**SEARCHING ALGORITHMS**

Linear Search

Binary Search

1.

2.

3. Find the middle element mid of array,i.e. arr[(low+high)/2]

4. if x==mid, then return mid. Else compare the element to be searched with m.

5. if x>mid, compare x with middle element of the elements on the right side of the mid. Change low to mid +1.

6. if x<mid, compare x with middle element of the elements on the left side of the mid. Change high to mid - 1.

7. Repeat steps 3 to 6 until low meets high, if both pointing to same, then it means element not found.

Iterative and Recursive function.

**SORTING**

Space used

1. In place – does not require any extra space: Bubble Sort.
2. out place – requires extra space – divide by conquer method: Merge Sort.

Stability

1. Stable – after sorting contents does not change the sequence of similar content in which they appear: Insertion Sort
2. Unstable - after sorting content changes the sequence of similar content in which they appear: Quick Sort

**Bubble Sort(Sinking Sort)**

Repeatedly compare each pair of adjacent items and swap them if they are in the wrong order.

Compare adjacent elements, swap if its small or big according to need as ascending or descending order.

After 1st iteration last element will be correct position.

Time complexity – O(n^2)

Space complexity – O(1)

Average time complexity is poor.

**Selection Sort**

Repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.

Select 1st element as minimum, and check whether this element is small than remaining elements. If this element is not small, then swap otherwise continue with next element by taking it as small number.

After 1st iteration smallest element will be in correct position.

Time complexity – O(n^2)

Space complexity – O(1)

**Insertion Sort**

Time complexity O(n^2)

for(i=1;i<n;i++)

{

key = arr[i];

for(j=i-1;(j>=0) && (key<arr[j]);j--)

{

arr[j+1] = arr[j];

}

arr[j+1]=key;

}

Example: 3 2 1 5 4

1st:

i =1

key = arr[i] = 2

j = i-1 = 0

j>=0 True && key<arr[0] i.e 2<3 True

arr[0+1] = arr[0] i.e arr[1] = 3

j-- -- >> j=0-1 = -1

arr[j+1] = arr[-1+1] = arr[0] = key -- >> arr[0] = 2

Now: 2 3 1 5 4

2nd //// need to dooooooooooooo

i =2

key = arr[2] = 1

j = i-1 = 2-1 = 1

j>=0 True && key<arr[1] i.e 1<3 True

arr[1+1] = arr[1] i.e arr[2] = 3

j-- -- >> j=0-1 = -1

arr[j+1] = arr[-1+1] = arr[0] = key -- >> arr[0] = 2

Now: 2 3 1 5 4

**Merge Sort**

It is a divide and conquer algorithm

Divide the input array in two halves and we keep halving recursively until they become too small cannot be broken further.

Uses extra space

1. Split the unsorted list
2. Compare each of the elements and grp them
3. Repeat step 2 until whole list is merged and sorted.

**Quick Sort**

If pivot element is