Name - Satvik Mittal
CST-SPL-2
Roll no. 34

Tutorial-3

Q1.
Sol

```
Int linearsearch(int A[], int n, int t)
if (abs(A[0]-t) > abs(A[n-1]-t)
  for( i=n-2 to 0 , i--)
    if (A[i]==t)
      return i;
  else
    for(i=0 to n-1; i++)
      if (A[i] == t)
        return i;
  }
```

Q2.
Sol. Iterative insertion sort :-

```
void insertion (int A[], int n)
{
  for(i=1 to n)
  {
    t = A[i];
    j = i;
  while (j>=0 && t < A[j])
  A[j+1] = A[j]
    j--;
  }
  A[j+1] = t;
}
}
```

Recursive Insertion sort :-

```
Void insertion (int A[], int n)
```

```
if (n<=1)
    return;
insertion (A, n-1);
int last = A[n-1]);
int j = n-2;
while ( j>=0 && A[j] >last)
{
    A[j+1] = A[j];
    j--;
}
A[j+1] = last;
}
```

- Insertion sort works if the elements to be sorted are provided one at a time with the understanding that the algorithm must keep the sequence sorted as more elements are added
- Other sorting algos like bubble sort, heap sort are considered external sorting technique, as they need the data to be sorted in advance.

Q3.
Sol.

Sorting	Best case	Worst case
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$
Count sort	$O(n)$	$O(n+k)$
quick sort	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$
heap sort	$O(n \log n)$	$O(n \log n)$

Q4.

Sol.

Sorting	Inplace	Stable	online
① Bubble	✓	✓	X
② Selection	✓	X	X
③ Insertion	✓	✓	✓
④ Count	X	✓	X
⑤ quick	✓	X	X
⑥ Merge	X	✓	X
⑦ Heap	✓	X	X

Q5. Binary Search :-

Sol. Iterative :-

```
int binarySearch(int arr[], int x)
{
    int l=0, r= arr.length-1;
    while (l<=r)
    {
        int m = l+(r-l)/2
        if (arr[m] < x)
            l = m+1;
        else
            r= m-1;
    }
    return -1;
}
```

Recursive :-

```
int binary Search (int arr[], int l, int r, int x)
{
    if (r>=l)
    {
        int mid = l+(r-l)/2;
```

```
if (arr[mid] == x)
    return mid;
else if (arr[mid] > x)
    return binarySearch(arr, l, mid-1, x);
else
    return binarySearch(arr, mid+1, r, x);
}
return (-1);
}
```

Linear Search :-

Iterative - Time complexity = O(n)
Space complexity = O(1)

Recursive - Time complexity = O(n)
Space complexity = O(n)

Binary Search :-

Iterative - Time comp = O(nlogn)
Space comp = O(1)

Recursive - Time comp = O(nlogn)
Space " = O(nlogn)

Q6)d. T(n)

↓

T(n/2)

↓

T(n/4)

↓

⋮

T(n/2^k)

Recurrence relation = T(n/2) + (0/1) ...

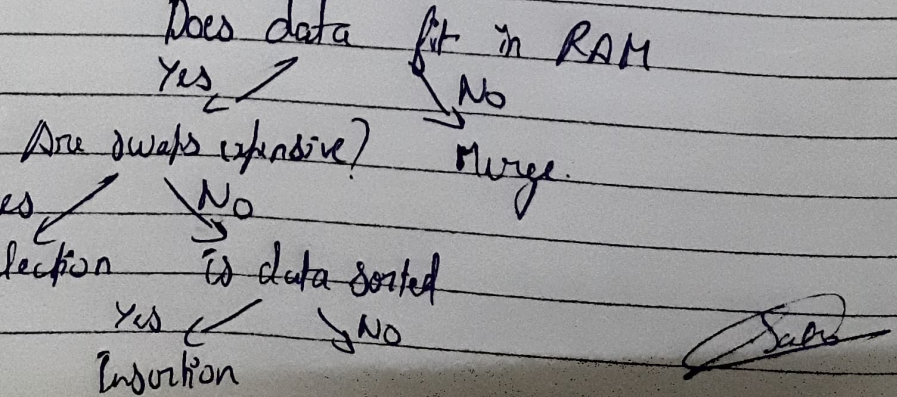
Q7.

Sol.
```
int n;
int A[n];
int key;
int i=0, j=n-1;
while (i<j)
{
  if ((A[i]+A[j])>=key)
    break;
  else if ((A[i]+A[j])<key)
      j--;
  else
      i++;
}
cout<<i<<" "<<j;
```

Time complexity = O(n log n)

Q8. (i) Given time
(ii) Space
(iii) Stable
(iv) No of swaps
(v) Will the data fit in RAM

- There is no best sorting algo, it depends on the situation of type of array provided



No

Can we use extra space

Yes / No

Does it need quick sort
to be stable

Yes / No

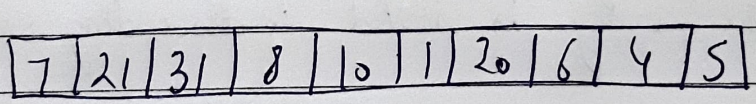
Merge Quick

Q9.

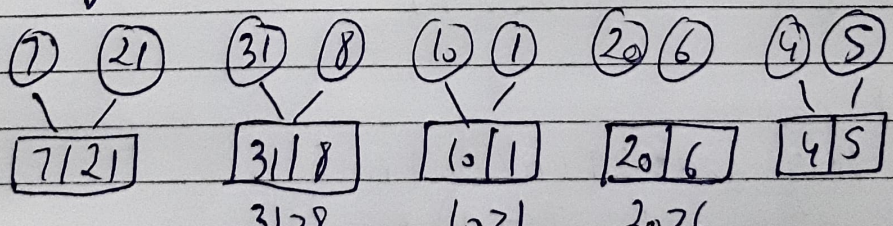
Sol. Inversion in an array shows how far the array is, from being sorted. If the array is already sorted, the inversion count is 0, but if the array is sorted in reverse order, the inversion count is maximum.

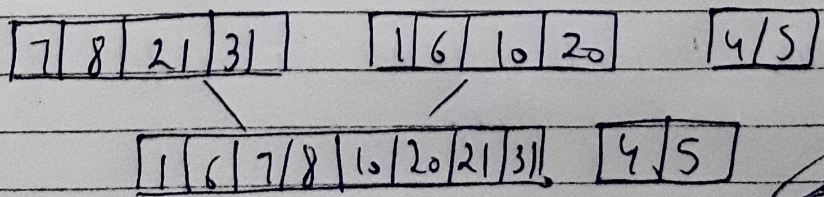
Condition for inversion

$a[i] > a[j]$ & $i < j$



Dividing the array:-





Total inversion in this step :- (12)

7>1, 7>6, 8>1, 8>6, 21>6, 21>20, 31>6, 31>7, 31>20
31>1, 21>6

1	4	5	6	7	8	10	20	21	31

6>4, 6>5, 7>4, 7>5, 8>4, 8>5, 10>4, 10>5, 20>4, 20>5
21>5, 31>4, 31>5

Total inversion = 14
inversion count = 31

Q10. Best case :
Time complexity = $O(n \log n)$

The best case occurs when the partition process always picks the middle element as pivot.

Worst case
Time complexity = $O(n^2)$
When the array is sorted in descending order.

Q11. Best case :-
Sol. Merge sort : $2T(n/2) + n$
quick sort : $2T(n/2) + n$

Worst case :-
Merge sort : $2T(n/2) + n$
quick sort : $T(n-1) + n$

Similarities - They both work on the concept of divide and conquer algorithm.

Page No :

Differences :-

Merge Sort	Quick sort
(i) The array is divided in 2 half	(i) The array is divided in any ratio.
(ii) Worst case - $O(n \log n)$	(ii) Worst case - $O(n^2)$
(iii) Requires extra space	(iii) Does not require extra space
(iv) It is external sorting algo	(iv) It is internal sorting algo.

Q13.

Sol
```
void bubblesort(int A[], int n)
{
  int i, j;
  int j=0;
  for(i=0; i<n; i++)
  {
    for(j=0; j<n-1; j++)
    {
      if(A[j]>[j+1])
      {
        swap(A[j], A[j+1])
        j=1;
      }
    }
    if(j==0);
    break;
  }
```

Solved

Q14.

Sol. When the data set is large enough to fit inside the RAM, we ought to use merge sort, because it uses the divide and conquer approach in which

it keeps dividing the array into smaller parts until it can no longer divide, then merge the array

External sorting :-

- It is used to sort massive amount of data. It is required when data doesn't fit inside the RAM, and instead they must reside in slower external memory.
- During sorting, chunks of small data that fits in main memory are read, sorted and written out to a temporary file
- During merging, the sorted files are combined.

Internal sorting :-

It is used when the entire collection of data is small enough to reside in RAM. Then there is no need of external memory.

eg. Insertion sort, quick sort, heap sort etc