TUTORIAL - 3

Que1) Write linear search pseudocode to search an element in a sorted array with minimum comparisons.

Sol

```
int Linear_Search (int *arr, int n, int key)
    for i >=0 to n-1
        if arr[i] = key
            return i;
    return -1
```

Que2) Write pseudo code for iterative & recursive insertion sort. Insertion Sort is called online sorting. Why? What about other sorting algorithms that has been discussed in lectures?

Sol

Iterative Insertion Sort

```
void insertion_sort (int arr[], int n)
    int i, temp, j;
    for i<-1 to n
        temp <- arr[i]
        j <- i-1
        While (j >=0 AND arr[j] >temp)
            arr[j+1] <- arr[j]
            j <- j-1
        arr[j+1] <- temp
```

Recursive Insertion Sort

```
void insertion_sort (int arr[], int n)
    if (n<=1)
        return
    insertion_sort (arr, n-1)
    last = arr[n-1]
    j = n-2
    while (j>=0 AND arr[j] > last)
        arr[j+1] = arr[j]
        j--;
    arr[j+1] = last
```

Insertion Sort is called online Sorting because it does not need to know anything about what values it will sort and the information is requested while the algorithm is running.

Que 3) Complexity of all the Sorting algorithms

Sol (i) Selection Sort

T.C = Best Case : $O(n^2)$
Worst Case : $O(n^2)$

S.C = $O(1)$

(ii) Insertion Sort

T.C = Best Case : $O(n)$
Worst Case : $O(n^2)$

S.C = $O(1)$

(iii) Merge Sort

T.C = Best Case : $O(n \log n)$

Worst Case : $O(n \log n)$

S.C = $O(n)$

(iv) Quick Sort

T.C = Best Case : $O(n \log n)$

Worst Case : $O(n^2)$

S.C = $O(n)$

(v) Heap Sort

T.C = Best Case : $O(n \log n)$

Worst Case : $O(n \log n)$

S.C = $O(1)$

(vi) Bubble Sort

TC = Best Case : $O(n^2)$

Worst Case : $O(n^2)$

S.C = $O(1)$

Que 4 Divide all the sorting algo into inplace / stable / online sorting.

Sol

Sorting	Inplace	Stable	Online
Selection Sort	✓		
Insertion Sort	✓	✓	✓
Merge Sort		✓	
Quick Sort	✓		
Heap Sort	✓		
Bubble Sort	✓		

Ques) Write recursive / iterative pseudo code for binary Search What is the Time & Space Complexity of Linear and Binary Search (Recursive & Iterative)

Sol

Iterative Binary Search

```
int Binary-Search (int arr [ ] , int l, int r, int x)
{
    While (l <= r)
    {
        int m ← (l+r)/2;
        if (arr[m] == x)
            return m;
        if (arr [m] < x)
            l ← m+1;
        else
            r ← m-1;
    }
    return -1;
}
```

Recursive Binary Search

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

Time Complexity : Best Case = $O(1)$

Average Case = $O(\log n)$

Worst Case = $O(\log n)$

Que6) Write recurrence relation for binary recursive Search

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

Que7) Find two indexes such that $A[i] + A[j] = K$ in minimum time Complexity.

Sol

Que8) Which Sorting is best for practical uses? Explain

Sol Quick Sort is the fastest general purpose Sort. In most practical situations, quick Sort is the method of choice. If stability is important and space is available, merge Sort might be best.

Que9) What do you mean by number of inversions in an array? Count the number of inversions in Array arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} using merge Sort.

Sol 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 array is Sorted in the reverse order, the inversion count is 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 merge_sort (int arr[], int array_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 = (right + left) / 2;
        inv_count += _merge_sort (arr, temp, left, mid);
        inv_count += _merge_sort (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;
    int 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, 31, 8, 10, 1, 20, 6, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int ans = merge_Sort (arr, n);
    cout << "Number of inversion are " << ans;
    return 0;
}
```

Que 10) In which cases Quick Sort will give the best and the Worst Case time Complexity?

Sol The Worst Case time Complexity of quick Sort is $O(n^2)$.

The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens 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.

Que 11) Write Recurrence Relation of Merge & Quick Sort in best & worst Case? What are the similarities & differences b/w complexities of two algo and why?

Sol

Merge Sort ⇒ $T(n) = 2T(n/2) + n$

Quick Sort ⇒ $T(n) = 2T(n/2) + n$

→ Merge Sort is more efficient & works faster than quick sort in case of larger array size or datasets.

→ worst Case Complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge sort.

Que 12) Selection Sort is not stable by default but can you write a version of stable Selection Sort.

Sol Stable Selection Sort

```
using namespace std;
void stableSelectionSort (int a[], int n)
{  for (int i=0; i<n-1; i++)
   {   int min = i;
       for (int j = i+1; j<n; j++)
           if (a[min] > a[j])
               min=j;
       int key = a[min];
       while (min > i)
       {  a[min] = a[min-1];
          min--;
       }
       a[i] =key;
   }
}
```

```
int main()
{
    int a[] = {4, 5, 3, 2, 4, 1};
    int n = sizeof(a) / sizeof(a[0]);
    StableSelectionSort(a, n);
    for (int i = 0; i<n; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

Ques13) Your computer has a RAM (Physical memory) of 2GB & you are given an array of 4 GB for sorting. Which algo you are going to use for this purpose & why? Also explain the concept of External & Internal sorting.

Sol The easiest way to do this is to use external sorting. We divide our source file into temporary files of size equal to the size of the RAM and first sort these files.

External Sorting - If the input data is such that it cannot adjusted in the memory entirely at once, it needs to be sorted in a hard disk, floppy disk or any other storage device. This is called external sorting.

Internal Sorting - If the input data is such that it can adjusted in the main memory at once, it is called internal sorting.