Date.
Page No.

Assignment - 02

Q1)

```
int Search (vector <int> & array, int key)
{
    int l = array.size();
    if (l = = 0){
        return -1;
    }
    for (i = 0; i < len; i++)
    {
        if (array [i] == key)
        {
            return 1;
        }
        if (array [i] > key)
        {
            return -1;
        }
    }
```

Q2) ① Recursive :

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

Date.
Page No.

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

② Iterative :

```
void Insertion (vector <int> arr)
{
    int n = arr.size();
    for (i = 0; i < n; i++)
    {
        int K = arr[i];
        int j = i-1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}.
```

The reason for insertion sort being called online sort algorithm is that it can simultaneously sort the value which are constantly being added to the vector or array.

None of the other sorting algorithm (Bubble, selection, heap, quick or merge) can work as effectively as insertion sort when it comes to constantly updating data.

Date.
Page No.

Q3)

	Best	Avg	Worst	(Best) Time comp	Space comp.
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(\log n)$
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Heap	"	"	"	"	$O(1)$
Count	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$

Q4)

	Inplace	Stable	Online sort
Bubble	✓	✓	X
Selection	✓	✓	X
Insertion	✓	✓	✓
Quick	✓	✓	X
Merge	X	✓	X
Heap	✓	✓	X
Count	✓	x	X

Date.
Page No.

Q5)

(i) Recursive

```
int Recursive( vector <int> &arr, int key, int low, int high)
{
    if (low > high)
    {
        return -1;
    }
    int mid = low + (high - low)/2;
    if (arr [mid] == key)
    {
        return mid;
    }
    else if (arr [mid] > key)
    {
    recursive (arr, key, low, mid -1);
    }
    else
    {
        recursive(arr, key, mid+1, high);
    }
}
```

(ii) Iterative

```
int Recursive (vector <int) &arr, int Key, int low, int high)
{
while ( low <= high)
{
    int mid = low + (high - low)/2;
```

```
Date.
Page No.
```

```
        if (arr [mid] = = key)
        {
            return mid;
        }
        else if (arr[mid] > key)
        {
              high = mid - 1;
        }
        else
              low = mid + 1;
        }
    }
    return -1;
}
```

Q6)

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

Q7)

```
void search (int arr[], int low, int high, int k)
{
        int sum = 0;
        sort (arr.begin (), arr.end ());
        while ( low <= high)
        {
            if ( arr( low) + arr( high ) == k)
            {
                cout << i << j;
                break;
            }
            else if (arr (low] + arr [high] > k)
            {   high = high - 1;
```

Date.
Page No.

```
    else {
            low = low + 1;
        }
    }
}
```

Q9) Total inversions = 31.

Q8) Based on what kind of practical work is being done the best sorting algorithm can be chosen accordingly.

(i) Quick Sort
- Very fast sorting algo.
- works well with large dataset
- works well when dataset is distributed
- Have avg time complexity of $O(n \log n)$.
- In place in nature

(ii) Insertion Sort
- Works well for simultaneously occuring data.
- Inplace and stable in nature.
- efficient with small or nearly sorted datasets

Q10) (i) Base Case
- $O(n \log n)$ time complexity
- If the pivots position is at half of the array which leads to equal balanced arrays making the performance of this algorithm to become optimal

(ii) Worst Case
- $O(n^2)$

Date.
Page No.

- vice versa of above
  when the sub arrays are of different size
  Performance drops.

Q11) Merge Sort

(i) Best Case

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

(ii) Worst Case.

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

Quick Sort

(i) Best Case

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

(ii) Worst Case

$$T(n) = T(n-1) + O(n).$$

Similarities

(i) Best case time complexity is same

(ii) Division of array into sub array is done.

(iii) Both work on "Divide & conquer" approach.

Difference

(i) Merge Sort is more stable than Quick since the mid is present at middle instead of pivot being random.

(ii) Merge sort is not inspace.

12
Q10)

```
void selection( vector <int> & arr)
{
    int n = arr.size ();
    for( int i=0 ; i<n-1 ; i++)
    {
        int min = i;
        for( j= i+1; j<n ; j++)
        {
            if (arr[j] < arr [min])
            {
                min = j;
            }
        }
        int min_value = arr[min];
        while (min > i) {
            arr[min] = arr[min -1]
            min --;
        }
    }
}
```

3
Q10)

```
void BubbleSort (vector <int> & arr)
{
    int n = arr.size();
    bool swapped = true;
    while ( swapped )
    {
        swapped = false;
        for( int i=0; i< n-1; i++)
        {
            if ( arr [i] > arr (i+1])
```

```
{
    swap (arr [i], arr [i+1]);
    swapped [ = true;
}
}
n--;
if (!swapped)
{
  break;
}
}
}.
```

Q1(0)4) Merge Sort is preferred choice for sorting a 4GB array on a computer with limited RAM capacity due to its efficiency, minimized memory usage scalability, stability & predictable performance.