the rest of the element excluding first element. Selection sort keeps doing this, until the array is sorted.

## Algorithm for Selection Sort:

The selection sort algorithm is performed using the following steps...

- **Step 1**: - Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

*Time complexity: O(n²)

## Program for Selection Sort

```
#include<stdio.h>
#include<conio.h>

void main()
{

  int size,i,j,temp,list[100];
  clrscr();

  printf("Enter the size of the List: ");
  scanf("%d",&size);

  printf("Enter %d integer values: ",size);
  for(i=0; i<size; i++)
      scanf("%d",&list[i]);

  //Selection sort logic

  for(i=0; i<size; i++)
 {
      for(j=i+1; j<size; j++)
      {
           if(list[i] > list[j])
          {
               temp=list[i];
               list[i]=list[j];
               list[j]=temp;
           }
      }
  }

  printf("List after sorting is: ");
  for(i=0; i<size; i++)
      printf(" %d",list[i]);

  getch();
}
```

**Bubble Sort:**

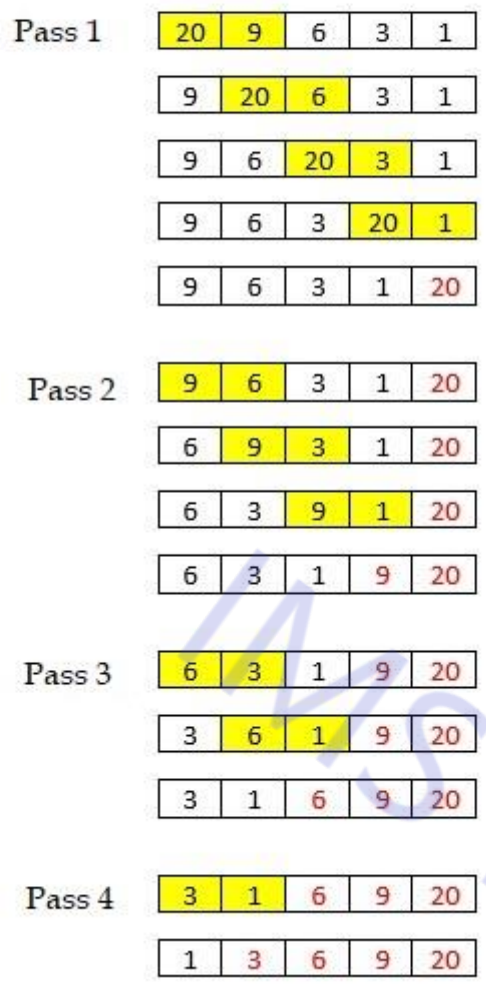
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of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

*Bubble sort is also known as sinking sort.

**Algorithm :**

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i]
  SWAP A[J] and A[i]
  [END OF INNER LOOP]
  [END OF OUTER LOOP
- **Step 4**: EXIT

Time Complexity: O($n^2$)



**Bubble sort program in C**

```
/* Bubble sort code */
#include <stdio.h>
int main()
{
  int array[100], n, c, d, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use < */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
     printf("%d\n", array[c]);

  return 0;
}
```

**Insertion sort:**

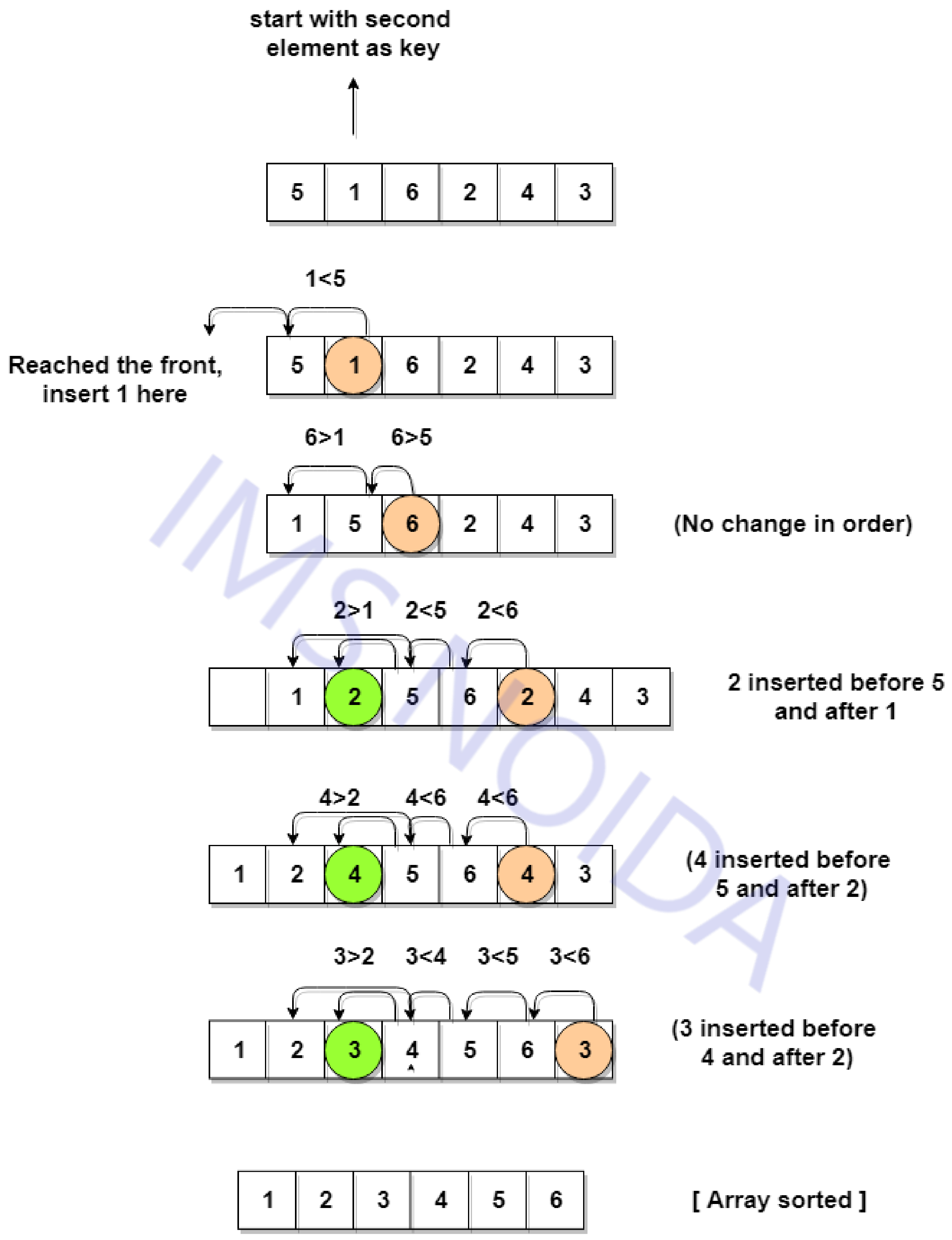
Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.

*Time complexity= O(n$^2$)



start with second element as key
5 1 6 2 4 3
1<5
Reached the front, insert 1 here
5 1 6 2 4 3
6>1 6>5
1 5 6 2 4 3
(No change in order)
2>1 2<5 2<6
1 2 5 6 2 4 3
2 inserted before 5 and after 1
4>2 4<6 4<6
1 2 4 5 6 4 3
(4 inserted before 5 and after 2)
3>2 3<4 3<5 3<6
1 2 3 4 5 6 3
(3 inserted before 4 and after 2)
1 2 3 4 5 6
[ Array sorted ]

**Algorithm:**

```
INSERTION-SORT(A)
   for i = 1 to n
        key ← A [i]
        j ← i – 1
         while j > = 0 and A[j] > key
                A[j+1] ← A[j]
                j ← j – 1
        End while
        A[j+1] ← key
  End for
```

## Insertion sort algorithm implementation in C

```
#include<stdio.h>
#include<conio.h>

int main()
{

   /* Here i & j for loop counters, temp for swapping,
    * count for total number of elements, number[] to
    * store the input numbers in array. You can increase
    * or decrease the size of number array as per requirement
    */
   int i, j, count, key, number[25];

   printf("How many numbers u are going to enter?: ");
   scanf("%d",&count);

   printf("Enter %d elements: ", count);
   // This loop would store the input numbers in array
   for(i=0;i<count;i++)
      scanf("%d",&number[i]);

   // Implementation of insertion sort algorithm
   for(i=1;i<count;i++)
     {
      key=number[i];
      j=i-1;
      while((key<number[j])&&(j>=0))
      {
         number[j+1]=number[j];
         j=j-1;
      }
      number[j+1]=key;
   }

   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
      printf(" %d",number[i]);

   getch();
   return 0;
}
```

**Quick Sort:**

Quicksort is a divide and conquer algorithm.

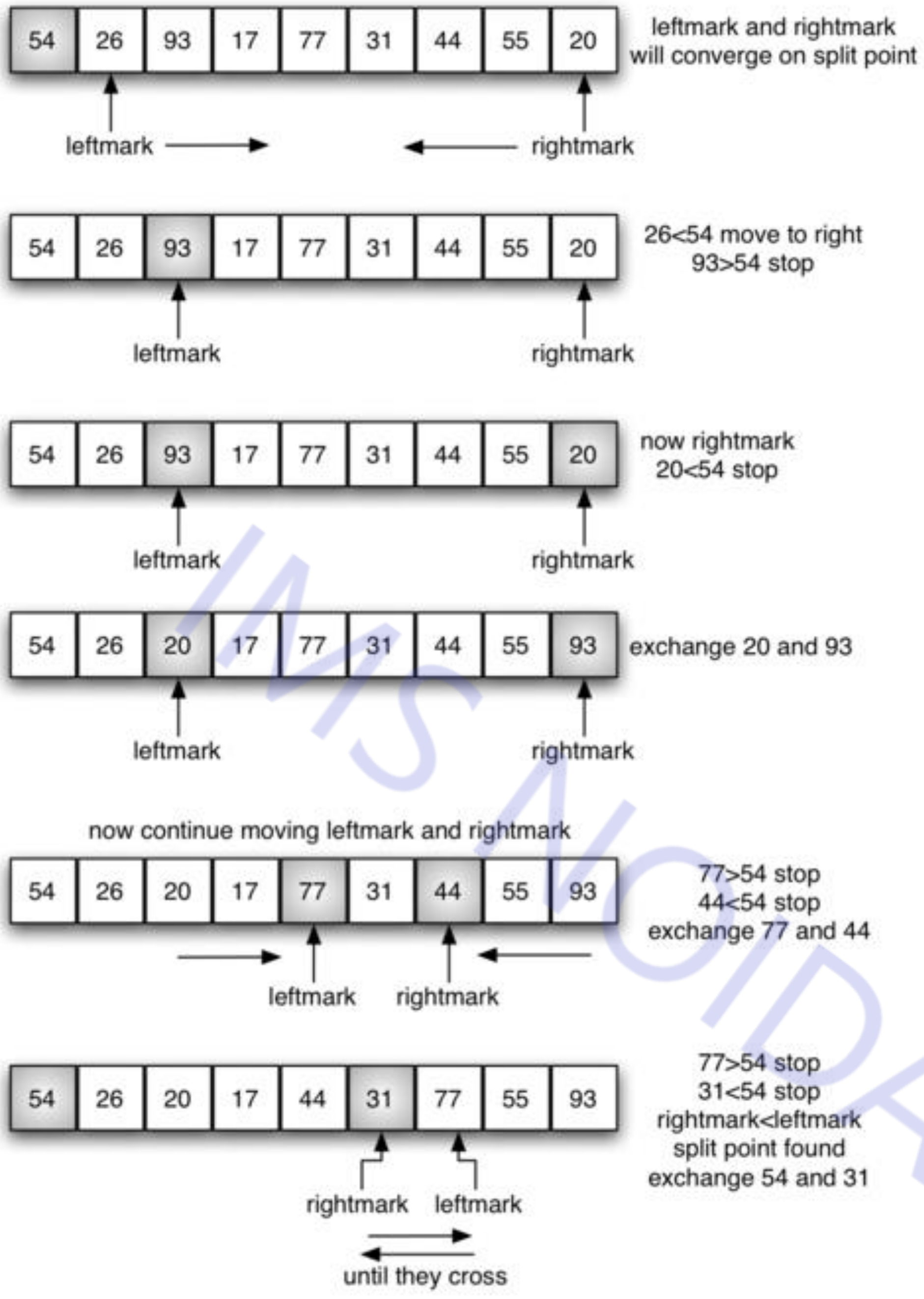
The steps are:

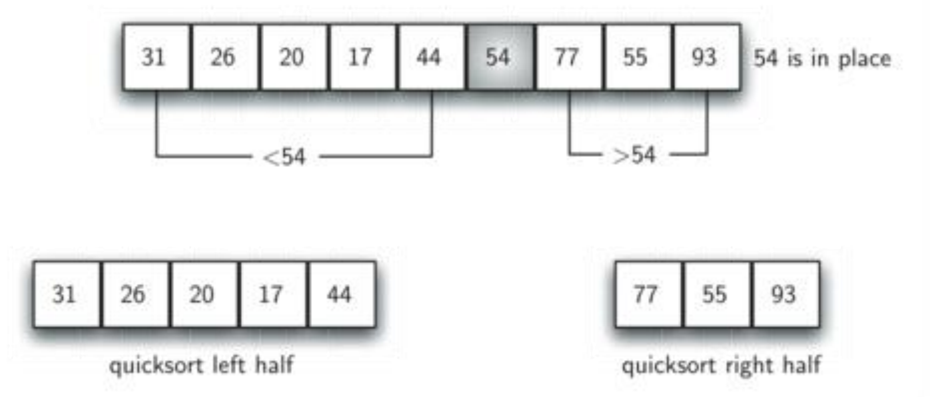
1) Pick an element from the array, this element is called as pivot element.

2) Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation.

3) Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

**Example:**


54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right
93>54 stop
now rightmark
20<54 stop
exchange 20 and 93
now continue moving leftmark and rightmark
77>54 stop
44<54 stop
exchange 77 and 44
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
until they cross



**Algorithm:**

Step 1 − Choose the lowest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

**Time Complexity:**

Worst Case Time Complexity: $O(n^2)$

Best Case Time Complexity: O(n*log n)

Average Time Complexity: O(n*log n)

**Program to implement Quick Sort:**

```
/* C Program to Perform Quick Sort using Recursion
*/
#include <stdio.h>

void quicksort (int [], int, int);

int main()
```

```
{
    int list[50];
    int size, i;

    printf("Enter the number of elements: ");
    scanf("%d", &size);
    printf("Enter the elements to be sorted:\n");
    for (i = 0; i < size; i++)
    {
        scanf("%d", &list[i]);
    }
    quicksort(list, 0, size - 1);
    printf("After applying quick sort\n");
    for (i = 0; i < size; i++)
    {
        printf("%d ", list[i]);
    }
    printf("\n");

    return 0;
}

void quicksort(int list[], int low, int high)
{
    int pivot, i, j, temp;
    if (low < high)
    {
        pivot = low;
        i = low;
        j = high;
        while (i < j)
        {
            while (list[i] <= list[pivot] && i <= high)
            {
                i++;
            }
            while (list[j] > list[pivot] && j >= low)
            {
                j--;
            }
            if (i < j)
            {
                temp = list[i];
                list[i] = list[j];
                list[j] = temp;
            }
        }
        temp = list[j];
        list[j] = list[pivot];
        list[pivot] = temp;
        quicksort(list, low, j - 1);
        quicksort(list, j + 1, high);
    }
}
```

**Output :**

```
Enter the number of elements: 6
Enter the elements to be sorted:
67
45
24
98
12
38
After applying quick sort
12 24 38 45 67 98
```

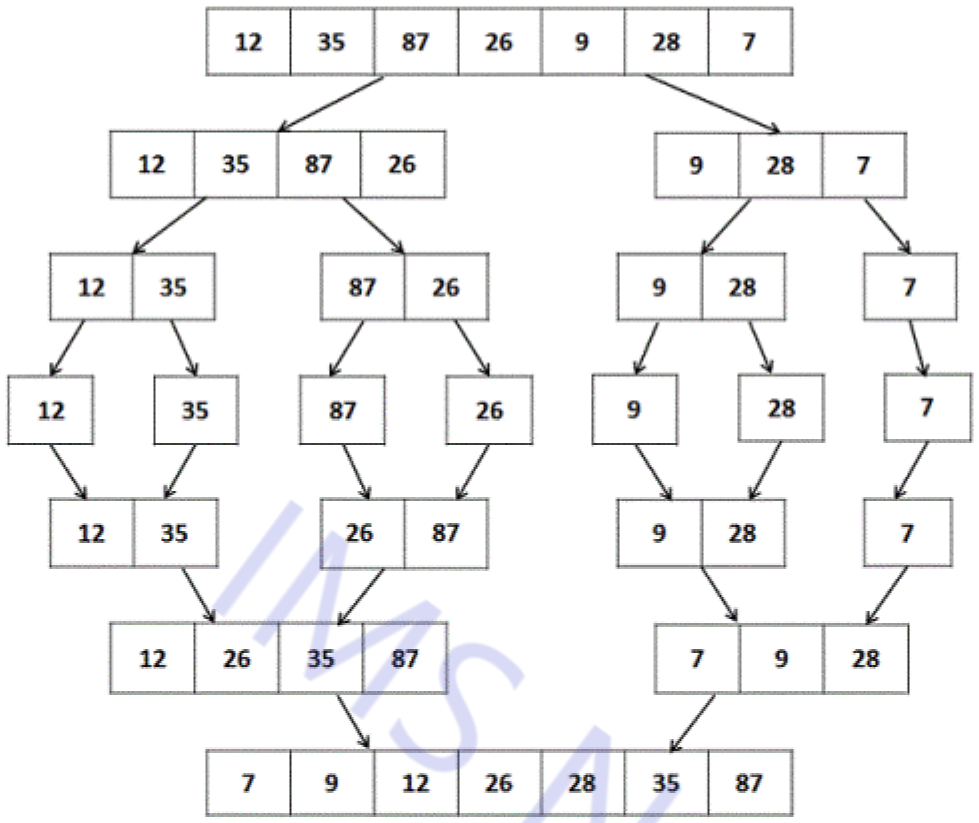
**Merge Sort:**

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

The concept of Divide and Conquer involves three steps:

1. Divide the problem into multiple small problems.
2. Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. Combine the solutions of the subproblems to find the solution of the actual problem.

Example:



**Algorithm:**

```
MergeSort(arr[], LB, UB )
If UB > LB
     1. Find the middle point to divide the array into two
halves:
             middle M = (LB+UB)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, LB, M)
     3. Call mergeSort for second half:
             Call mergeSort(arr, M+1, UP)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, LB, M, UB)
```

**Program to implement merge sort:**

```
#include<stdio.h>
#include<conio.h>

void mergesort(int [],int, int );
void merge(int [],int ,int ,int ,int);

int main()
{
        int a[30],n,i;
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter array elements:");

        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        mergesort(a,0,n-1);

        printf("\nSorted array is :");
        for(i=0;i<n;i++)
                printf("%d ",a[i]);

        getch();
        return 0;
}


void mergesort(int a[],int i,int j)
{
        int mid;

        if(i<j)
        {
                mid=(i+j)/2;
                mergesort(a,i,mid);    //left recursion
                mergesort(a,mid+1,j);  //right recursion
                merge(a,i,mid,mid+1,j);//merging of two sorted sub-arrays
        }
}
```

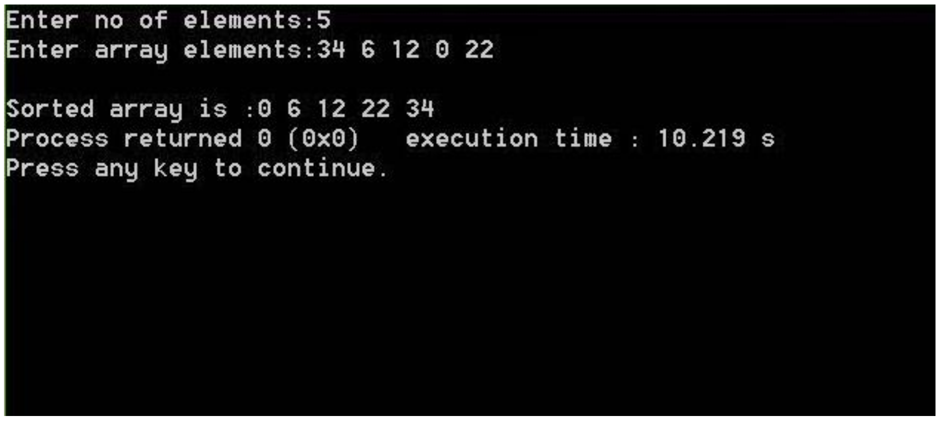
```
void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[50];  //array used for merging
        int i,j,k;
        i=i1;   //beginning of the first list
        j=i2;   //beginning of the second list
        k=0;

        while(i<=j1 && j<=j2)  //while elements in both lists
        {
                if(a[i]<a[j])
                {
                        temp[k]=a[i];
                        i++;
                }
                else
                {
                        temp[k]=a[j];
                        j++;
                }
        k++;
        }

        while(i<=j1)
        {
                temp[k]=a[i];//copy remaining elements of the first list
                i++;
                k++;

        }
        while(j<=j2)
        {
                temp[k]=a[j];//copy remaining elements of the second list
                j++;
                k++;
        }
        //Transfer elements from temp[] back to a[]
        for(i=i1,j=0;i<=j2;i++,j++)
                a[i]=temp[j];
}
```

**Output:**



```
Enter no of elements:5
Enter array elements:34 6 12 0 22

Sorted array is :0 6 12 22 34
Process returned 0 (0x0)   execution time : 10.219 s
Press any key to continue.
```

**Time Complexity:**

Worst Case Time Complexity : O(n*log n)

Average Time Complexity : O(n*log n)

Best Case Time Complexity : O(n*log n)