Name - GOVIND PRATAP SINGH
Sec. - DS 1
Roll No. - 52 (2021689)

Ans. 1.

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

Ans. 2 Iterative:

```
void insertionSort(int arr[], int n)
{
    int i,j,t;
    for(i=0; i<n; i++)
    {
        t = arr[i];
        j = i-1;
```

```
        while(j>=0 && arr[j]>t)
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = t;
    }
}
```

Reursive:

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

Insertion sort is called an online sorting algorithm as it processes its input in a serial manner i.e. in the order that the input is fed to the algorithm.

As for the other sorting algorithms discussed in lectures, the ones that are online sorting algorithms are: ~~Binary sort~~, count sort (with modification) and ~~Count sort~~, while bubble sort, selection sort, merge sort and radix sort are not online sorting algorithms.

Ans3.

Sorting Technique	Time Complexity		
	Best	Average	Worst.
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Count Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$

Ans.4.

Sorting Algorithm	Type		
	Inplace	Stable	Online
Bubble Sort	✓	✓	✗
Selection Sort	✓	✗	✗
Insertion Sort	✓	✓	✓
Count Sort	✗	✓	✗
Quick Sort	✓	✗	✗
Merge Sort	✗	✓	✗
Heap Sort	✓	✗	✗
Radix Sort	✗	✓	✗

Ans.5. Recursive:

```
int binarySearch(int arr[], int l, int r, int key)
{
  if (l > r)
  {
    return 0;
  }
  else
  int mid = (l + (r - l)) / 2;
  if (arr[mid] == key)
  {
    return 1;
  }
  else if (arr[mid] < key)
  {
    binarySearch(arr, mid+1, r, key);
  }
else
```

```
        binarySearch(arr, l, mid-1, key);
    }
}
```

Iterative:

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

Sorting Algorithm	Time Complexity			Space Complexity
	Best	Avg.	Worst	
Binary Search (recursive)	O(log n)	O(log n)	O(log n)	O(log n)
Linear Search (")	O(1)	O(n)	O(n)	O(1)
Binary Search (iterative)	O(log n)	O(log n)	O(log n)	O(1)
Linear Search (")	O(1)	O(n)	O(n)	O(1)

Ans.6.

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

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

7.

```
void funcSeven(int arr[], int n, int key)
{
    int i, j, t;
    for(i=0; i<n; i++)
    {
        t=arr[i];
        if(key>arr[i])
        {
            for(j=i+1; j<n; j++)
            {
```

```
if (t == arr[j] == key)
{
    print (i);
    print (j);
    return ;
}
}
}
}
```

8. There is no "best" sorting technique per se, although different sorting algorithms are best when used for specific tasks.
For instance, when dealing with large datasets, megresort and quick sort are preferred, while for smaller datasets; insertion sort or selection sort may be preferred.
In general, quick sort is often regarded as the "best" sorting algorithm.

9. The number of inversions in an array refers to the number of steps it will take for that array to be sorted. For e.g. a sorted array has inversion count 0, whereas a reverse sorted array will have to go through n inversions to be sorted correctly.

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

{ 7, 21, 31, 8, 10, 1, 20, 6, 4, 5 } — 1

{ 7, 21, 31, 8, 10 } { 1, 20, 6, 4, 5 } — 2

{ 7, 21, 31 } { 8, 10 } { 1, 20, 6 } { 4, 5 } — 4

{ 7, 21 } { 31 } { 8 } { 10 } { 1, 20 } { 6 } { 4, 5 } — 8

{ 7 } { 21 } { 31 } { 8, 10 } { 1 } { 20 } { 6 } { 4, 5 } — 12

{ 7, 21 } { 31 } { 8, 10 } { 1, 20 } { 6 } { 4, 5 } — 14

{ 7, 21, 31 } { 8, 10 } { 1, 6, 20 } { 4, 5 } — 16

{ 7, 8, 10, 21, 31 } { 1, 4, 5, 6, 20 } — 18

{ 1, 4, 5, 6, 7, 8, 10, 20, 21, 31 } — 19

∴ 19 inversions.

10. Quick sort gives best time complexity in cases where the pivot element divides the array into two roughly equal parts. Whereas it gives the worst case time complexity ($O(n^2)$) when the largest or smallest element is repeatedly chosen as the pivot element.

11.
```
void quickSort(int arr[], int l, int h) — T(n)
{
    if(l < h)
    {
        int p = partition(arr, l, h)      — O(n)
        quickSort(arr, l, p-1)            — T(n/2)
        quickSort(arr, p+1, h);           — T(n/2)
    }
}
```

~~int part~~

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

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

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

for merge sort the best and worst case time complexity is nlogn, hence same recurrence relation.

Worst case for quick sort.

$$T(n) = 2T(n-1) + n$$

The similarities between merge and quick sort's time complexities are that they both take O(nlogn) time for best and Average cases whereas quick sort takes $O(n^2)$ for worst case and merge sort's worst case time complexity is also O(nlogn)

12.
```
void stableselectionSort( int arr[], int n)
{
  for( int i=0; i<n-1; i++)
    {
      int min= i;
      for(int j = i+1; j<n; j++)
      {
        if( arr[min] > arr[j])
        { min = j;
        }
      int key= arr[min]
      while( min>i)
        {
          arr[min]= arr[min -1];
      } min--;
```

```
            arr[i] = key;
        }
    }
```

13.
```
bool bubbleSort(int arr[], int n)
{
    int i, j, swap, temp;
    swap = 1;
    while (swap == 1)
    { swap = 0;
      for(i = 0; i < n-1; i++)
      {
          for(j = 0; j < n-i-1; j++)
          {
              if(arr[j] > arr[j+1])
              {
                  swap = 1;
                  temp = arr[j+1];
                  arr[j+1] = arr[j];
                  arr[j] = temp;
              }
          }
      }
    }
```

14. For the given purpose I will use Merge sort algorithm or any external sorting algorithm. We use the given methods as they break large datasets into many smaller pieces and sort them individually, which

improves efficiency.

External Sorting : When the internal memory or RAM of a system is not big enough to accomodate a file's data for sorting, the sorting is performed by storing the data in a hard disk, SSD or other external storage devices. This type of sorting is called external sorting.

Internal Sorting : If the ~~Inter~~ internal memory of a system is sufficient for performing sorting without the need of external storage, it is called internal sorting.