DAA

Tutorial 3 :

Q1 Write linear search pseudocode to search an element in sorted array with minimum campanisions.

```
Int linear_search( 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 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 is also called online sorting algorithm because it will 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 element are added in.

other sorting algorithm like bubble-sort, insertion sort, heap sort e.t.c are considered external sorting technique as they need the data to be sorted in advance.

Q3 Complexity of all sorting algorithm

Sorting	Best Case	Worst Case
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion 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	$O(n\log n)$	$O(n\log n)$

Recursive / Iterative pseudocode for binary search.

Iterative:

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

Recursive:

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

Time Complexity = O(n)

Space Complexity = O(n)

06

$T(n)$

$\downarrow$

$T(n/2)$

$\downarrow$

$T(n/4)$

$\vdots$

$T(n/2^k)$

recurrance relation : $T(n/2) + (0/1)$

07

```
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(nlogn)?

Can we use Extra Space ?

- yes → Does it need to be Stable.
  - yes → Merge Sort
  - No → quick Sort.
- No → quick sor.

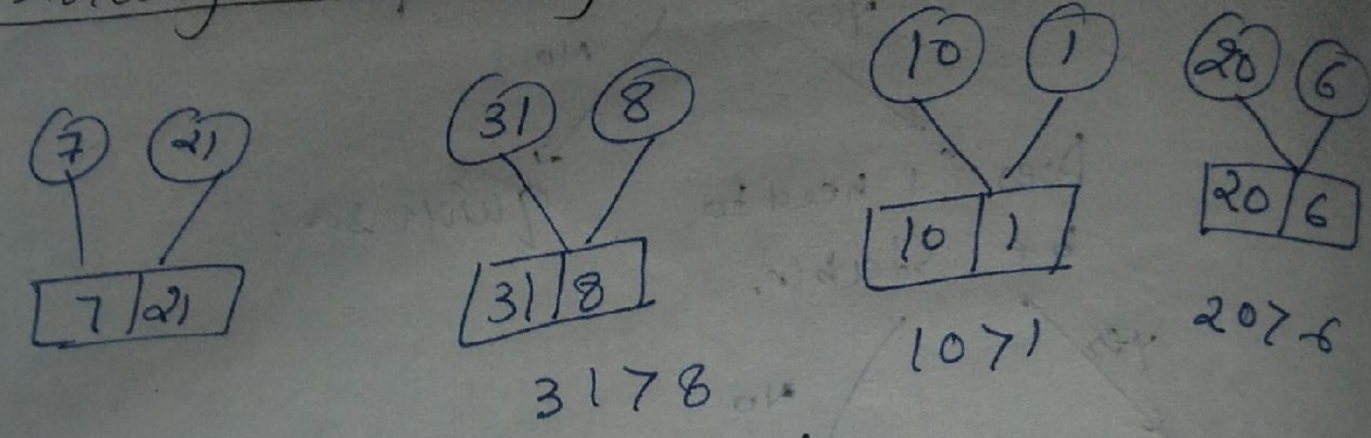
Q) Inversion in an array indicates how far the array is from being Sorted if the array is alredy. sorted the the inversion Convert is 0, but if the array is Sorted in reverse order, then the inversion Count is maximum.

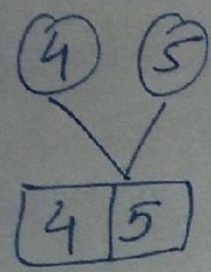
<u>Condition</u> <u>for</u> <u>inversion</u> :

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

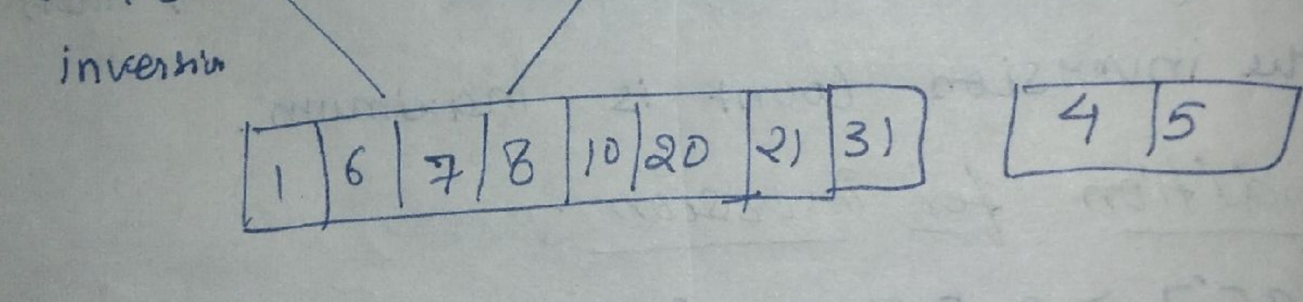
7	21	31	8	10	1	20	6	4	5

Dividing the array:





inversion = 3.



7>1, 7>6, 8>1, 8>6, 21>10, 21>20, 8>

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

Total inversion in this step = 12.

1	4	5	6	7	8	10	20	21	31

= invcount = 1

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

Total inversion in this step = 14.

Inversion Count = 31

Best Case.

Time complexity = O(n log n)

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

Worst Case.

Time Complexity : O(n^2)

When the array is sorted in ascending descending order.

Q4 Best Cases:

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 the divide & Conquer algorithm. Both have best Case Complexity of $O(n \log n)$

Differences

Merge Sort	Quick Sort
(i) The array is divided into just 2 half.	(i) The array is divided in any ratio.
(ii) Worst Case Complexity is $O(n \log n)$	(ii) Worst Case Complexity $O(n^2)$
(iii) It requires Extra Space i.e not inplace	(iii) It does not require Extra Space i.e inplacement.

Merge	Quick
It is External sorting algoritm, it is stable	(iv) It is internal sorting algoritm & not stable.
Works consistently on any fine of data set	(V) Works fast on small data set.

Selection sort is not stable by default but you can write a version of stable selection sort

```
void selection (int A[], int n)
{  for (int i=0; i<n-1; i++)
     { int num = i;
       for (int j= i+1; j<n; j++)
     {
         if (A[min] > A[j])
             min = j;
         int key > A [min];
```

```
while (min > i)
{   A[min) = A [min-1]
        min--;


    }

    A[i] = key;

  }
}
```

Q13

```
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] > A [j+1])
          {
          swap (A[j], A [j+1])
             j= 1;
    }
  }          if (j==0)
               break;
  }
```

Q14) When the data set is large enough to fit inside RAM, we ought to use Merge sort. because it uses the divide & Conquer approach in which it keeps dividing the array into smaller parts. Until it can no longer be splited. it then merge the array divided in n parts. therefore at the time only a part of array is taken on ram.

External Sorting:

It is used to sort massive amount of data. It is required when the data doesn't fit inside the RAM & insted they must reside in the slower External memory.

During sorting, chunks of small data that can fit in main memory are read, sort and written out to a temporary file.

During Merging, the sorted subfiles are combines into a single large files.

Internal Sorting:

It is a type of sorting which is used when the intrue collection of data is small enough to reside within RAM. Then there is no need of external memory for program execution. It is used when input is small.

eg Insertion sort, quick sort, heap e.t.c