Ans 1

Linear Search -

```
for i = 0 to Array.length - 1

    if Array[i] == target
        return i
    end if

    if Array[i] > target
        break
    end if
    i = i + 1
end for

return -1
```

Ans 2 Iterative -

```
for i = 1 to Array.length - 1

    current = Array[i]
    position = i
    While position > 0 AND Array[position - 1] > current
        Array[position] = Array[position - 1]
        position = position - 1
    end while

    Array[position] = current
    i = i + 1
end for
```

Recursive -

```
Void insertionSort (int arr[], int n)
if (n <= 1)
  return

insertionSort (arr, n-1)

val = arr[n-1]
pos = n-2

while pos >=0 AND arr[pos] > val
      arr [pos+1] = arr[pos]
      pos = pos -1
end while

arr [pos+1] = val
```

Insertion sort is called online sorting because it does not need to know anything about any values it will sort and the information is requested while algorithm is running.

Date ___/___/_____ Page No.: ___

Ans 3

Bubble Sort - $O(N^2)$

Insertion Sort - $O(N^2)$

Selection Sort - $O(N^2)$

Merge Sort - $O(N * \log N)$

Quick Sort - $O(N * \log N)$

Count Sort - $O(N+k)$

Heap Sort - $O(N \log N)$

Ans4 Inplace Sorting -

Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Quick Sort

Stable Sorting -

Merge Sort, Insertion Sort, Bubble Sort

Online Sorting -

Insertion Sort

Ans5 Iterative -

```
Low = Lbound (Array)
high = Ubound (Array)

Do while low <= high
    middle = (low + high) / 2
    if target = Array (middle)
        found = true
        Exit DO

    Else if Target < Array (middle)
        high = middle - 1

    Else
        low = middle + 1
    End if
```

Recursive -

```
bool binarySearch (array, start, end, target)

    While Starte < end

        mid = (start + end)/2

        If target == array[mid]
            return mid

        Else if target > array[mid]
            return binarySearch (array, mid+1, end, target)

        Else
            return binarySearch (array, start, mid-1, target)

        End while
End function
```

Binary Search -
Time Complexity = O(logN)
Space Complexity -

Recursive - O(logN) Iterative - O(1)

Linear Search -
Time Complexity = O(N)
Space Complexity = O(1)

Date ___/___/_____

Ans 6 Recurrence Relation →

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

Ans 7

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

Void find (int nums[], int n , int target)
{
  Unordered_map <int, int> map;
  for (int i=0; i<n ; i++)
  {
    if (map.find (target-num[i] != map.end())
    {
      cout << "Pair :" << nums [map [target-nums[i]] << nums[i];
      return ;
    }
  map[nums[i]] = i;
  }
  cout << " Pair not found" ;
}
```

Time complexity → O(n)

Page No.:

Ans 8 Quick Sort is used for practical uses. It is one of the quickest algorithms. and It is an in place algorithm since it requires a modest auxiliary stack.

⇒ Sorting n objects takes only n(log n) time.
⇒ It's inner loop is relatively short
⇒ This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Ans 9

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

For arr[] = { 7, 21, 31, 8, 10, 1, 20, 6, 9, 5 }

Using merge sort, we will have 31 inversions

Ans 10

1 Best time complexity of Quick sort is O(N log(N)) and it takes place when we select pivot as mean element.

2 Worst time complexity is $O(N^2)$. It occurs when the pivot is always an extreme element, happens when array is sorted or reverse sorted and either first or last is picked as pivot.

Ans11 Quick Sort

Worst case -

$$T(N) = N + T(N-1)$$

Best case -

$$T(N) = 2T(N/2) + N$$

Merge Sort

Worst case -

$$T(N) = kT(N/k) + \text{O(n)}\, n \log k - k + 1$$

Best case -

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

Quick sort has a best linear performance when the input is sorted or nearly sorted. It has worst-case quadratic performance when input is sorted in reverse or nearly sorted in reverse. Merge sort performance is much more constrained and predictable than the performance of quicksort. Because of that reliability avg. case of merge sort is slower than avg. case of quick sort. A good merge sort implementation will have better avg. performance.

Ans 12 Stable selection Sort →

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

Void Sort (int arr[], int n){

  for (int i=0 ; i<n ; 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;
  }
}
```

Ans 13

```
void sort(int arr[], int n){
  int flag = 0

for(int i=0; i<n-1; i++){
      flag = 0;
  for(int j = 0; j<n-1-i; j++){

      if(arr[j] > arr[j+1]){
          swap(arr[j], arr[j+1]);
          flag = 1;
      }
  }
  if(flag == 0) break;
}

}
```

Ans 14 For this purpose we will use external sorting algorithm.

Reason - Since the data being sorted do not fit into the main memory and they must reside in the slower external memory. Hence external sorting is the only possible way.
External sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. The smaller dataset one usually sorted using the merge sort algorithm.

Internal Sorting -

Internal sorting are type of sorting which is used when the entire collection of data is small enough that sorting can take place within main memory. There is no need for external memory for execution of sorting program. Ex - bubble sort, insertion sort.