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

2) Iterative →

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

Page No.
Date

Recursive →

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

In insertion sorting, by adding elements at last of the array, sorting is not affected. Therefore transition is online sorting algorithm.

Inplace sorting algo → Bubble sort, selection sort, insertion

Stable sorting algo → Bubble sort

External sorting algo → It may merge sort

Internal sorting algo → Bubble sort, insertion sort, selection sort.

Page No.
Date

3. Bubble sort →

Best case - $O(n^2)$
Avg case - $O(n^2)$
Worst case - $O(n^2)$
Space complexity - $O(1)$

Selection sort -
Time complexity → Best = $O(n^2)$
Avg = $O(n^2)$
Worst = $O(n^2)$
Space complexity → $O(1)$

Insertion sort -
Time complexity → Best $O(n)$
Avg $O(n^2)$
Worst $O(n^2)$
Space complexity → $O(1)$

Count sort
Time complexity = $O(n+K)$
Space complexity = $O(n+K)$

Quick sort
Time complexity → Best case = $O(n \log n)$
Avg = $O(n \log n)$
Worst = $O(n^2)$
Space complexity → $O(\log n)$

Merge sort
Time complexity → Best case - $O(n \log n)$

Average case = $O(n \log n)$
Worst case = $O(n \log n)$
Space complexity → $O(n)$

Heap sort
Time complexity → Best : $O(n \log n)$
Average : $O(n \log n)$
Worst : $O(n \log n)$

4)

Inplace	Stable	Online
• Bubble sort	• Bubble sort	• Insertion sort
• Selection sort	• Merge sort	
• Insertion sort	• Insertion sort	
	• Counting sort	

5) Iterative

```
int binarysearch ( int a[], int n , int key )
{
   int l = 0 , h = n-1 ;
   while ( l <= h )
   {
      int mid = (l+h) /2 ;
      if (a[mid] == key )
         return mid ;
      else if (a[mid] < key )
         l = mid + 1 ;
      else if ( a[mid] > key )
         h = mid -1 ;
   }
   return -1 ;
}
```

Time complexity = $O(\log_2 n)$
Best case → $O(1)$
Space complexity = $O(1)$

Recursive

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

Time complexity ⇒ $O(\log_2 n)$
Best case = O(1)
Space complexity ⇒ $O(\log_2 n)$

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

7)

```
# include <iostream>
# include < algorithm >
using namespace std ;
void find ( int a[], int n, int key )
{
  sort ( a, a+n ) ;
  int left = 0, right = n-1, f = 0 ;
  while ( left < right )
{
  int current sum = a[ left ] + a[ right ] ;
  if ( current sum == key )
{     cout << a[ left ] << a[ right ]
```

```
            f = 1;
            break;
        }
        else if ( current sum < key )
            left ++;
        else    right --;
    }
    if ( f == 0 )
        cout << " does not exist ";
}
```

8) Quick sort is best for practical use due to its average case time complexity of O(nlogn) which is efficient for most datasets. It also has good cache locality and can be implemented as in place algorithm minimizing auxillary space requirements.

9) The inversion count for an array is the number of steps it will take for the array to be sorted or how far away an array is from being sorted.

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

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 → 4

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 +1 +1 +1 +1 → 7

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 +1 +1 +1 +1 → 7

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 +1 → 4

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 +1 → 4

7 21 31 8 10 1 20 6 4 5 → 0

7 21 31 8 10 1 20 6 4 5
+1 +1 +1 → 3

7 21 31 8 10 1 20 6 4 5
+1 +1 → 2

7 21 31 8 10 1 20 6 4 5 → 0

no of inversions = 31

10) Worst case → occurs when the pivot element is either greatest or smallest element.

Best case → when we select the pivot as the mean.

$T(N) = 2 * T(N/2) + N \rightarrow$ constant

Now $T(N/2) = 2 * T(N/4) + N/2 \rightarrow$ constant

$T(N) = 2 * (2 * T(N/4) + N/2 * \text{constant}) + N \rightarrow$ constant
$= 4 * T(N/4) + 2 * \text{constant} * N$
we can say

$T(N) = 2^k * T(N/2^k) + k * \text{constant} * N$
then $2^k = N$
$k = \log_2 N$
so
$T(N) = N * T(1) + N * \log_2 N$

$T.C = O(N \log N)$

11) Merge sort

```
void sort ( int a[], int l, int r )   T(n)
{
    if (l < r)
    {
        int mid = l + (r-l) / 2;
        sort (a, l, m);                T(n/2)
        sort (a, m+1, r);              T(n/2)
        merge (a, l, m, r);            O(n)
    }
}
```

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

$= \alpha T(n/2) + \beta(n)$

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

Quick sort

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

Worst case :- $T(n) = T(n-1) + T(0) + O(n)$

Similarities

1. Best case complexity → Same best case complexity. This occurs when the input is already sorted or nearly sorted.

2. Divide and conquer → Both algs use divide and conquer approach to sort array.

Differences

1. Worst case complexity
   a. merge → $O(n \log n)$
   b. quick → $O(n^2)$

2. Stability → Merge sort is stable it preserves the relative order of equal elements.

3. Space complexity → Quick sort → inplace
   Merge sort → requires additional space.

12)
```
void swap ( int &a , int &b )
{
   int temp = a ;
   a = b ;
   b = temp ;
}
void stable ( int arr [] , int n )
{
  for ( int i = 0 ; i < n-1 ; ++i )
  {
      int minIndex = i ;
      for ( int j = i+1 ; j < n ;
      {
        if ( arr [j] < arr [ minIndex ] )
        {
            minIndex = j ;
        }
      }
   int minValue = arr [minIndex] ;
```

```
        while ( minIndex > i )
        {
            arr[minIndex] = arr[minIndex - 1];
            minIndex --;
        }
        arr[i] = minVal;
    }
}
```

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

14) Merge sort :-
- efficient use of disk

- minimize disk I/O
- scalability
- predictable performance

External sorting → that can handle massive amounts of data when the data being sorted does not fit into the main memory and instead must sit in a slower external memory
Eg → k way merge sort.

Internal sorting → does not require extra memory.