Name : T Rishita Reddy | Tutorial - 3 |
Sec: I
Rollno: 47

Q.1

Ans→
```
void linearSearch(int A[], int n, int key)
{
    int flag=0
    for(int i=0; i<n; i++)
    {
        if (A[i] == key)
        {
            flag=1;
            break;
        }
    }
    if (flag==0)
    cout << "Not found";
    else
        cout <<" found";
}
```

Q.2

Ans→ Iterative:
```
for i=1 to n-1
    t = A[i] , j=i-1
    while (j>=0 && A[j]>t)
    {
        if (A[j+1] = A[j])
            j--;
    }
    A[j+1]=t;
}
```

Recursive:

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

}
```

Insertion sort is an online algorithm because insertion sort considers one input element per iteration & produces a partial solution without considering future elements.

But in case of other sorting algorithm / we require access to the entire input, thus they are offline algorithm.

Q.3

Ans:

| Algorithm | Worst case | Best Case | Average Case |
|---|---|---|---|
| Bubble Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion Sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| Count Sort | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ |
| Quick Sort | $O(n^2)$ | $O(n\log(n))$ | $O(n(\log n))$ |
| Merge Sort | $O(n(\log n))$ | $O(n\log n)$ | $O(n\log n)$ |
| Heap Sort | $O(n(\log n))$ | $O(n(\log n))$ | $O(n(\log n))$ |

Q.4

Sol$^n$:

| Algorithm | Implace | Stable | Online |
|---|---|---|---|
| Bubble Sort | ✓ | ✓ | ✗ |
| Selection Sort | ✓ | ✗ | ✗ |
| Insertion Sort | ✓ | ✓ | ✓ |
| Count Sort | ✗ | ✓ | ✗ |
| Merge Sort | ✗ | ✓ | ✗ |
| Quick Sort | ✓ | ✗ | ✗ |
| Heap Sort | ✓ | ✗ | ✗ |

Q.5

Sol$^n$: Recursive:

```
int binarySearch(int arr[], int l, int r, int key)
 {
    if (r>=l)
     {
        int mid = l+(r-l)/2;
        if (arr[mid]==key) return mid;
         if (arr[mid]) > key
        return binarySearch(arr, l, mid-1, key);
        return binarySearch(arr, mid+1, r, key);
     }
    return -1;
 }
```

Iterative:

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

| | Time Complexity | | Space Complexity | |
|---|---|---|---|---|
| | Recursive | Iterative | Recursive | Iterative |
| Linear Search | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |
| Binary Search | $O(\log n)$ | $O(\log n)$ | $O(1)$ | $O(1)$ |

Q.6

Ans⇒ $T(n) = T(n/2) + 1$

Q.7

Sol^n: 

```
void Sum(int A[], int k, int n)
{
  Sort(A, A+n);
  int i=0, j=n-1;
  while(i<j)
  {
    if(A[i]+A[j]==k)
      break;
```

```
else if (A[i] + A[j]>k)
        j--;
    else
        i++;
  }
print (i, j)
}
```

Here sort function has $O(n(\log n))$ complexity & for while loop it is $O(n)$

$\therefore$ Overall complexity $= O(n(\log n))$

Q.8

Ans→ In pratical use, we mostly prefer merge sort. Because of its stability & it can best for very large data. Further more, the time complexity of merge sort is same in all cases that is $O(n(\log n))$.

Q.9

Sol$^n$: Inversion count for any Array indicates - how far (or close) the array is from being sorted. If the array is already sorted, inversion count is 0, but if the array is sorted in reverse order the inversion count is maximum.

Pseudo Code for Inversion Count:

```
int getInvCount (int arr[], int n)
{
  int c=0;
  for (i=0; i<n-1; i++)
  {
     for (int j=i+1; j<n; j++)
     {
        if (arr[i]>arr[j])
            c++;
     }
```

}
return 0;

}

arr [] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

Total inversion ⇒ 31

Q.10

Ans ⇒ When the array is already sorted or sorted in reverse order. Quick sort gives the worst case time complexity i.e. $O(n^2)$. But when the array is totally unsorted, it will give best case time complexity i.e. $O(n \log n)$.

Q.11

Solⁿ:

| Algorithm | Recurrance Relation | |
|---|---|---|
| | Best Case | Worst Case |
| Quick Sort | $T(n) = 2T(n/2) + n$ | $T(n) = T(n-1) + n$ |
| Merge Sort | $T(n) = 2T(n/2) + n$ | $T(n) = 2T(n/2) + n$ |

Both the algorithms are based on the divide & conquer algorithm. Both the algorithms have the same time complexity in the best case & average because both the algorithms divides array into supports, sort them & finally merge all the sorted parts.

Q.12

Solⁿ: As the selection sort is not stable because it changes the relative positive of some elements after sorting.

Selection sort can be made stable if instead of swapping the minimum element is placed in its position without swapping i.e. by placing the number in its position by pushing every. Element one step forward. In simple words use insertion sort technique which means inserting element in its correct place.

Pseudo Code for stable Selection Sort:

```
void StableSelectionSort (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;
   }
 }
```

Q.13

Soln: Pseudo Code for Modified bubble Sort

```
void bubble(int A[] ,int n)
{
  for (int i=0 ; i<n; i++)
  {
   int swaps = 0;
```

```
for (int j=0; j<n-i-1; j++)
 { if (A[j] > A[j+1])
   { Swap(A[j], A[j+1]);
     Swaps++;
   }
 }
if (swaps==0)
  break;
}
}
```

Q.14

Sol$^n$: For the array of 4GB, we use the External Sorting because array size is greater than the RAM of our computer.

→ External Sorting: These are sorting algorithms that can handle large data amounts which cannot fit in the main memory. Therefore only a part of the array resides in the RAM during execution.

Ex: K-way Merge Sort.

→ Internal Sorting: These are sorting algorithms where the whole array needs to be in the RAM during execution.

Ex: Bubble Sort, Selection Sort, etc.