Assignment - 2

1.
```
int search (const vector <int>& arr, int target) {
    int n = arr.size()
    if (n = =0) {
        return -1
    }
    int index = -1
    for (int i = 0; i<n ; i++) {
        if (arr [i] = = target) }
            index = i;
            break;
        else if ( arr[i] > target) }
            break;
        }
    }
    return index;
}
```

2. Iterative :

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

Date.
Page No.

Recursive :

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

    a[j+1] = key;
}
```

It is called online sort because it works by taking one element at a time and inserting it into its correct position relative to the elements that have already been sorted.

3. Bubble sort : $O(n^2)$
Selection sort : $O(n^2)$
Insertion sort : $O(n^2)$
Merge sort : $O(n \log n)$
Quick sort : $O(n \log n)$
Heap sort : $O(n \log n)$
Count sort : $O(n+k)$
Radix sort : $O(n * k)$

4.

Algorithms	Inplace sort	Stable sort	Online sort
Bubble sort	✓	~~X~~ ✓	X
Selection sort	✓	X	X
Insertion sort	✓	X	✓
Quick sort	✓	X	X
Heap sort	✓	X	X
Count sort	X	✓	X
Radix sort	~~X~~ ✓	X	X
Merge sort	✓	✓	X

5. Recursive:

```
int search (const vector<int>& a, int target, int left, int right){
        if (right >= left) {
            int mid = left +(right -left))/2;
            if (a[mid] == target)
                return mid;
            if (a[mid] > target)
                return search( a, target, left, mid-1);
            return search(a, target, mid+1, right);
    }
    return -1
  }
```

Iterative:

```
int search( const vector<int>& a, int target) {
    int left =0;
    int right = a.size()-1;
    while (left <= right) {
        int mid = left + (right - left)/2;
        if (a[mid] == target) {
            return mid
        }
```

Date.
Page No.

```
if (a[mid] > target)
        right = mid - 1;
    else :
        left = mid + 1;
    }
  return -1
}
```

	Time complexity	Space complexity
Interative	O(log n)	O(1)
recursive	O(log n)	O(log n)

6. The search space in halved every time target is not found and algorithm continues searching. In best case scenario it can be expressed as $T(n) = O(1)$

So relation : $T(n) = T(n/2) + O(1)$

7.
```
pair <int, int > find (vector <int> & a, int k) {
    sort (a.begin (), a.end () ):
    int left = 0;
    int right = a. size () - 1;
    while ( left < right) {
        int sum = a[left] + a[right] ;
        if (sum == k) {
            return {left, right}
        }
        else if ( sum < k) {
            left ++; }
        else {
            right++; }
```

8. The choice of sorting algorithm often depends on the specific requirement of the application and characteristics of the input data.

- Quicksort is used as default sorting algorithm in many programming languages & libraries.
- Merge sort is preferred when stability is required or when the data is stored in external memory
- Heapsort is often used in priority queue implementations
- Insertion sort is often used as part of more complex algorithm or in hybrid sorting.

9.
```
int merge (vector<int>& a, int low, int mid, int high){
    vector<int> temp(high - low +1);
    int inversion count = 0;
    int i = low;
     int j = mid +1;
    int k = 0;
    while (i<=mid && j<=high) {
        if (arr[i]<= arr[j]) {
            temp[k++] = a[i++];
        else {
            temp [k++] = a[j++];
            inversion count += mid - i +1
        }
    }

    return inversioncount;
}
```

```
int mergesort (vector<int>& a, int low, int high) {
        int inversioncount = 0;
        if (low < high) {
            int mid = low + (high - low) / 2
            inversioncount += mergesort (a, low, mid);
            inversioncount += mergesort (a, mid+1, high);
            inversioncount += merge (a, low, mid, high)
        }
        return inversioncount;
}
```

10. Best - time complexity:

$O(n \log n)$

This happens when pivot element chosen is the median element or approximately the median element of array.

Worst case complexity:

$O(n^2)$

This happens when the pivot element is either the smallest or the largest.

11. Recurrence in merge sort:

Best case: $T(n) = 2T(n/2) + O(n)$

Worst case: $T(n) = 2T(n/2) + O(n)$

Recurrence in quick sort:

Best case : $T(n) = 2T(n/2) + O(n)$

Worst case: $T(n) = T(n-1) + O(n)$

Similiarities between Merge & Quick sort

- Both algorithms have same recurrence relation in best case $T(n) = 2T(n/2) + O(n)$
- Both algorithm have a time complexity of $O(n \log n)$ in best case.

Difference between Merge & quick sort

- Space complexity : Merge sort requires additional space for merging process, while quick sort can be implemented
- Stability : Merge sort is a stable sorting algorithm, compared to quick sort

12.

```
void selection( vector<int>& a, int n) {
    for(int i =0; i<n-1 ; i++) {
        int minIndex = i ;
        for (int j = i+1 ; j<n; ++j) {
            if (a[j] < a[minIndex]) {
                minIndex = j}
            }
        }
    int minvalue = a[minIndex]]
    while( minIndex >i) {
        a[minIndex] = a[minIndex] -1 ];
        minIndex --;
    }
    arr[i] = minValue)
    }
}
```

13.
```
void bubblesort( vector<int>& a) {
    int n = a.size() ;
    bool swapped ;
    for (int i = 0 ; i < n-1 ; i++) {
        swapped = false;
        for( j = 0; j < n-i-1; j++) {
            swap (arr[j], arr[j+1]) ;
            swapped = true;
        }
    }
    if (!swapped) {
        break;
    }
  }
}
```

14. Merge sort will be used for such problem since it is an external sorting algorithm

External sorting:
- It deals with sorting that cannot fit entirely into available ~~memory~~ memory.
- These are designed to minimize the number of disk I/O operation required to read & write data during sorting process

Internal sorting:
- It is a sorting that accomodates entirely within the available memory of computer.
- These algorithm operate entirely within the memory making them effective.