Q1) Write linear search algorithm/pseudocode to search an element in a sorted array with minimum comparisons.

```
boolean linear(int a[], int e, int n)
 {
   int c=0;
   for(int i=0;i<n;i++)
   {
     if(a[i]==e)
       c=1;
     else
       c=0;
   }
   if(c==1)
   return true;
   else
   return false;
 }
```

Q2) Write pseudocode for iterative and recursive insertion sort. Insertion sort is called online sorting. Why? What about other sorting algorithms that have been discussed in lectures?

Iterative -

```
void insertion(int a[], int n)
{
```

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

Recursive-

```
void Insertion(int arr[],int i,int n)
 {
  int val=arr[i];
  int j=i;
  while(j>0 && arr[j-1]>val)
  {
   arr[j]=arr[j-1];
   j--;
  }
  arr[j]=val;
  if(i+1==n)
  Insertion(arr,i+1,n);
 }
}
```

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

Q3) Complexity of all sorting algorithms that have been discussed.

Sorting Technique	Best	Average	Worst
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Count	$O(n+k)$	$O(n+k)$	$O(n+k)$

Q4) Divide all sorting algorithms into stable/inplace/online sort.

Sorting Algorithm	In-place	Stable	Online
Bubble	Yes	Yes	No
Selection	Yes	No	No
Insertion	Yes	Yes	Yes
Quick	Yes	No	No
Merge	No	Yes	No
Heap	Yes	No	No

Q5) Write recursive/iterative pseudocode for binary search. What is the time and space complexity of linear and binary search?

Iterative-

```
int Binary(int a[], int x)
{
 int low=0, high= a.length-1;
 while(low<=high)
 {
  int mid=(low+high)/2;
  if(x==a[mid])
  return mid;
  else if(x<a[mid])
  high=mid-1;
  else
  low=mid+1;
 }
 return -1;
}
```

Recursive-

```
int Binary(int a[], int l, int h, int x)
{
 if ((l>h)
   return -1;
 int mid=((l+h)/2;
 if(x==a[mid])
 return mid;
else if(x < a[mid])
 return Binary(a, [redacted], mid-1, x);
else
 return Binary(a, mid+1, [redacted], x);
}
```

Time Complexity ⇒ Iterative – O(log n)
Recursive – O(log n)

Space Complexity ⇒ Iterative – O(1)
Recursive – O(log n)

Q6) Write recurrence relation for binary recursive search.

Recurrence Relation – $T(n) = T(n/2) + 1$

Derivation –

$$T(n) = T(n/2) + 1$$
$$T(n/2) = T(n/4) + 1 \quad \ldots \quad T(n/4) = T(n/2^2)$$
$$T(n/4) = T(n/8) + 1 \quad \ldots \quad T(n/8) = T(n/2^3)$$
$$\vdots$$
$$T(n/2^{k-1}) = T(n/2^k) + 1 \ (k\text{-times})$$

Adding all equations,

$$T(n) = T(n/2^k) + k$$
$$\frac{n}{2^k} = 1$$
$$n = 2^k$$

$$\log n = k$$
$$T(n) = T(1) + \log n$$
$$T(n) = O(\log n)$$

Q7) Find two indexes such that A[i]+A[j]=k in minimum time complexity.

```
vector <int> find(a[],k,n)
{
 vector<int> sol;
 for i=0 to n-1
   for j=0 to n
     if a[i]+a[j]=k
      sol.pushback(i)
      sol.pushback(j)
      return sol
```

Q8) Which sorting is best for practical uses? Explain.

Quick sort is the fastest general purpose sort. In most practical situations, quick sort is a method of choice. If stability is important and space is available, merge sort might be best. In some performance critical applications, the focus maybe on just sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead.

Q9) What do you mean by number of inversions in an array? Count the number of inversions in arr={7,21,31,8,10,1,20,6,4,5} using merge sort.

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

Inversions ⇒ (7,1)(7,6)(7,4)(7,5) (21,8)(21,10)(21,1)(21,20
(21,6)(21,4)(21,5)(31,8)(31,10)(31,1)(31,20)(31,6
(31,4)(31,5)(8,1)(8,6)(8,4)(8,5) (10,1)(10,6)(10,4)

(10,5)(20,6)(20,4)(20,5)(6,4)(6,5)
Total inversions = 31

Q10) In which cases quick sort will give the best and the worst case time complexity?

Best Case - The best case occurs when the partition process always picks the middle element as pivot. Following is the recurrence relation for the best case,

$$T(n) = 2T(n/2) + O(n)$$

Worst Case - The worst case occurs when the partition process always picks greatest or smallest element as pivot. Its above partition process is considered where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing/decreasing order.

Q11) Write recurrence relation of merge and quick sort in best and worst case. What are the similarities and differences b/w complexities of two algorithms and why?

Quick Sort ⇒ Recurrence Relation - Best - $T(n) = 2T(n/2) + O(n)$
Worst Case - $T(n) = T(n-1) + O(n)$

Merge Sort ⇒ Recurrence Relation - Best - $2T(n/2)+O(n)$
Worst Case - $2T(n/2)+O(n)$

Time Complexity ⇒

	Best Case	Worst Case
Quick Sort	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$

Q12) Selection sort is not stable by default but can you write a version 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];
  while(min>i)
  {
   a[min]=a[min-1];
   min--;
  }
  a[i]=key;
}}
```

Q13) Bubble sort scans whole array even when array is sorted. Can you modify the bubble sort so that it does not scan the whole array once it is sorted.

```
void bubble(int a[],int n)
{
 int c,temp;
 for(int i=0;i<n-1;i++)
 {
  c=0;
  for(int j=0;j<n-i-1;j++)
  {
   c=1;
   if(a[j]>a[j+1])
   {
    temp=a[j];
     a[j]=a[j+1];
    a[j+1]=temp;
   }
  }
  if(c==0)
  break;
 }
}
```

Q14) Your computer has a RAM of 2GB and you are given an array of 4GB of sorting. Which algorithm are you going to use for the purpose and why? Also explain the concept of external and internal sorting.

As the size of the given array exceeds the size of RAM, we will use k-way merge sort technique as it takes a part of array and sort it instead of loading the whole array into main memory.

External sorting - This algorithm loads a part of array and sort it instead of loading the whole array in RAM. It is used to sort array of large sizes.

Internal sorting - This algorithm needs whole array to be loaded into RAM during execution.