1. Bool search (int arr[], int target) {

int R = arr.size();

for (int i=0; i < R; i++) {

if (arr[i] == k) {

return true;

}

else if (arr[i] > k) {

break;

}

}

return false;

}

2.

Iterative

→ for

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

Recursive

→ void insertion(int arr[]; int n){

```
    if(n<=1) {
        return;
    }
    insertion(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 is also called online sorting algorithm as it doesn't wait for the user to enter all the data it starts sorting element as soon as the first element is entered unlike other sorting technique which work on full data

3. Complexity of sorts

	Best	average	worst
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
quick	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
count	$O(n+k)$	$O(n+k)$	$O(n+k)$

4.

In-place sorting	online sorting	stable sorting
→ Bubble sort	→ Insertion sort	→ Insertion sort
→ Selection sort		→ Bubble sort
→ insertion sort		→ merge sort
→ Heap sort		

5. iterative

```
int BinarySearch(int arr[], int k){
    int l = 0;
    int h = arr.size() - 1;
    while (l <= h){
        int mid = l + (h-l)/2;
        if (arr[mid] == k){
            return mid;
        }
        else if (arr[mid] < k){
            l = mid+1;
        }
        else {
            h = mid-1;
        }
    }
    return -1;
}
```

T.C = O(log n)

S.C = O(1)

recursive

```
bool  Binary( int arr[], int l, int r, int key)

{
    if (l > r):
        return false;
    int mid = l + (r - l)/2;
    if(arr[mid] == key)
        return true;
    else if(arr[mid] > key)
        Binary(arr, l, mid-1, key);
    Binary(arr, mid+1, r, key);
}
```

T.C = O(logn)
S.C = O(1)

6. The recurrence relation for binary recursive search can be expressed as

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

6. According to me quicksort is best for practical use as it is considered one of the fastest sorting algorithms for average and typical cases. Its avg case time complexity is $O(n \log n)$ and worst case is $O(n^2)$ but it rarely occurs as the pivot element is chosen randomly every time and also it doesn't require any extra memory

7. In an array, an inversion occurs when two elements are out of their sorted order.
for eg:- Consider an array A, if there are two indices i and j such that $i<j$ and $A[i] > A[j]$, then the pair $(A[i], A[j])$ forms an inversion

Counting inversions using merge sort:-

→
```
int mergesort(int arr[], int temp[], int l, int r)
{
    int mid, inv=0;
    if(r>l){
        mid=(r+l)/2;

        inv = inv + mergesort(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=inv+(mid-i);
        }
    }
    while(i<=mid-1){
        temp[k++]=arr[i++];
    }
    while(j<=r){
        temp[k++]=arr[j++];
    }
    for(i=l; i<=r; i++){
        arr[i]=temp[i];
    }
    return inv;
}
```

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

8. The Best Case scenario for quick sort is when the chosen pivot element divides the array into two approximately equal halves. ($T.C = O(n \log n)$)
As for the worst case scenario is when the chosen pivot element divides the array into 2 unbalanced halves. ($T.C = O(n^2)$);

9. Merge sort → Best Case = $T(n) = 2T(n/2) + O(n)$
Worst Case = $T(n) = 2T(n/2) + O(n)$

Quick sort → Best Case = $T(n) = 2T(n/2) + O(n)$
Worst Case = $T(n) = T(n-1) + O(n)$

Similarities
→ Both the sorting algorithm works on divide and conquer strategy.
→ Both algorithms have best case and worst case time complexity as $O(n \log n)$

Diffsence:- Merge sort is more effivate on large arrays while quick sort work efficiently on smaller sized array

:- Merge sort require additional space proportional to the size of the input array, while quick sort is an in-place sorting technique which means it doesn't require any extra space making it more memory efficient

12. Stable version of selection sort:-

```
void selection( int arr[]){
    int n= arr.size(),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. Modified version of bubble sort that can resolve the problem is :-

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