```
bool linearsearch(int arr[], int size, int target){
    for(int i=0; i<size; i++){
        if(arr[i] == target){
            return true;
        }
    }
    return false;
}
```

Iterative insertion sort :-

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

Recursive Insertion sort

```
void recursiveInsertionSort(int arr[], int n){
    if(n<=1)
        return;
    recursiveInsertionSort(arr, n-1);
    int last = arr[n-1];
    int j = n-2;
```

```
while ( j>=0 && arr[j] > last) {
    arr[j+1] = arr[j];
    j--;
}
arr[j+1] = last;
```

(ii) It is an online sorting algo because it sort a list as it recieves new elements one by one

(iii) Merge Sort:- Doesn't sort list online. it first divides list into smaller subsets, sorts those recursively

(iii) Quick sort:- Doesn't sort list online, it choose a pivot elements & partitions list into sublists. So the full list must be present

Ans3

Name	Time Complexity	Best Case T.C	Average Case T.C	Worst Case T.C	Space Complexity
(i)	Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
(ii)	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
(iii)	Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
(iv)	Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
(v)	Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n\log n)$
(vi)	Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$
(vii)	Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$
(viii)	Radix Sort	$O(kn)$	$O(kn)$	$O(kn)$	$O(d+n)$

Q	Name of Sorting	Inplace	Stable	Online
1	Bubble Sort	✓	✓	x
2	Selection Sort	✓	x	x
3	Insertion Sort	✓	✓	✓
4	Merge Sort	x	✓	x
5	Quick sort	x	x	x
6	Heap Sort	✓	x	✓
7	Counting Sort	x	✓	x
8	Radix Sort	x	✓	x

Q5 Recursive binary search

```
int binarysearch(int arr[], int low, int high, int key)
    if (low < high){
        int mid = low + (high - low) / 2
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < target)
        return binarysearch(arr, mid+1, high, key);
        else
        return binarysearch(arr, low, mid - 1, key);
        }
    return -1;
    }
```

Iterative Binary search

```
int binarysearch( int arr[], int size, int key)
    {
    int low = 0, high = size - 1;
    while ( low <= high ){
        int mid = low + (high - low)/2
```

```
if (arr[mid] == target)
    return mid;
elsif ( arr[mid] < target)
    low = mid + 1;
else
    high = mid - 1;
return -1;
```

Linear search :-

Time Complexity :- O(n)

Space Complexity :- O(1)

Binary search (Recursive)

Time Complexity = O(log n)

Space Complexity = O(log n)

Binary Search (Iterative)

Time Complexity = O(log n)

Space Complexity O(1)

Q6 In each step of binary search, the algorithm divides the search in half, efficiently reducing the size of problem to half. Therefore the recurrence relation

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

with base case

$$T(1) = O(1)$$

Q⑦

```
void findpairwithsum(int a[], int size, int k, int &index1,
                                                int &index2)
{
    sort(A, A+size);
    int left = 0, right = size-1;

    while(left < right) {
        int sum = A[left] + A[right];
        if (sum == k)
        {
            index1 = left+1;
            index2 = right;
            return;
        }
        else if (sum < k){
            left++;
        }
        else {
            right--;
        }
    }
    index1 = -1;
    index2 = -1;
}
```

Time Complexity = $O(n \log n) + O(n)$
simplifies to $O(n \log n)$

Ans 8 Quicksort is a popular choice due to its average time complexity $O(n \log n)$ & good performance on a wide range of inputs. It's efficient, in place & often outperforms other algos in sorting making it suitable for most scenarios

Ans 9 Counting the number of inversions in an array is a measure of how far the array is from being sorted

Now let's count number of inversions
Divide the array into two halves
Recursively count inversions in each half
During merge step, count no of inversions

Array :- {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

Step 1: Divide array into two halves:
Left : {7, 21, 31, 8, 10} | Right : {1, 20, 6, 4, 5}

Step 2: Recursively Count
In left half:
Left : {7, 21, 31} | Right {8, 10}
Inversions : (8, 7), (10, 8) -> 2 inversions

In right half:
Left : {1, 20} | Right : {6, 4, 5}
Inversions : (6, 1), (6, 4), (6, 5) -> 3 inversion

Step 3: Merge Sorted halves & count

Total inversions :- 2 (left half) + 3 (right half) + 16 (merge step) = 21 inversions

Ans 10 Best Case Scenario
-> Occurs when the chosen pivot divides the array into two nearly equal halves in each partition.
Time Complexity O(n log n)
Worst - Case Complexity :- Occurs when the chosen pivot is either the smallest or largest element in array in each partition. step

- Result in partitioning step dividing the array into one subarray with n-1 element & another with 0.
- Time complexity is $O(n^2)$

Ans 11

Merge sort

Best Case $T(n) = 2T(n/2) + O(n)$

Worst Case $T(n) = 2T(n/2) + O(n)$

Quicksort

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

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

Similarities :

① Both have same recurrence relation in best case scenario : $T(n) = 2T(n/2) + O(n)$. Because both divide into two halfes

Difference

1> In worst case scenario merge sort has $O(n \log n)$ & Quicksort has $O(n^2)$

2> Merge Sort is stable but quicksort Not.

3> Space Complexity of merge Sort is $O(n)$ and for quick sort it is $O(\log n)$

Q12

```
Void stableSelectionSort ( int arr[], int n ) {
    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] = minvalue;
}
}
```

Ans 13 When dealing with a dataset that exceeds the available physical memory, we need to resort to external sorting algorithms

Algorithm Choice :-

One of the most suitable external sorting algorithms for this scenario is Merge Sort

Reason

(i) Efficient use of Memory
(ii) Minimization of I/O
(iii) Stable Sorting

1 Internal Sorting :-

Algorithms that operate entirely within the computer main memory. These algorithms assume that the entire set fits into memory at once allowing for fast random access

2 External Sorting

Necessary when dataset is too large to fit entirely into memory & must be store on external devices. These algorithms divide the dataset into smaller chunks that can fit into memory, sort these chunks ~~literally~~ internally