DATE ___/___/___

Assignment - 2

Q1: using namespace std;

```
string search (int arr [], int n, int x)
  for (int i=0; i<n; i++)
     { if (arr [i] == x)
          return "Found";
       else
          return "Not found";
```

Q2. void sortReverse (int a[], int n)

```
{ if (n<=1)
     return;
  SortReverse (a, n-1);
  int last = a [n-1];
  int j = n-2;
  while (j>=0 && a[j] > last)
     {a[j+1] = a[j];
        j--;
     }
  a[j+1] = last;
}
```

DATE ___/___/___

Iterative

```
void insert (int n) {
for (i = 1; i<n ; i++)
  { int t = a[i];
    int j = i - 1;
    while (j >= 0 && a[j] >= t)
      { a[j+1] = a[j];
        j--;
      }
    a[j+1] = t;
}
```

Insertion sort considers one input element per iteration and produces a partial solution without considering future Elements. Thus insertion sort is an online algorithm.

Other sorting algorithms are →

- ✓ Inplace sorting algorithm - That don ot take any extra space
- ✓ Stable sorting algorithm - when the related order of equal elements donot

change after sorting

✓ External sorting algorithm → those algorithm that can handle maximum amount of data when the data being sorted does not fit into the main memory

✓ Internal sorting Algorithm → opposite to the External sorting algorithm here main memory is required and no External memory is there.

Q3. ✓ Bubble sort → Time Complexity → $O(N^2)$ for all cases

Space complexity → $O(1)$

✓ Selection sort → Time complexity → $O(N^2)$ for all cases

Space complexity = $O(1)$

✓ Insertion sort → Time complexity = $O(N^2)$ for all cases

Space complexity = $O(1)$

DATE ___/___/___

Count sort - Time complexity → O(N+K)
Space complexity → O(n+K)

✓ Quick sort - Time complexity → O(nlogn) - Best/Av
O(n²) - worst
Space Complexity → O(nlogn)

✓ Mergesort - Time complexity → O(nlogn) - All case
Space Complexity → O(n)

✓ Heapsort → Time complexity → O(nlogn) - All cas
Space complexity → O(1)

04. Inplace sorting Algorithm

→ Bubble sort
→ Selection sort
→ Insertion sort
→ Quick sort
→ Heap sort

Stable Sorting Algo
→ Insertion sort
→ Merge sort

DATE

Online sorting Algorithm

→ Insertion sort.

Binary (int array[], int target, low, high)

OS: if low > high:
    return Not found

mid = (low+high)/2
if (array[mid] == target):
    return mid.

elseif (array[mid] > target):
    return binary (array, target, low, mid-1)

else:
return binary (array, target, mid+1, high)

Iterative.

```
binary (int a[], int target):
    low = 0
    high = length (array) - 1;
    while low <= high:
        mid = (low + high)/2
        if a[mid] == target:
```

DATE ___/___/___

return mid;
else if (a[mid] < target)
    low = mid + 1;
else
    high = mid - 1;

return Not found.

Time complexity → O(log n)
Space Complexity ≥ O(1)
] Binary search.

Time complexity = $O(n^2)$
space complexity = O(1)
] linear search.

Q6. $T(n) = T(n/2) + 1$.

Q8. Quick sort is the best for practical use.

Date

Page

9. **Inversions**

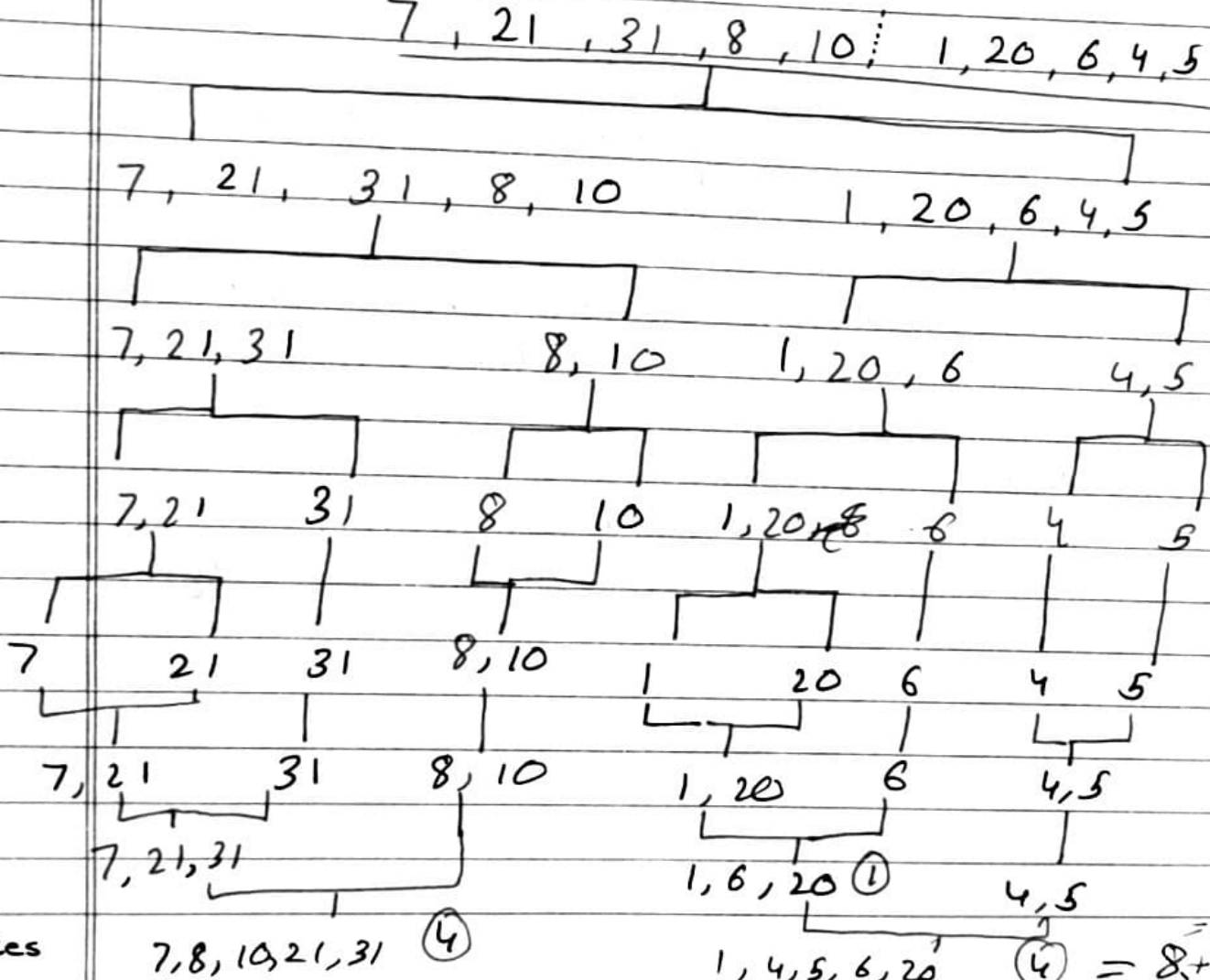
Number of inversion means how many element is shifted before placing new element at its prescribed location.

for eg →

1, 2, 7, 10, 12, 13 has to accomodate 4

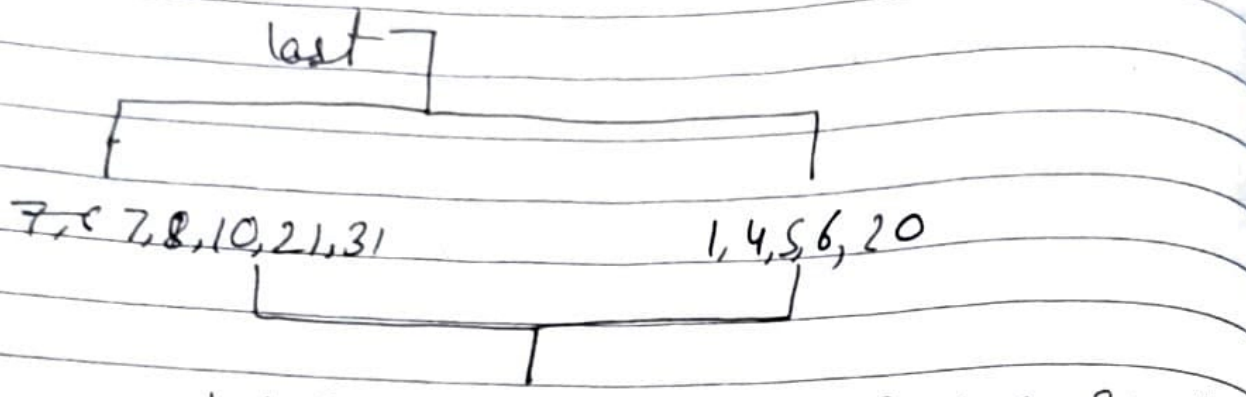
1, 2, 4, 7, 10, 12, 13 ← inversion = 4

Merge Sort



Date

Page



Total = 17 + 9 = 26

10. Quick Sort

Best → $O(n \log n)$ ↓

Condition → pivot is exactly the middle element

Worst → $O(n^2)$ ↓

Condition → pivot is either the first or last element.

Average → $O(n \log n)$ ↓

Condition → Array is roughly divided in equally half subarrays

12. **STable Quick Sort**

```
void SQS(vector<pair<int, int>& arr) {
    /* first element holds value second holds
       occurence */
    int n = arr.size() - 1;

    for (int i = 0; i < n; i++)
    {
        int min_idx = i;
        for (int j = i+1; j < n; j++)
        {
            if (arr[j].first < arr[min_idx].first)
            {
                min_idx = j;
            }
        }
        pair<int,int> key = arr[min_idx];
        while (min_idx > i) {
            arr[min_idx] = arr[min_idx-1];
            min_idx--;
        }
        arr[i] = key;
    }
}
```

Date

Page

13. Bubble sort without scanning whole array -

Beside double check loop we can use single loop in which one element is at i position and other is at i+1. if a[i] < a[i+1] then go for next comparison if not then call bubble sort for i to n indexs.

Comparison Code

```
void bs(int a[], int i, int n)
{
    if (
```

13. Bubble sort after sorting.

This can be achieved by pairing element with boolean data type. if it is sorted then it will store true else false. If true is found at 0th index break the condition and come out else iterate for bubble sort.

Code

```
void bs(vector<int, bool>& arr)
{
    int n = arr.size() - 1;
```

```
for(i=0; i<n; i++)
{
    if (arr[0].second == true)
    {
        break;
    }
    else
    {
        for(j=i+1; j<n; j++)
        {
            if (arr[j]<arr[i])
            {
                Swap(arr, i, j);
            }
        }
    }
}
arr[0].second = true; // after first call turns a[0] true.
}
```

Note ** → if array is of size n, i.e n > n & then if a element is added after sorting array will not able to sort again. to achieve capability to sort again, while inserting a element change arr[0].second to false again

Merging of data greater than available memory

We cant sort data of 4GB in 2GB memory to do so we have to use External Sorting methods like External merge sort or External quick sort.

External Sorting → It is sorting in which data is sorted and stored in slower memory for a while. Then it is retrieved from file to sort with other data. This chunking of data allowes us to sort large data with less resorces usage.

Internal Sorting → It is sorting in which less memory is directly used and no secondary memory is used. It can only accomodate chunck smaller than available memory.

Speed Comparison.

Internal > external

Storage can be sorted

External > internal