Name ⇒ Kavya Bansal

DAA Assignment

Section ⇒ D & 1

Roll No → 63

Q1 → Write linear search pseudocode to search an element in a sorted array with minimum comparisions.

Sol ⇒

```
int linearSearch ( vector<int> arr, int key)
{
    int n = arr.size();
    for (i=0 to n-1)
    {
        if (arr[i] == key)
        {
            return i;
        }
    }
    return -1;
}
```

Q2 → Write pseudo code for iterative and recursive insertion sort. Insertion sort is called online sorting. Why? What about other sorting algorithms that has been discussed in lectures?

Sol → Iterative ⇒

```
void insertionSort ( vector<int> & arr)
{
    int n = arr.size(), i, j, temp;
    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] = temp;
    }
}
```

Recursive ⇒

```
void insertionSort(vector<int>&arr, int n)
{
    if(n<=1)
        return;

    insertionSort(arr, n-1);
    int temp = arr[n-1];
    j = n-2;
    while (j>=0 and arr[j] > temp)
    {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = temp;
}
```

Insertion Sort is called an online sorting algorithm because it can sort an array as it receives new elements without needing the entire list to be presented beforehand.

Among other sorting algorithms merge sort and bubble can also be adapted to work in online manner.

Q3→ Complexity of all the sorting algorithms that has been discussed in the lectures.

Sol→

	Time complexity			Space complexity
	Best	Worst	Average	
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	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(\log n)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$

Q4 ⇒ Divide all the sorting algorithms into inplace / stable / online sorting.

	Inplace	Stable	Online
Selection	✓	✗	✗
Insertion	✓	✓	✓
Bubble	✓	✓	✗
Quick	✓	✗	✗
Merge	✗	✓	✗
Heap	✓	✗	✗

Q5→ Write recursive / iterative pseudo code for binary search. What is the time and space complexity of linear and binary search (recursive and iterative).

Sol→ Iterative ⇒

```
int binarySearch (vector<int> arr, int key)
{
    int s = 0;
    int e = arr.size() - 1;
    int mid = s + (e - s) / 2;
    while (s <= e)
    {
        if (arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] > key)
        {
            e = mid - 1;
        }
        else
        {
            s = mid + 1;
        }
        mid = s + (e - s) / 2;
    }
    return -1;
}
```

Recursive ⇒

```
int binarySearch (vector<int> arr, int s, int e, int key)
{
    if( e >= s)
    {
        int mid = s+(e-s)/2;
        if(arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] > key)
        {
            return binarySearch (arr, s, mid-1, key);
        }
        else
        {
            return binarySearch (arr, mid+1, e, key);
        }
    }
    return -1;
}
```

Linear Search ⇒ Time Complexity ⇒ Best ⇒ O(1)
Worst ⇒ O(n)
Average ⇒ O(n)

Space complexity ⇒ O(1)

Binary Search ⇒ Time complexity ⇒ Best ⇒ O(1)
Worst ⇒ O(logn)
Average ⇒ O(logn)

Space complexity ⇒ O(1)

Q6 – Write recurrance relation for binary recursive search.

Sol ⇒ $T(n) = T(n/2) + 1$

Q7→ Write two index such that A[i] + A[j] = k in minimum time complexity.

Sol⇒

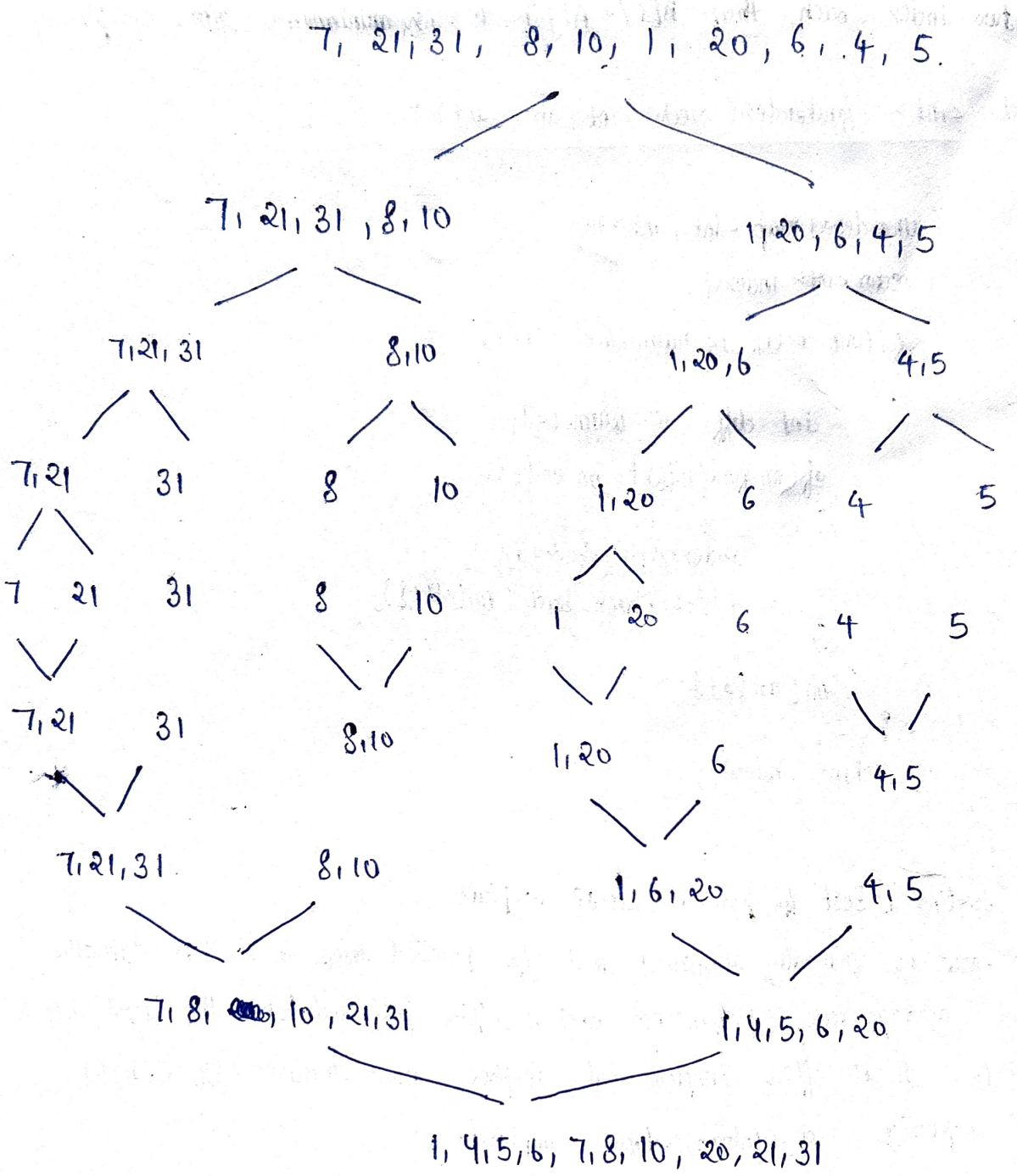
```
vector <int> findIndex( vector <int> arr, int k)
{
    unordered_map <int, int> m;
    vector <int> indexes;
    for(int i=0; i< nums.size(); i++)
    {
        int diff = k - nums[i];
        if(m.find(diff) != m.end())
        {
            indexes.push_back(i);
            indexes.push_back(m[diff]);
        }
        m[arr[i]] = i;
    }
    return indexes;
}
```

Q8→ Which sorting is best for practical uses? Explain

Sol→ Quick Sort is generally considered best for practical uses as it has typically excellent average case performance and is often preferred when the input size is large. It is often implemented in-place and requires only O(log n) space complexity and O(nlogn) time complexity.

Q9→ 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→ The number of inversions in an array is how unsorted the array is. An inversion occurs when two elements in an array are out of order relative to each other.



Total inversions = 31

Qno - In which cases Quick Sort will give the best and worst case time complexity?

Sol- The best-case time complexity occurs when the chosen pivot element divides the input array into two roughly-sized subarrays with each partition being balanced. T.C ⇒ $O(n \log n)$

The worst case time complexity of quick sort occurs when the chosen pivot element consistently divides the input array into extremely unbalanced partitions. (when input array is already sorted in ascending or descending order)
T.C ⇒ $O(n^2)$.

Q11- Write the recurrance relation of Merge and Quick sort in best and worst case? What are the similarities and differences between complexities of two algorithms and why?

Sol- Merge Sort ⇒

- Best case recurrance relation ⇒ $T(n) = 2T(n/2) + O(n)$
- Worst case recurrance relation ⇒ $T(n) = 2T(n/2) + O(n)$.

Quick Sort ⇒

- Best case recurrance relation ⇒ $T(n) = 2T(n/2) + O(n)$.
- Worst case recurrance relation ⇒ $T(n) = T(n-1) + O(n)$.

Similarities ⇒ Both the algorithms have best-case time complexity $O(n \log n)$ when the input is well behaved.
Both the algorithms use divide and conquer approach.

Differences ⇒ Merge sort is stable while quick sort is not.

Merge sort typically requires $O(n)$ additional space for merging while quick sort is implemented in place requiring only $O(\log n)$ additional space.

Q12→ Selection sort is not stable by default but can you write a version of stable selection sort.

Sol→

```
void selectionSort ( vector<int>& arr)
{
    int n = arr.size();
    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 mval = arr[min];
        while (min>i)
        {
            arr[min] = arr[min-1];
            min--;
        }
        arr[i] = mval;
    }
}
```

Q13→ Bubble sort scans the whole array even when array is sorted. Can you modify the bubble sort so that it doesn't scan the whole array once it is sorted.

Sol→

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

Q14- Your computer has a RAM of 2GB and you are given an array of 4GB for sorting. which algorithm you are going to use for this purpose and why? Also explain the concept of External and Internal Sorting.

Sol- As per the given conditions, we cannot perform the sorting entirely in memory using traditional sorting algorithms. Instead we would need an external sorting algorithm. One such algorithm that is used is External Merge Sort.

External Merge Sort is an algorithm designed to handle large datasets that cannot fit entirely in the memory. It works by dividing the dataset into smaller chunks that can fit in memory, sorting these smaller chunks internally and then merging these sorted chunks together to produce the final sorted output.

* Internal sorting algorithms are designed to sort datasets that can fit entirely in memory.

* External sorting algorithms are designed to sort datasets that are too large to fit entirely in memory.