Ans1 pseudo code for insertion sort.

```
#include <iostream>
using namespace std;

int linear_search(int arr[], int key, int size) {
    for(int i=0; i<size; i++){
        if(arr[i]== key){
            return i;
        }
        else if(arr[i] > key){
            return -1; //element not found
        }
    }
    return -1;
}
```

Ans2 (1) Insertion sort iterative

```
#include<iostream>
using namespace std;
void insertion_iter(int arr[], int size){
    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;
    }
}
```

(11) Insertion Sort recursive

```
void insertion_recursive(int arr[], int n){
    if (n<=1)
        return;
    insertion_recursive(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;
}
```

Insertion sort is sometimes called an "online sorting algorithm" because it can sort list of elements as they are being received one at a time, without having to wait for the entire list to be received or processed first.

Ans 3 Complexity of all sorting algorithms.

	Best case	Average case	Worst case	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort		$O(n \log n)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Ans 4

(1) Inplace : Sorts the input array by rearranging the elements within the array itself. for eg.
- Bubble Sort
- Selection Sort
- Insertion Sort
-

(2) Stable : Preserves the relative order of equal elements in the array. Elements with the same value are sorted in same order.
- Bubble Sort
- Insertion Sot
- Merge Sort

- Count Sort

(3) Online : Sorts the stream of elements as they arrive.
- Insertion Sort.

Ans5 Recursive code for binary search.

```
int binary(int arr[], int l, int r, int x){
    if (r >= l)
        int mid = l + (r - l)/2 ;
        if (arr[mid] == x)
            return mid;
        if(arr[mid] > x)
            return binary(arr, l, mid-1, x);
        return binary(arr, mid+1, r, x);
    }
    return -1;
}
```

* Iterative code for binary search

```
int binary (int arr[], int n, int x){
    int l=0, r=n-1;
    while(l <= r){
        int mid = l + (r-l)/2 ;
        if (arr[mid] == x)
            return mid;
        if(arr[mid] < x)
            l = mid+1;
        else r = mid-1;
    }
} return -1
```

	Best Case	Avg. Case	Worst	Space Comp.
Binary Search (Recursive)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary (Iterative)	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Linear Search (Recursive)	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linear (Iterative)	$O(1)$	$O(n)$	$O(n)$	$O(1)$

Ans 6 → The recurrence relation expresses the time complexity of the binary search algorithm in terms of its sub-problems. The algo divides the input array in half each iteration & solves a sub-problem of size $n/2$.

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

where

$T(n)$ → array size is n. (Time complexity)

$T(n/2)$ → array size is $(n/2)$. (")

$O(1)$ → is the time complexity for comparing middle element to target element.

Ans7 → ~~The~~ The following algorithm is suitable for the desired task.

Step 1: Sort the input array in non-decreasing order.

Step 2: Initialize two pointers i & j, to point to the first & last elements of the array respectively.

Step 3: While i < j, compute the sum A[i] + A[j].

Step 4: If sum == k, return i & j.

Step 5: If sum < k, increment i by 1.

Step 6: If sum > k, decrement j by 1.

The time complexity of the above algorithm is O(n).

Ans8

- Quicksort is widely used sorting algorithm that has an average time complexity of O(n log n) & is often faster than other popular sorting algorithms. Quicksort is particularly efficient for large datasets & can be easily implemented in place to save memory. However, its worst case time complexity is O(n²) which can occur when the input data is already sorted.

Ans9 → To count the number of inversions in the given array {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}.

Ans 10 →

* Best case → the pivot element chosen should be the median of the array. If the pivot is chosen as the median at each step, then the partioning step will divide the array into two sub-arrays of equal size, resulting in balanced tree of recursive calls. In this case, the time complexity of Quick Sort is O(nlogn).

* Worst Case

In worst case the pivot element chosen at each step is either largest or smallest element in the sub-array.

Note that: Worst occurs when the array is already sorted or reverse sorted. Time complexity will be $O(n^2)$.

Ans11 → Recurrence Relation for merge sat

Best Case → $T(n) = 2T(n/2) + O(n)$

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

Recurrence relation for quick sat

Best Case → $T(n) = 2T(n/2) + O(n)$

Worst Case → $T(n) = T(n-1) + O(n)$

Similarities between the time ~~array~~ algorithms is that they have same ~~worst~~ average & ~~best~~ case time complexity i.e. ~~$O(n^2)$~~ & $O(n \log n)$ ~~respectively~~.

The difference between the time complexities of merge & quick sort are that merge sat has worst case time complexity of $O(n \log n)$ & quick sort has the worst case time complexity of $O(n^2)$.

Both merge & quick sat have the same space complexity i.e. $O(n)$ as they need to create the temporary array.

Ans 12 → Yes, it is possible to implement the stable version of selection sort.

```
#include <iostream>
using namespace std;
void selectionSort(int arr[], int n){
    for(int 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 temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
```

In the above code, we maintain a key variable to hold the value of minimum element found in the inner loop.

Ans 13 → Yes, we can modify the bubble sort algorithm to optimize it so that it does not scan the entire array once it is already sorted.

```
#include <iostream>
using namespace std;

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

Ans 14 → It is not possible to load the entire array into the memory for sorting using internal sorting. In this case we would need to use external sorting algorithms that operate on disk instead of memory.

External sorting is the technique used to sort large data sets that can not be held in memory at once. It involves a combination of internal & external sorting techniques.

The most commonly used extenal sorting algorithm is extenal merge sort. In this algorithm the data is split into smaller parts that can fit into memory & each chunk is sorted using an intnal sorting, such as quick sort or heapsort. The sorted chunks are then merged together in a series of passes, where the data is read from the disk, merged & written back to disk.