Assignment 2

Q1

Sol Pseudo Code for insertion sort-

```
#include <iostream>
using namespace std;

int binarysearch (int arr[], int key, int size){

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

Q2 (i) Insertion sort iterative

```
#include< iostream>
using namespace std;

void insertion-iteration (int arr[], int size) {
    for(int i=1 ; i< size ; i++) {
        int j = i-1;
        while (j>0 && arr[j] >key){
```

```
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
  }
}
```

(ii) Insertion sort recursive :-

```
Void insertion-recursive (int arr[], int n){

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

Insertion sort is something called an online sorting algorithm "because it can sort list of elements as they are being recieved one at a time, without having to wait for entire list to be recieved or procesed first.

Q3

complexity of all sorting algorithm

	Best Case	Avg.- Case	Worst case	Space complexity
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n \cancel{\log n})$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(n \lg n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort		$O(n \lg n)$	$O(n^2)$	$O(n)$
Heap sort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$

Q4

(1) Inplace: sorts the input array by rearranging the elements within the array itself. for e.g

- Bubble sort
- selection sort
- Insertion sort

(2) Stable : Pressure the relative order of equal elements in the array itself. Elements with the same value are sorted in same order -

- Bubble sort
- Insertion sort
- Merge sort
- Count sort.

(3) Online : Sorts the stream of elements as they arrive.

- Insertion sort.

Q5 Recursive code for binary search

```
int binary (int arr [], int l, int r,
            int x) {

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

Iterative code for binary search

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

	Best case	Avg. Case	Worst case	space complexity
(1) Binary Search (Recursive)	O(1)	O(log n)	O(log n)	O(log n)
(2) Binary (Iterative)	O(1)	O(log n)	O(log n)	O(1)
(3) Linear Search (Recursive)	O(1)	O(n)	O(n)	O(n)
(4) Linear Iterative.	O(1)	O(n)	O(n)	O(1)

Q6 The recurrence relation expresses the time complexity of binary search algorithm into terms of its sub-problems. The algo divides the input array in half each iteration & solve a sub problem of size n/2

$$T(n) = T(n/2) + O(1)$$

where,

$T(n) \rightarrow$ array size is n (Time Complexity)

$T(n/2) \rightarrow$ array size is n/2 (" ")

$O(1) \rightarrow$ is the times complexity for comparing middle elements to target element.

Q7 The following algorithm is suitable for the desired task –

Step 1: Start the input array in non-decreasing order.

Step 2: Initialize true pointers i & j, to point to the first & last elements of the array respectively.

Step 3: while i<j, compute A[i] + A[j]

Step 4: if sum == k, return i & j.

Step 5: If sum < k, increment i by 1.

Step 6: If sum > k, decrement i by 1.

~~Step~~

The time complexity of above algorithm is O(n).

Q8

Quicksort is widely used sorting algorithm that has an average time complexity of

O(n log n) & is often faster than others popular sorting algorithms. Quicksort is particula -rly efficient for lar datasets & cans be easily implemented in place to save memory. However, its worst case time complexity is $O(n^2)$ which can occurs when the input data is already sorted.

Q9 To count number of inversions in the given array {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

```
#include <iostream>
using namespace std;
int getInvcount (int arr[], int n) {
        int inv_count = 0;
        for (int i=0 ; i<n-1 ; i++)
            for (int j= i+1 ; j<n; j++) {
                if (arr[i] >arr[j])
                    inv_count ++;
        }  }
        return inv_count;
}
```

The number of steps it will take for array to be sorted or how far away any array is from being sorted. it is the inversion count for any array.

Q10

* Best case → The pivot element chosen should be the median of the array. If the pivot is chosen as the median at each step, then the partioning step will divide the array into two sub-arrays of equal size, resulting in balanced tree of recursive calls, In this case, the time complexity of Quick

sort is $O(\log n)$.

* Worst Case → In worst case the pivot element chosen at each step is either largest or smallest element with sub-array.
Time Complexity will be $O(n^2)$.

Q11 Recurrence Relation for merge sort

Best Case → $T(n) = 2T(n/2) + O(n)$
Worst case → $T(n) = 2T(n/2) + O(n \log n)$

Recurrence relation for Quick sort
Best Case → $T(n) = 2T(n/2) + O(n)$
Worst Case → $T(n) = T(n-1) + O(n)$

The difference between the time complexities of merge & Quick sort are that merge sort has worst case time complexity of $O(n \log n)$ & Quick sort has the worst case time complexity of $O(n^2)$.

Q12 Yes, It is possible to implement the stable version of selection sort.

```
#include <iostream>
using namespace std;

Void selectionsort (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 [j] < arr [min]) {
                min =j;
        }
    }
    int temp = arr[i];
    arr [i] = arr[min];
    arr[min] = temp;
  }
}
```

In the above code, we maintain a key variable to hold the value of min element found in the inner loop.

Q18 Yes, we can modify the bubble sort algorithm to optimize it so that it does not scan the entire array once it is already sorted.

```
#include<iostream>
Using namespace std;

Void bubblesort (int arr[], int n) {
        bool swapped;
        for (int i=0; i<n; i++) {
            swapped = false;
        for (int j=0 ; j< n-i-1 ;j++) {
            swap (arr [j], arr [j+1]);
            swapped = true;
        }
    }
    if (! swapped) {
```

```
        break;
      }
    }
}
```

Q14

It is not possible to load the entire array into the memory for sorting using internal sorting. In this case we would need to use external sorting algorithms that operate on disk instead of memory.

External sorting is the technique used to sort large data sets that can not be held in memory at once. It involves a combination of internal & external sorting techniques.

The most commonly used external sorting algorithm is external merge sort. In this algorithm the data is split into smaller parts that can fit into memory & each chunk is sorted using an internal sorting, such as Quick sort or heap sort. The sorted chunks are then merged together in a servis of passes, wher the data is read from the disk, merged & written back to disk.