Assignment - 2

1.
```
bool linearSearch (int* arr, int key, int size)
{
    for (int i = 0; i < size && arr[i] <= key ; i++)
    {
        if (arr[i] == key)
            return true;
    }
    return false;
}
```

2.
```
void insertionSort(int* arr, int n)
{
    for (int i = 1 ; i < n-1; i++)
    {
        k = arr[i];
        j = i - 1
        while ( arr [j] > k && j >= 0)
        {
            arr [j+1] = arr [j]
            j--;
        }
```

```
    arr[j+1] = k;
  }
}
```

Recursive

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

Insertion sort is called online sorting because we don't require the ~~whole~~ whole array instead ~~of~~ ~~the~~ ~~num~~ ~~number~~ it can work even if the no. of elements keep increasing. But other algorithms like selection and bubble sort require the whole array.

3\.

Bubble	Selection	Insertion
$B \to O(n)$	$B \to O(n^2)$	$B \to O(n)$
$A \to O(n^2)$	$A \to O(n^2)$	$A \to O(n^2)$
$W \to O(n^2)$	$W \to O(n^2)$	$W \to O(n^2)$

Merge

B → O(n log n)
A → O(n log n)
W → O(n log n)

Quick

B → O(n log n)
A → O(n log n)
W → O(n log n)

Count

B → O(n+k)
A → O(n+k)
W → O(n+k)

4. Inplace Sort : Bubble, Selection, Insertion, Quick, heap.
Stable : Insertion, Selection, Merge
Online: Insertion

5. Iterative

```
bool binarySearch(int*arr, int n, int key)
{
        int low =0 , up= n-1;
        while (low<= up)
        {
          int mid = low + (up-low)/2;
          if (arr[mid]== key)
                return true;
          else if (arr[mid] < key)
                low = mid+1;
          else
                up = mid-1;
        }
        return false;
}
```

Recursive

```
bool ~~void~~ binarySearch (int* arr, int low, int up)
{
    if (low > up)
        return false;
    int mid = low + (up - low)/2;
    if (key > arr[mid])
        ~~low = mid + 1~~; binarySearch (arr, mid+1, up);
    else if (key < arr[mid])
        ~~up = mid - 1~~; binarySearch (arr, low, mid);
    else
        return true;
}
```

T.C
$O(\log n)$

S.C
$O(1)$

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

Quicksort is best for practical use. It is considered as one of the fastest sorting algorithms for avg. case. It's average case time complexity is $O(n \log n)$ and worst case is $O(n^2)$ but occurs rarely as the pivot element is chosen randomly every time. Also, it is an in-place sort.

9. In an array, an inversion occurs when two elements are not in sorted order.

```
int mergeSort (int arr[], int temp [], int l, int r)
{
    int mid, inv=0;
    if (r>l){
        mid =(r+l)/2;
        inv = inv + merge sort(arr, temp, l, mid);
        inv = inv + mergeSort (arr, temp, mid+1, r))
        inv = inv + merge(arr, temp, l, mid+1, r);
    }
    return inv;
}

int merge (int arr[], int temp[], int l, int mid, int r)
{
    int i, j, k;
    int inv=0;
    i=l;
    j= mid;
    k=l;
    while ((i<= mid-1) && (j<=r))
    {
        if (arr[i]<= arr[j])
            temp[k++] = arr [i++];
        else
        {
            temp[k++] = arr [j++];
            inv += mid-i+1;
        }
    }
    while (i<=mid-1)
        temp [k++] = arr [i++];
    while (j<= r)
        temp [k++] = arr [j++];
```

return inv;
}

```
int mergeS(int arr[], int size)
{
    int temp[size];
    return mergesort(arr, temp, 0, size-1);
}
```

18. The best case is when the pivot element divides the array into two equal halves ($T.C = O(n \log n)$)

• The worst case is when the pivot element divides the array into two unbalanced halves. ($T.C = O(n^2)$).

19. Merge Sort :

<u>Best Case</u> : $T(n) = 2T(n/2) + O(n)$

<u>Worst Case</u> : $T(n) = 2T(n/2) + O(n)$

<u>Quick Sort</u>

<u>Best</u> : $T(n) = 2T(n/2) + O(n)$

<u>Worst</u> : $T(n) = T(n-1) + O(n)$

<u>Similarities</u>

- Both algo. work on divide and conquer strategy.
- Both algo. have best and worst case time complexity of $O(n \log n)$.

<u>Differences</u>

- Merge sort is more efficient on larger arrays.
- Merge sort requires extra space proportional to the size of input array, whereas quicksort is an in-place sort.

12.
```
void selection (int arr[])
{
        int n = arr.size();
        int i = 0;
        for (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 m = arr[min];
            while(min>i)
            {
                arr[min] = arr[min-1];
                min--;
            }
            arr[i] = m;
        }
}
```

13.
```
void bubble(int arr[])
{
    int n= arr.size(), i, j=0;
    bool swap;
    for(i=0; i<n-1; i++)
    {
        swap = false;
        for (j=0; j<n-1-i; j++)
        {
            if (arr[j]>arr[j+1])
            {
                swap(arr[j], arr[j+1]);
            }   swap = true;
        }
```

```
if (!swap)
{
    break;
}
}
}
```