# Assignment → 3

**Ans-1** Linear Search for Sorted array

* Pseudocode

1) Algo START
2) we are calling linear_search function, from main
2.1) linear_search (a[], n, loc, key)
2.2) initialize loc=-1
2.3) for i=0 to n-1 && a[i]<=key && loc==-1
2.3.1) if (a[i]==key)
2.3.2) loc=i
2.4) return loc
3) END

a= array
n= size of array
loc= using as flag
key= item to found

**Ans-2** Pseudocode for inseration sort (Iterative)

1) Inseration_Sort (A)
1.1) for j=2 to A.length
1.2) key = A[j]
1.3) //insert A[j] into the sorted sequence A[1... j-1]
1.4) i= j-1
1.5) while i>0 && A[i]>key
1.6) A[i+1] = A[i]
1.7) i= i-1
1.8) A[i+1] = key

Yashraj

→ Inseration sort is also called Online Algorithm/Sorting, because inseration sort Considered one input element per iteration and produces a partial solution without Considering future element. Inseration Sort produces the optimum result.

→ other Sorting Algorithm consider as offline sorting, like Selection Sort we sort the Array by repeatedly finding the minimum element (Considering Ascending Order) from unsorted part and putting it at beginning. which require access to entire input,

**Ans-3** Complexity of Sorting Algorithm

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

**Ans-4**

Comparision of sorting Algorithm

| | Stable | Inplace | Online |
|---|---|---|---|
| • Bubble | ✓ | ✓ | |
| • Selection | X | ✓ | |
| • Insertion | ✓ | ✓ | ✓ |
| • Merge | ✓ | X | |
| • Quick | X | X | |
| • Heap | X | ✓ | |

**Ans-5** Iterative Binary Search

1) int binarysearch(int a[], int k)

1.1) intializing $l=0$, $h=A.length-1$

1.2) while $l<=h$

1.3) mid = $(l+h)/2$ ;

1.4) X=a[mid]; return mid

1.5) x< a[mid] ; h=mid-1;

Time Complexity →
Best → $O(1)$
Average → $O(\log_2 n)$
Worst → $O(\log_2 n)$

Space Complexity → $O(1)$

a= array
k= key to found

Yashwant

1.6) $x > a[mid]$; ~~search~~ $L = mid - 1$;

Recursive Binary Search

1) int binary_search(int a[], int k, int L, int h)

1.1) int mid = l+(r-1)/2;

1.2) x = a[mid]; return mid

1.3) x < a [mid]; return binary_search(a, k, L, mid-1);

1.4) x > a [mid]; return binary_search( a, k, mid+1, h);

| Time Complexity | | Space Complexity | |
|---|---|---|---|
| Best case → | O(1) | Best → | O(1) |
| Worst case → | O(logn) | Average → | O(logn) |
| Average case → | O(logn) | Worst → | O(logn) |

Ans 6 Recurrence relation is used for determine the relation between the time Complexity of problem + time Complexity of Subproblem's Solution.

sorted, T(n)

```
bool binarySearch (int *arr, int l, int r, int key)
{ if (l>r)
      return false;
  int mid = (l+r)/2;
  if (arr[mid] == key)
      return true;
  else if (arr[mid] < key)
      return binarysearch (arr, mid+1, r, key);  → T(n/2)
  else
      return binarysearch (arr, l, mid-1, key);  → T(n/2)
}
```

$$\left[\begin{array}{c} T(n) = T(n/2)+1 \\ T(1) = 1 \end{array}\right]$$

Yashwant

Ans-8 Quick sort is the fastest general purpose sort. In most practical situation, quicksort is the method of choice. If stability is important ~~of choice~~ of space is available, merge sort might be best.

Ans-9 Inversion Count for an array indicates - how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in the reverse order, the inversion count is the maximum.

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

```
# include <bits/stdc++.h>
using namespace std;
int merge_sort (int arr[], int temp[], int left, int right);
int merge (int arr[], int temp[], int left, int mid, int right);
int mergesort (int arr[], int array, int size)
{ int temp [array_size];
    return merge_sort (arr, temp, 0, array_size -1);
}
int merge_sort (int arr[], int temp[], int left, int right)
  { int mid, inv_count =0;
     if (right > left)
     { mid = left + (right - left)/2;
        inv_count += mergesort (arr, temp, left, mid);
        inv_count += mergesort (arr, temp, mid+1, right);
        inv_count += merge (arr, temp, left, mid+1, right);
     }
       return inv_count;
  }
int merge (int arr[], int temp[], int left, int mid, int right)
 { int i, j, k, inv_count = 0;
     i = left;
     j = mid;
```

```
k = left;
while ((i<=mid-1) && (j<=right))
{ if (arr[i] <= arr[j])
    temp[k++] = arr[i++];
  else { temp[k++] = arr[j++];
    inv_count = inv_count + (mid-i);
    }
  } while (i<=mid-1)
      temp[k++] = arr[i++];
    while (j<=right)
      temp[k++] = arr[j++];
    for (i=left; i<=right; i++)
        arr[i] = temp[i];
    return inv_count; }
int main()
{ int arr[] = {7, 21, 51, 8, 10, 1, 20, 6, 4, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  int ans = mergesort(arr, n);
  cout << "no. of inversion are" << ans;
      return0; }
```

Ans10 The worst case time complexity of Quicksort is $O(n^2)$. The worst case occurs when the picked pivot is always an extreme (Smallest or largest) element. This happen when input array is sorted or reverse sorted and either first or last element is picked as pivot. The best case of quick sort is when we will select pivot as a mean element.

Ans11 Recurrence relation of:

a) Mergesort $\Rightarrow T(n) = 2T(n/2) + n$

b) Quicksort $\Rightarrow T(n) = 2T(n/2) + n$