Name: Pratham Sesodia

Section: ML

Roll no: 41

Assignment -2

Q1.

Ans

```
int linearSearch (int a[], int n, int t)
      int index = -1
       for (int i=0 ; i<n ; i++)
       { if (a[i] == t)
           { index = i
               break;
           }
          else if (a[i] < t)
                 break;
        }
        return index
   }
```

Q2. Iterative approach

Ans-

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

Recursive approach

```
void insertionRecursive (int a[], int n)
{   if (n <=1)
         return;
 insertionRecursive (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 sorting is also known as Online sorting because that processes its input in a serial fashion i.e. in order that the input is fed to algorithm, without having entire input available from beginning.

Other sorting algorithms are!

- Bubble sort
- Insertion sort
- Selection sort
- Count sort
- Merge sort
- Quick sort
- Heap sort

Q3.

Ans.

Sorting	Time Complexity			Space complexity
	Best	Avg	Worst	
Bubble sorting	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sorting	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sorting	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Count sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$

Q4.

Ans.

Inplace	Stable	Online
Bubble sort	Bubble sort	Insertion sort
Selection sort	Insertion sort	
Insertion sort	Counting sort	
Quick sort	Merge sort	
Heap sort		

Q5.

Ans. Iterative approach

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

Recursive approach

```
bool binary (int a [], int l, int r, int key
{ if (l > r)
      return false;
   int mid = l+(r-l)/2 ;
   if (a[mid] == key)                  → O(1)
      return true;
   else if (a[mid] < key)              → T(n/2)
      binary ( a , mid+1, r, key);
   else                                → T(n/2)
      binary ( a, l, mid-1, key);
}
```

Linear Search Pseudo Code

```
for (i = 0 ; i < n ; i++)
    { if (arr[i] == key)
          return i;
    }
return -1;
```

Time complexity = $O(n)$ (Worst) (Average)
Best = $O(1)$
Space complexity = $O(1)$

Binary Search Pseudo Code (Recursive)

```
bool binary search (int arr, int l, int r, int key)
{ if (l > r)
      return False;
  int mid = l + (r-l)/2;
  if (arr[mid] == key)
      return true;
  else if ( arr[mid] < key)
      binary search( arr, mid+1, r, key);
  else
      binary search ( arr, l, mid-1, key);
}
```

TC = $O(\log n)$ (Worst) (Average)
Best = $O(1)$
SC = $O(\log_2 n)$ (Space complexity)

Q6.

Ans. Recursive approach

```
bool binary (int a [], int l, int r, int key
{ if (l > r)
     return false;
  int mid = l+(r-l)/2 ;
  if (a[mid] == key)                      → O(1)
      return true;
  else if (a[mid] < key)                  → T(n/2)
      binary ( a , mid+1, r, key) ;
  else                                    → T(n/2)
      binary ( a, l, mid-1, key);
}
```

Recurrence Relation = $\boxed{T(n) = T(n/2) + 1}$

Q7.

Ans

```
# include <iostream>
using namespace std;
int main ()
{ int a[] = {2, 7, 11, 15};
  int n = sizeof (a) / sizeof (a[0]); int key;
  cout << "Enter the number:";
    cin >> key;
  for ( int i = 0 ; i < n ; i++)
   { int temp = i+1;
       if ( a[i] + a[temp] == key)
         cout << "Indices are:" << i << "," << temp;
    }
  }
```

This program has a time complexity of O(n).

Q8.

Ans. Quick sort is best for practical uses because of its impressive average case time complexity of O(n log n) and requires little space and exhibits good cache locality. It is an in-place sorting that means no additional storage space is needed to perform sorting.

Q9.

Ans. Number of inversions in an array indicates how far or close the array is from being sorted. If it is already then inversion count is 0, but if array is sorted in reverse order then the inversion count is maximum.

Pseudo code to count the no. of inversions using merge sort.

```
int mergeSort (int a[], int n)
{ int temp [n];
    return inversion (a, temp, 0, n-1);
}

int inversion (int a[], int temp, int l, int r)
{ int mid, invCount = 0;
  if (r>l)
  {mid = (r+l)/2;
   invCount += inversion (a, temp, l, mid);
   inv Count += inversion (a, temp, mid+1, r);
   invCount += merge (a, temp, l,mid +1, r);
  }
 return invCount;
}


int merge (int a[], temp, int l, mid, r)
{ int invCount = 0;
  int i = l, j = mid, k = l;
  while ((i <= mid -1)&&(j <= r))
  { if (a[i] <= a[j])
     { temp[k++] = a[i++];
     }
     else
     { temp[k++] = a [j++];
       inv Count = invCount + (mid -i);
     }
  }
   while (i <= mid -1)
      temp [k++] = a[i++];
while (j <= r)
  temp[K++] = a[j++];
for (int i = left ; i <= r ; i++)
     a[ i]= temp [i];
 return invCount ;
}
```

On inserting input as {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} output will come out as 31 number of inversions.

Q10.

Ans. The best case time complexity for quicksort is when the pivot divides the array into two equal halves leading to balanced partitions. At this time, the time complexity of quick sort is $O(n \log n)$.

The worst case time complexity for quick sort is when the pivot chosen for partitioning does not split the array in equal two halves instead leads to unbalanced partitioning. This happens when the pivot is smallest or largest element of array, resulting in a time complexity of $O(n^2)$.

Q11.

Ans. Merge Sort

```
void mergeSort (int A[], int l, int r)
{ if (l < r)
   { int mid = l + (r - l)/2 ;
     mergeSort (A, l, mid);     → T(n/2)
     mergeSort (A, mid+1, r);   → T(n/2)
      merge (A, l, mid, r);     → O(n)
   }
}
```

Recurrence Relation $\boxed{T(n) = 2T(n/2) + n}$

Quick Sort

```
void quick(int a[], int l, int h)
{ if(l<h)
  { int p=partition(a,l,h); → O(n)
    quick(A,l,p-1);         → T(n/2)
    quick(A,p+1,h);         → T(n/2)
  }
}
```

Recurrence Relation $T(n) = 2T(n/2) + n$

Differences between Quick and Merge sort

Quick	Merge
1) It is not stable	It is stable.
2) Used for arrays.	Used for linked lists
3) In-place sorting method.	Not In-place sorting method
4) Internal sorting	External sorting.
5) Less space required	More space required.

Similarities in Quick and Merge sort

1) Both works on Divide and Conquer technique.
2) Both have best case time complexity $O(n \log n)$.
3) Both have average case time complexity as $O(n \log n)$.

Q12.

Ans. Pseudo code of Stable Selection Sort

```
void stable (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] ;
     for (int k = min ; k > i ; k--)
          a[k] = a[k-1] ;
       a[i] = key ;
   }
}
```

Q13.

Ans. Bubble sort can be modified so that it doesn't scan the whole array once it is sorted by stopping the algorithm if the inner loop didn't cause any swap.

Pseudo code

```
void modified (int a[], int n)
{ bool swapped ;
   for (int i=0 ; i < n-1 ; i++)
   { swapped = false ;
     for (int j=0 ; j < n-i-1 ; j++)
       { if (a[j] > a[j+1])
           { swap (a[j], a[j+1]) ;
             swapped = true ;
           }
       }
```

```
    if (swapped == false)
      break;
   }
 }
```

Q14.

Ans. External sorting such as K-way merge sort is best suited for this purpose as we can divide our source file into smaller temporary files, sort the temporary files and then creating a new file using these temporary files.

The concept of External sorting is to divide our source file into smaller temporary files, sorting the temporary files and then making a new file using them. It is used when all data to be sorted can't be placed in a memory at a time.

The concept of Internal sorting is when all data is placed in the main memory and cannot take input beyond its size.