Assignment 2

1.

```
void linear_search (int arr [], int key, int n)
{
    for (int i = 0; i<n ; i++ )
    {
        if (a[i]==key){
            return i ;
            break ;
        }
    }
    return -1
}
```

2. for iterative insertion sort -

```
void iterative_insert (int a[], int n)
{
    for (int i=1 ; i< n ; i++)
    {
        int t = a[i] ;
        int j=i-1 ;
        while (j >=0 && a[j]>t )
        {
            a[j+1]=a[j] ;
            j-- ;
        }
        a[j +1] =t ;
    }
}
```

for recursive insertion sort -

```
void recursive_insert (int a[], int n )
{
    if (n<=1)
        return
    else
    {
        recursive_insert( a ,n-1
        t = a[n-1];
        j = n-2
        while (j >=0 && a[j] > t )
        {
            a [j +1] = a [j] ;
            j --;
        }
        a[j +1] =t ;
    }
}.
```

Insertion sort is called an online sorting algo because it bulds the sorted list one element at a time. At any given point during the sorting, the elements to the left of the current element are already sorted. This makes it suitable for situations where elements are continuously arriving in the input stream & we want to maintain a sorted list dynamically.

Merge sort and Quick sort are not typically considered online sorting algo because they're not designed to efficiently handle elements arriving incrementally. Bubble and heap sort can be adapted to work in and online fashion by using appropriate data structures. Selection sort is not an inherently online sorting algo, as the entire input has to be present before sorting

3.

Algorithm	Complexity best	avg	worst
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2 \cancel{\log n})$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Count	$O(n)$	$O(n+K)$	$O(n+K)$
Radix	$O(d(n+K))$	$O(d(n+K))$.	$O(d(n+k))$

4.

Inplace Sorting	Online Sorting	Stable Sorting
Insertion Sort	Insertion Sort	Bubble sort
Selection Sort	Bubble Sort	Insertion sort
Bubble Sort	Heap Sort	Merge sort
In-place Merge Sort		Count sort
		Radix sort

5. for recursive binary search

```
bool binarysearch (int a[], int l, int r, int key)
{
          if (l > r)
              return false;
          int mid = l + (r-l)/2;
O(1) ____ if (a[mid] == key)
                return true;
          else if (a[mid] > key)
O(n/2) ______ binarysearch (a, l, mid -1, key);
O(n/2) ______ binarysearch (a, mid +1, r, key);
}
```

for iterative binary search

```
bool binarysearch (int a[], int l, int r, int key)
{
     while (l <= r)
     {
          int = l + (r-1)/2;
          if (a[mid] == key)
               return true;
```

```
else if (a[mid] > key)
        r = mid - 1;
    else
        l = mid + 1;
  }
  return false;
}
```

Liniar search :- T.C = O(n) ~~best~~(avg, worst), O(1)(best)
S.C = O(1)

Binary search :- recursive :
T.C best - ~~$O(\log_2 n)$~~ $O(1)$
worst - $O(\log_2 n)$
avg - $O(\log_2 n)$

S.C - $O(\log_2 n)$

iterative
T.C - best - $O(1)$
worst - $O(\log_2 n)$
avg - $O(\log_2 n)$

S.C - O(1).

6. Recursive relation for binary recursive search

T(n) - function declaration
O(1) - if condition
O(n/2) - recursive call for right
O(n/2) - recursive call for left

$$\boxed{T(n) = 2\,T(n/2) + 1}$$

7.

```
#include <iostream>
using namespace std;
#define max 10
void sum (int *i, int *j, int *k, int a[], int *f)
{   int m, n;
    for (m = 0; m < (*i); m++)
    {
        for (n = m + 1; n < (*j); n++)
        {
            if ((a[m] + a[n]) == a[k])
            {
                cout << m << " " << n << " " << (*k) << endl;
                f++;
                break;
            }
        }
    }
}
```

```
int main()
{
    int a[max];
    int n, t, i, j, k, l, f = -1;
    cout << "Enter no. of elements" << endl;
    cin >> n;
    f = 0;
    cout << "Enter elements" << endl;
    for (j = 0; j < n; j++)
    {
        cin >> a[j];
    }
    i = n, j = n;
    for (k = n-1; k > 0; k--)
    {
        sum(&i, &j, &k, a, &f);
        i--, j--;
        if (f != 0)
            break
    }
    if (f == -1)
    {
        cout << "no sequence found";
    }
    }
    return 0;
}
```

8. Which sorting is best for practical uses ? Explain.

The sorting which is best for practical use is Quick sort

T.C - $O(n \log n)$ - avg, best

$O(n^2)$ - worst

Inplace : Yes

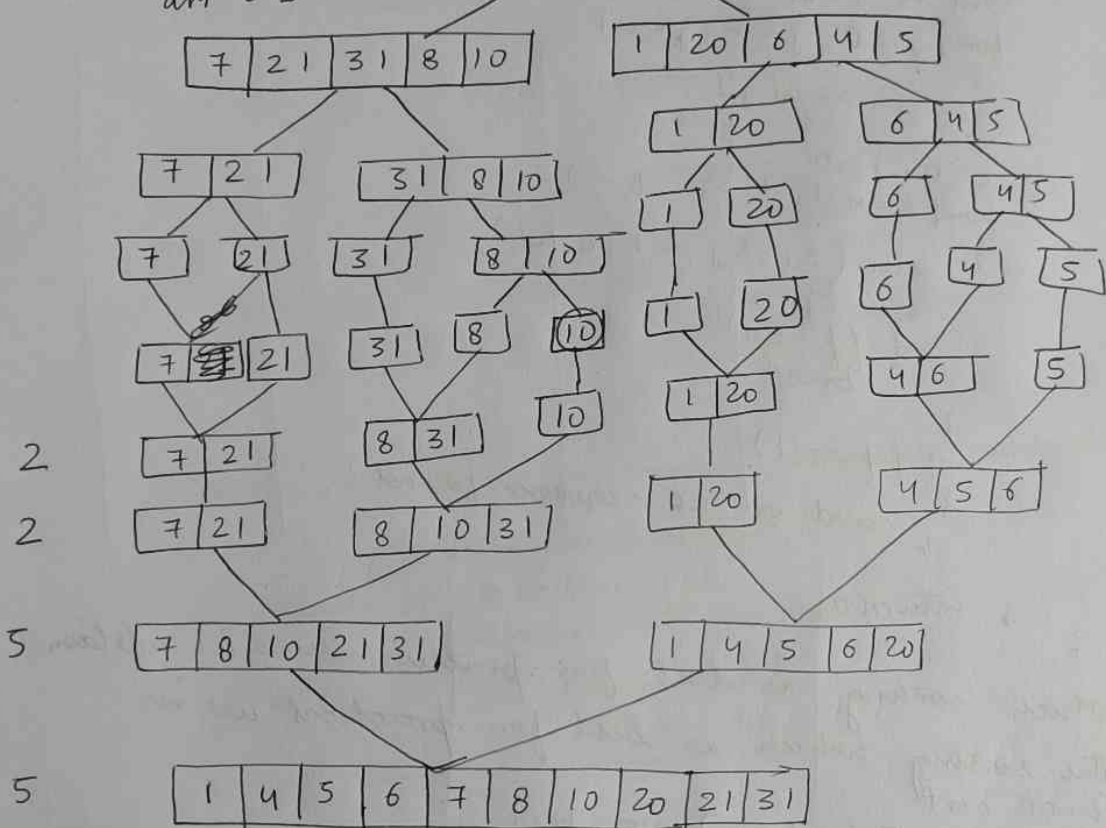
Stable - NO.

Practical Consideration : Quick sort is often the best choice for general purpose sorting due to its average case time complexity of $O(n \log n)$. Its widely used in libraries and frameworks like C++ & Java. ~~Java~~ E However, its worst-case time complexity might be a concern for specific scenarios and case must be taken to avoid worst-case behaviour

Overall, Quick sort strikes a balance between efficiency, simplicity and practicality making it is popular choice for sorting large datasets in real-world applications.

9. An inversion occurs when two elements in the array are out of their sorted order. More formally, in an array 'arr' if there are two indices 'i' and 'j' such that 'i < j' but 'arr[i] > arr[j]' then the pair (arr[i], arr[j]) is an inversion.

arr [ ] = { 7, 21, 31, 8, 10, 1, 20, 6, 4, 5 }



no. of inversions = 14

10. Quick sort has a best time complexity of $O(n \log n)$ and a worst case time complexity of $O(n^2)$ where n is the no. of elements in array.

Best case: ~~The best~~ case This occurs when pivot chosen at each step divides the array into two appropriately equal parts. In this case, each partitioning step divides the array in half, resulting in a balanced tree structure.

Worst case: When the pivot chosen at each step is either the smallest / largest element in the array. This results in highly unbalanced partitions with one partition containing all elements except the pivot and the other partition containing only the pivot. In this scenario, each partitioning step reduces the size of the array by only 1 element.

11.

Recurrance relation

Merge Sort

Best case ⇒ $T(n) = 2T(n/2) + O(n)$

Worst case ⇒ $T(n) = 2T(n/2) + O(n \log n)$

Quick sort

Best case ⇒ $T(n) = 2T(n/2) + O(n \log n)$

Worst case ⇒ $T(n) = T(n-1) + O(n)$

Similarities ⇒

→ both algorithms have divide and conquer approach.

→ ~~best case~~ complexity for both is $O(n \log n)$ for best and average.

Differences ⇒

→ quick sort has worst-case time complexity of $O(n^2)$ when pivot selection is poor or the input is sorted while merge sort maintains $O(n \log n)$ in all cases.

→ merge sort uses additional space proportional to the input size for merging step, making it less memory efficient compared to Quick Sort.

→ Quick sort is generally faster in practice due to its smaller constant factors and better cache performance.

12.
```
#include <iostream>
#include <vector>
using namespace std;
void stablesort (vector <int> arr) {
    int n = arr.size();
  for (int i = 0; i < n-1; i ++) {
      int minindex = i;
      for(int j = i+1; j<n; j++) {
          if (arr[j] < arr[ minIndex]) {
              minindex = j;
      }
  }.
int minValue = arr [minindex];
while (minindex > i) {
        arr [minindex] = arr [ minindex - 1];
        minindex --;
  }.
```

```
        arr[i] = min Value;
    }
}
int main(){
    vector<int> arr = { 4, 3, 5, 1, 2 }
    count << "Original array";
    for (int num : arr) {
        cout << num << " ";
    }
    stablesort (arr);
    cout << "Insorted array";
    for (int num : arr) {
        cout << num << " "; }
    return 0;
}
```

13.

```
#include <iostream>
#include <vector>
using namespace std;
void bubblesort (vector <int> arr) {.
int n = arr.size ();
bool swapped;
for (int i = 0; i < n-1; i++) {
    swapped = false;
    for (int j = 0; j < n - i - 1; j++)
    {
        if (arr[j] > arr[j+1])
        {
            swap (arr[j], arr[j+1])
            swapped = true;
        }
    }
    if (swapped)
        break;
}
} }
int main () {
    vector <int> arr = { 64, 34, 25, 12, 22, 11, 90 }
    cout << "Original array";
    for (int num : arr) {
        cout << num << " ";
    }
    bubblesort (arr);
    cout << "Sorted array";
    for (int num : arr) {
        cout << num << " ";
    }
    return 0;
}
```

14. Enternal sorting

class of algorithms designed for situations where data to be storted is too large to fit entirely into computer's main memory.

Internal sorting

refers to process of sorting data that fits entirely into computer's main memory (RAM)
suitable for relatively small dataset that can be accomodated in available memory.

Algorithm choice

Enternal merge sort

→ The data set is divided into smaller blocks that fits into available memory.

→ sorted blocks are written back to eternal storage

→ The sorted blocks are sequentially merged to produce the final sorted result.