Name - Tisha Aggarwal
Sec - CST SPL2
Roll No - 31

DAA - Tutorial 3

Q1)

```
int linearS (int A[], int n, int k)
{ if((A[0]-t) >(A[n-1]-k))
  { for (i:n-1 to 0 ; i--)
    { if ( A[i]==k)
      { return i ;
      }
    else
     for (i=0 to n-1 ; i++)
      if (A[i]==t)
        return i;
     }
```

Q2) 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;
}
```

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;
    }
}
```

→ Online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the beginning. Insertion sort considers one input element per iteration and produces a partial solution without considering future elements. Whereas, other sorting techniques like bubble sort etc are external sorting techniques as they need data to be sorted in advance.

Q3) Sorting

1) Bubble Sort :- $O(n^2)$ [Best Case] ; $O(n^2)$ [Worst Case]
2) Selection Sort :- $O(n^2)$ [Best and Worst Case]
3) Insertion Sort :- $O(n)$ [Best Case] ; $O(n^2)$ [Worst Case]
4) Count Sort :- $O(n)$ [Best Case] ; $O(n+k)$ [Worst Case]
5) Quick Sort :- $O(n \log n)$ [Best Case]; $O(n^2)$ [Worst Case]
6) Merge Sort :- $O(n \log n)$ [Worst and Best Case]
7) Heap Sort :- $O(n \log n)$ [Worst and Best Case]

Q4)

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

Qs) Iterative pseudo code for binary search :- (2)

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

Recursive code :-

```
int binarys ( int arr [], int l, int r, int a)
    { if (r ≥ l)
    { int mid = l + (r - l) / 2;
      if (arr [mid] = = a)
         return mid;
     else if (arr [mid] > x)
      return binarys (arr, l, mid - 1, a);
        else
      return binarys ( arr, mid + 1, r, a);
     }
     return (-1);
    }
```

* Iterative - Time Complexity : O(n log n)
  Space Complexity : O(1)

  Recursive - Time complexity : O(n log n)
  Space complexity : O(log n)

* Linear Search :-
  Iterative - Time Complexity : O(n)
  Space complexity : O(1)

  Recursive - Time Complexity : O(n)
  Space complexity : O(n)

Q6) $T(n)$
↓
$T(n/2)$
↓
$T(n/4)$
⋮
$T(n/2^k)$

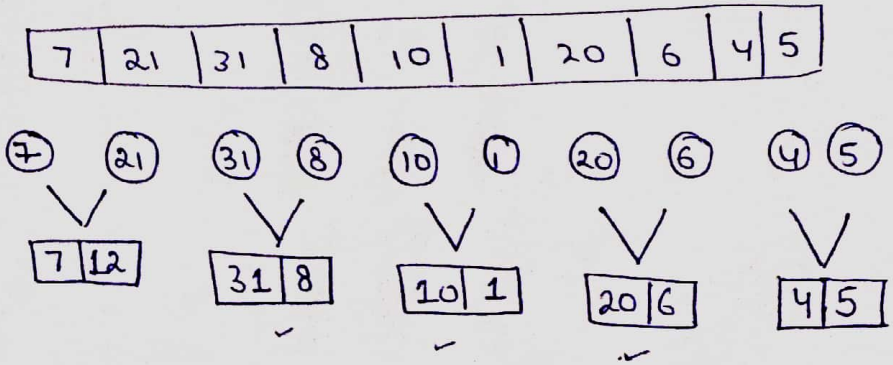
recurrence relation = $T(n/2) + (0/1)$.

Q7)
```
Int A[n], key, n;
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) Quicksort is most commonly used as it is the fastest general purpose sort.
It can be done without taking extra memory i.e., in-place.

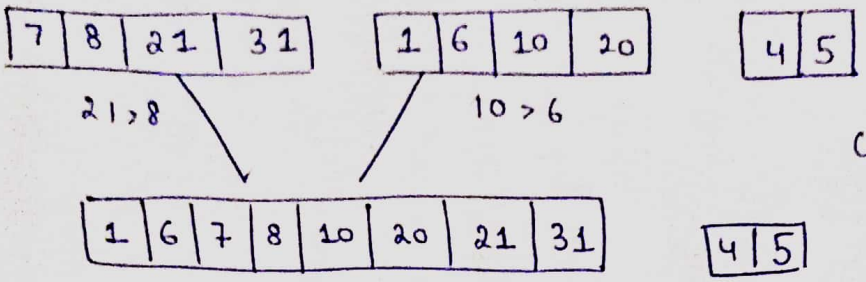
Q9) Inversion Count for an array indicates how close the array is from being sorted. If array is already sorted, then the inversion count is 0, but if array is sorted in reverse order, the inversion count is the maximum.

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



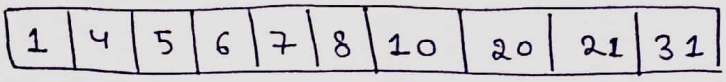
Count = 3

(5)



Total inversion = 12

7>1, 7>6, 8>1, 8>6, 21>10, 21>20, 31>1, 31>6, 31>7, 31>20, 21>1, 21>6



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 = 14

Total count of inversion = 31. (Ans)

Q10) Best Case when partition process always picks middle element as pivot.

Time Complexity : $O(n \log n)$

Worst Case is when array is sorted in ascending or descending order.

Time Complexity : $O(n^2)$

Q11) Best Case : Merge Sort - $2T(n/2) + n$
Quick Sort - $2T(n/2) + n$

Worst Case : Merge Sort - $2T(n/2) + n$
Quick Sort - $T(n-1) + n$

→ Differences :-

Merge Sort	Quick Sort
1) The array is divided into 2 half.	1) The array is divided in any ratio
2) Worst Case Complexity is $O(n \log n)$	2) Worst Case Complexity is $O(n^2)$
3) It requires extra space	3) It does not need extra space

Merge Sort	Quick Sort
4) It's external sorting algorithm and is stable.	4) It is unstable and internal sorting algorithm
5) Works consistently on any size.	5) Works fast on small data set.

→ Similarities - They both work on concept of divide and Conquer algorithm. And both has best case complexity of O(nLogn).

Q12)
```
void Selection (int A[], int n)
{
for (int i=0; i<n-1; i++)
    {
     int min=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 bubble Sort (int A[], int n)
    { int i, j;
    int f =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])
         f=1;
   } }
   if (f==0)
     break;
 }
```

Q13) When data set is large enough to fit inside RAM, we use merge sort as it uses the divide and conquer approach which keeps dividing array into smaller parts until it can no longer be divided, then it is merged. Thus at a 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 and instead they must reside in the slower external memory. During sorting, chunks of small data that can fit in main memory are read and sorted to a temporary file. During merging, sorted subfile combines into a single large files.

Internal Sorting - It is a type of sorting which is used when entire data is small enough to reside within RAM. Then we don't need external memory for program execution. It is used when input is small.
Eg: Bubble, quick etc.