Aaditya Kumar
Sec: DS-1
R.No.: 01

TCS-409.

Assignment-02.

1.)

```
int expSearch(int arr[], int t)
{
    int n= arr.length();
    if (array[0]== t)
        return 0;

    int i=1;
    while(i<n && arr[i]<=t)
        i=i*2;

    return binarySearch(arr, i/2, min(i, n-1), t);
}
int binarySearch(int arr[], int l, int r, t)
{
    while(l<=r)
        int m= l+(r-l)/2;
        if (arr[m]==t)
            return m;

        if(arr[m]<t)
            l=m+1;
        else
            r=m-1;

    return -1;
}
```

2.) Iterative Insertion Sort →

```
~~int~~ void
~~iterative~~ insSort(int arr[])
{
    int n = arr.length();
    for (i=0; i<n; 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~~ int insSort(int arr[], int n)
{
    if (n<=1)
        return;
    insSort(arr, n-1);
    int key = arr[n-1];
    int j=n-2;
    while( j>=0 && arr[j] > key)
        arr[j+1]=arr[j]
        j=j-1;
    arr[j+1]=key;
}
```

It is called online sort because it can sort a list as it receives it. This means that as each new element is added to the list, it can be immediately inserted into its correct position in the sorted list.

3.) Time complexities →

- Bubble Sort:

  Worst case = $O(n^2)$

  Best case = $O(n)$

- Selection Sort:

  Worst: $O(n^2)$

  Best: $O(n^2)$

- Insertion Sort:

  Worst: $O(n^2)$

  Best: $O(n)$

- Heap Sort:

  Worst: $O(n\log n)$

  Best: $O(n\log n)$

  Avg: $O(n\log n)$.

- Radix Sort:

  Worst: $O(kn)$

  Best: $O(kn)$

  Avg: $O(kn)$

4.)

Inplace Algo: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Heap Sort

Stable Sorting Algo: Insertion, merge & Bubble sort

Online Sorting Algo: Insertion Sort.

5.)

Recursive Binary Search:

```
int  bins (int arr[], int l, int r, int t)
  {
       if (r >= l)
            int m = l + (r-l)/2;
            if (arr[m] == t)
                 return m;
            if (arr[m] > t)
                 return bins(arr, l, m-1, t);
            else
                 return bins(arr, m+1, r, t);
       return -1;
  }
```

iterative Binary Search:

```
int  bins (int arr[], int l, int r, int t)
  {
     while (l <= r)
          int m = l + (r-l)/2;
          if (arr[m] == t)
               return m;
          if (arr[m] < t)
               l = m+1;
          else
               r = m - 1;
  }  return -1;
```

Time & space complexities:

• Linear Search:

time comp = $O(n)$ worst case.

Space comp = $O(1)$ for iterative, $O(n)$ for recursive

• Binary Search:

time comp: $O(\log n)$ worst case

Space comp: $O(1)$ for iterative, $O(\log n)$ for recursive

6.) Recurrence relation for binary search:

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

where:

$T(n)$: time taken to search in a sorted array of size n.

7.)

```
#include<iostream>
#include<vector>
#include<unordered-map>
using namespace std;

pair<int, int> findind (vector<int>& nums, int t)
    {
        unordered_map <int,int> numMap;
        for (int i=0; i< nums.size(); ++i)
        {
            int c= t - nums[i];
            if (numMap.find (c) != numMap.end())
                return make-pair (numMap[c], i);
        }
        numMap [nums[i]] = i;
        }
        return make-pair(-1, -1);
    }
```

```
int main()
{
    vector<int> nums={2,7,11,15};
    int t=9;
    pair<int, int> i= findind (nums, t);
    if (i.first!=-1 && i.second!=-1)
        cout<<i.first<<i.second<<endl;
    else
        cout<<"no such pair exists."<<endl;

    return 0;
}
```

8.) ~~Merge~~ Quick sort is a fast, effiecient & commonly used sorting algorithm that's often the best choice for practical situations. It's used in Computer Science for tasks like arranging lists & arrays, searching, merging & normalization.

9.) An inversion occurs when 2 elements in an array are out of their sorted order, If there are 2 indices 'i' & 'j' in an array 'arr' such i<j but arr[i] > arr[j], then the pair (arr[i], arr[j]) is an inversion.

code→

```
#include <iostream>
# include <vector>
using namespace std;
long long merge(vector<int>& arr, int l, int m, int r)
{
    vector<int> temp (r-l+1);
    int i=l, j=m+1, k=0;
    long long inv=0;
```

```
while(i<=m && j<=r)
{
    if(arr[i]<=arr[j])
        temp[k++]=arr[i++];
    else
        temp[k++]=arr[j++];
        inv+=m-i+1;
}
while(i<=m)
    temp[k++]=arr[i++];
while(j<=r)
    temp[k++]=arr[j++];

for(int i=l, k=0; i<=r; ++i, ++k)
    arr[i]=temp[k];


return inv;
}
long long mergesort(vector<int>&arr, int l, int r)
{
    long long inv=0;
    if(l<r)
        int m= l+(r-l)/2;
        inv += mergesort(arr, l, m);
        inv += mergesort(arr, m+1, r);
        inv += merge(arr, l, m, r);

    return inv;
}
int main(){
    vector<int> arr = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5};
    long long inv= mergesort(arr, 0, arr.size()-1);
    cout<<"no.of inversions ="<< inv<<endl;
    return 0;
}
```

10.) Quicksort will give best case time complexity when pivot element is the median element, or atleast consistently reduces the problem size by a constant factor.

It will give the worst case time complexity when the pivot element is consistently the smallest or the largest element in the array, leading to an unbalanced partition & inefficient sorting

11) Recurrence reln for mergesort in best & worst case:

$T(n) = T(n/2) + On$

Recurrence reln for Quicksort:

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

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

Both algorithms are comparison sorts & have an avg-time complexity of $O(n\log n)$.

Quicksort's worst-case time complexity is $O(n^2)$ when the pivot selection leads to ~~an~~ unbalanced partitions, whereas mergesort's worst case time complexity is always $O(n\log n)$ regardless of the input.