Rudraksh Agarwal
48
ML

Assignment-2

Q→1 Write linear search pseudocode to search an element in a sorted array with minimum comparisons?

Ans→

```
int linear_search(const int array[], int size, int target)
{
  if(size==0)
  {
    return -1;
  }
  if(array[0]== target)
  {
    return 0;
  }
  int i=1;
  while(i<size && array[i]<=target)
  {
    if(array[i]==target)
    {
      return i;
    }
    i++;
  }
  return -1;
}
```

Q→2 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?

Ans → Iterative Insertion sort :

```
void insertion_sort (int A[], int n)
{
   for(int i=1 ; i<n; i++)
   {
      int key = A[i];
      int j=i-1;
      while (j>=0 && A[j] >key)
      {
         A[j+1] = A[j];
         j=j-1;
      }
      A[j+1] = key;
   }
}
```

Recursive insertion sort :

```
void insertion_sort(int A[], int n){
   if (n <= 1)
   {
     return;
   }
   insertion_sort(A, n-1);
   int key = A[n-1];
   int j = n-2;
   while (j>=0 && A[j] >key) {
      A[j+1] = A[j];
      j=j-1;
   }
   A[j+1] = key;
}
```

Insertion sort is termed "online sorting because it maintains a sorted sublist within the larger list.
As new elements arrive, it inserts them into the appropriate position within the sorted sublist, keeping the list sorted in real time.

Assume an array : [9,4,2,2,3,4,1]
Let this array as [9a, 4a, 2a, 2b, 3a, 4b, 1a]
↳ Each iteration of insertion sort.

1st → [9a, 4a, 2a, 2b, 3a, 4b, 1a]
2nd → [4a, 9a, 2a, 2b, 3a, 4b, 1a]
3rd → [2a, 4a, 9a, 2b, 3a, 4b, 1a]
4th → [2a, 2b, 4a, 9a, 3a, 4b, 1a]
5th → [2a, 2b, 3a, 4a, 9a, 4b, 1a]
6th → [2a, 2b, 3a, 4a, 4b, 9a, 1a]
7th → [1a, 2a, 2b, 3a, 4a, 4b, 9a]

Since in our actual array 2a, 4a were before 2b, 4b respectively, we can say that the element present in actual array is same after other is sorted array and hence insertion sort is online sorting.

Other Online Sorting →

1. Insertion Sort ✓
2. Heap sort ✓
3. Quick Sort ✓
4. Bubble sort ✓
5. Selection Sort ✓
6. Count sort ✓
7. Radix Sort ✓
8. Merge Sort ✗

Q-③ Complexity of all the sorting algorithms that has been discussed in lectures?

Sol" →

	Sort	Time Complexity	Space Complexity
①	Bubble Sort	$O(n^2)$	O(1)
②	Heap sort	O(n log n)	O(1)
③	Selection sort	$O(n^2)$	O(1)
④	Insertion sort	$O(n^2)$	O(1)
⑤	Quick sort	$O(n^2)$	O(n)
⑥	Merge sort	O(n log n)	O(n)
⑦	Counting Sort	O(n+k)	O(n+k)
⑧	Radix sort	O(n*k)	O(n+k)

* All space & time complexity for worst cases only.

Q-④ Divide all the sorting algorithms into inplace / stable/online sorti

Sol" →

	Sort	Inplace	Stable	On-line
①	Bubble sort	1 (True/Yes)	1	0
②	Heap sort	1	0 (False/No)	0
③	Insertion Sort	1	1	1
④	Quick sort	1	0	0
⑤	Count sort	0	1	0
⑥	Merge sort	0	1	0
⑦	Radix sort	0	1	0
⑧	Selection sort	1	0	0

Q-5 Write recursive/Iterative pseudo code for binary search. What is the Time & Space complexity of linear & Binary search (Recursive & Iterative).

Ans: Binary Search Iterative :

```
int Binary_Search (int arr[], int size, int target)
{
    int l = 0;
    int h = size-1;
    while (l <= h)
    {
        int mid = l + (h - l)/2;
        if (array[mid] == target)
        {
            return mid;
        }
        else if (array[mid] < target)
        {
            l = mid+1;
        }
        else
        {
            h = mid-1;
        }
    }
    return -1;
}
```

Recursive Binary Search:

```
int Binary_search (int array[], int target, int low, int high)
{
   if (low > high)
   {
     return -1;
   }
   int mid = low + (high - low) / 2;
   if (array [mid] == target)
   {
      return mid;
   }
   else if (array [mid] < target)
   {
      return Binary_search (array, target, mid+1, high);
   }
   else
   {
     return Binary_search (array, target, low, mid-1);
   }
}
```

Linear search (Iterative)

Time Complexity = O(n)
Space Complexity = O(1)

Binary Search (Iterative)

Time Complexity = O(log(n))
Space Complexity = O(1)

Linear Search (recursive)

Time Complexity = $O(n)$
Space Complexity = $O(1)$
↳ no stack used

Binary Search (recursive)

Time Complexity = $O(\log(n))$
Space Complexity = $O(\log(n))$ ← stack to recursive calls.

Q-16. Write recurrance relation for binary recursive search.

Sol": pseudocode:

```
T(n) ⇒ Binary search (a, i, j, x)        — O(1)
       mid = (i+j)/2                     — O(1)
       if (a[mid] == x)
          return (mid)
       if (a[mid] > x)
       ** return Binary search (a, i, mid-1, x)  ← T(n/2)
       else
       ** return Binary search (a, mid+1, j, x)  ← T(n/2)
```

reason for **

Assume $x = 20$

10	20	30	40 (Mid)	50	60	70

← search in left side of mide as $x < a[mid]$

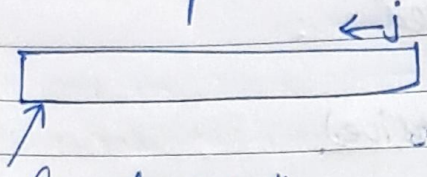
10	20	30

return 20 O(1)

if $x = 10$; array would again split in half & we can say $T(x) = T(n/2) + 1$.

Q→⑧. Find two indexes such that $A[i] + A[j] = k$ in min time complexity?

Ans. Here, we will have 2 pointers



x element array

if a[i] + a[j] == sum → return
if a[i] + a[j] < sum → increase i
if a[i] + a[j] > sum → increase j

→ This will only work if array is sorted.

if array is not sorted we have to use linear search approach.

Two pointer approach + merge sort approach

Time complexity merge sort → O(nlogn)

Time Complexity two pointer approach → O(n)

Total = O(nlogn + n)

Since nlogn > n it will be dominent

Total Time Complexity = O(nlogn) Ans.

Linear Search approach

Time Complexity linear Search first a element i is fixed which j iterates so,

Time Complexity = $O(n^2)$

On comparing we find merge sort + two pointer approach is better than linear search.

```
function_search (A[], n, k)
{
    merge_sort (A, n);   // merge sort func. call.
    i=0;
    j= n-1;
    while (i<j)
    {
      if ((A[i]+A[j])==k)
      {
        return i,j;
      }
      else if ((A[i]+A[j])<k)
      {
        i++;
      }
      else
      {
        j--;
      }
    }
    return -1, -1
}
```

② → ⑧ Which Sorting is best for practical uses? Explain.

Ans. We can't say or assume any particular algorithm as best algorithm because each algorithm has its own advantages & disadvantages. We must consider sorting as per use. For ex →

1. **Merge Sort:**
Can be used where stability is important and has ample amount of free memory so suitable for large database.

2. **Quick Sort:**
It can be used to (where) data is uniformly distributed & more emphasizes on time. Uses less memory & can be used in database.

3. **Heap Sort:**
No additional memory is required but is little slower than above two. Can be used in database sorting.

4. **Insertion Sort:** Used is smaller database or questions where some part needs to be sorted.

5. **Selection Sort:** Time Complexity being high is practically not used as much but can be implied in a smaller sets.

6. **Bubble Sort:** Rarely Used.

Q→(a) What do you mean by number of inversions in an array? Count the no. of inversions in Array arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} using merge sort?

Ans. **Inversions:**

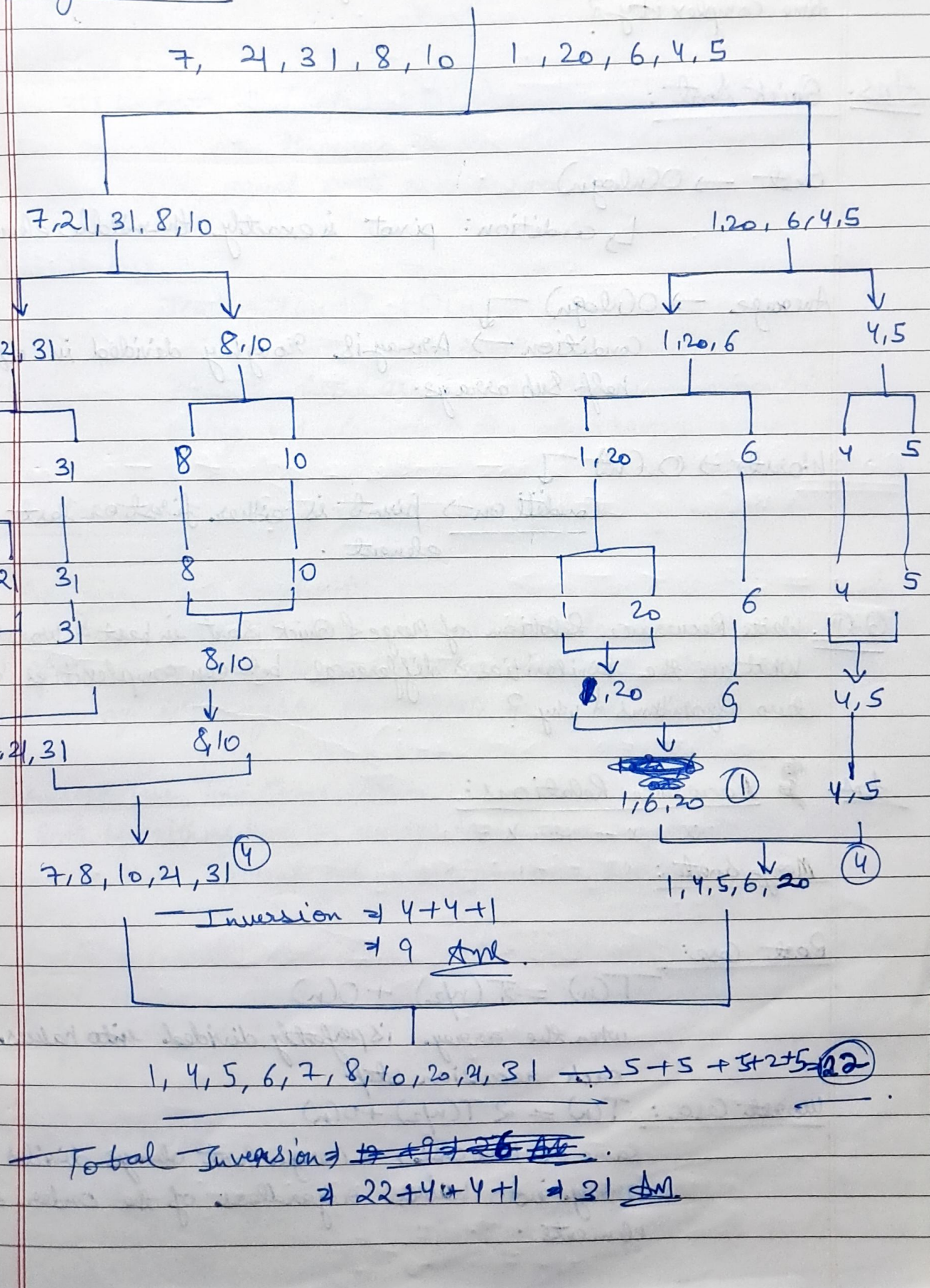
No. of inversions means how many element is shifted before placing new element its prescribed location.

For ex→

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

1, 2, 4, 7, 10, 12, 13 ← inversions = 4.

Merge Sort


7, 21, 31, 8, 10 | 1, 20, 6, 4, 5
7,21,31,8,10
1,20,6,4,5
7,21,31
8,10
1,20,6
4,5
7,21
31
8
10
1,20
6
4
5
7
21
31
8
10
1
20
6
4
5
7,21
31
8,10
8,20
6
4,5
7,21,31
8,10
1,6,20
1
4,5
7,8,10,21,31
4
1,4,5,6,20
4
Inversion → 4+4+1
→ 9
1, 4, 5, 6, 7, 8, 10, 20, 21, 31 → 5+5+5+2+5 = 22
Total Inversion → 22+4+4+1 → 31

Q-10. In which cases Quick Sort will give the best & the worst case time complexity?

Ans. Quick Sort:

Best → O(nlogn)
↳ condition: pivot is exactly the middle element.

Average → O(nlogn)
↳ Condition → Array is roughly devided in equally half sub arrays.

Worst → O(n^2)
↳ Condition → pivot is either first or last element.

Q-11. Write Recurrance Relation of Merge & Quick Sort in best & worst cases? What are the similarities & differences between complexities of two algorithms & why?

Ans. Recurrence Relations:

Merge Sort:

Best Case:
$T(n) = 2T(n/2) + O(n)$
when the array is perfectly divided into halves in each recursion step.

Worst Case: $T(n) = 2T(n/2) + O(n)$
Same as best case, as merge sort always divides the array into halves regardless of the order of elements.

Quick Sort :

Best Case :

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

when the pivot chosen divides the array into roughly equal parts in each recursion step.

Worst Case :

$T(n) = T(n-1) + O(n)$

when the pivot chosen is either the smallest or largest element in the array, resulting in one partition having $n-1$ elements & the other having none.

Similarities :

1) Best Case Time Complexity :
Both Merge and Quick Sort have a best case time complexity of $O(n \log n)$ when their respective pivot or midpoint divides the array into roughly equal parts in each recursion step.

2) Average Case Time Complexity :
Both Algorithms have an average case time complexity of $O(n \log n)$, making them efficient for most practical scenarios.

3) Divide & Conquer : Both Merge Sort & Quick Sort follow the divide and conquer paradigm, dividing the problem into smaller subproblems & solving them recursively.

Differences :

1. Worst Case Time Complexity : While merge sort has a worst case time complexity of $O(n \log n)$, Quick Sort can degrade to $O(n^2)$ in

the worst case, particularly when the pivot selection leads to highly unbalanced partitions.

2. Stability: Merge sort is stable, meaning it preserves the relative order of equal elements, whereas Quick sort is not inherently stable.

3. Space Complexity: Merge sort typically requires additional space proportional to the size of the input array for the merging process, while Quick sort is an in-place sorting algorithm & doesn't require additional space for merging.

Q→⑫. Selection sort is not stable by default but can you write a version of stable selection sort?

Ans:
```
#include <iostream>
using namespace std;
void stable_selection_sort( int arr[ ], int n)
{
  for(int i=0 ; i<n-1 ; ++i)
  {
    int min_index = i;
    for( int j=i+1 ; j<n ; ++j)
    {
      if (arr [j] < arr[min_index])
      {
        min_index=j;
      }
    }
    int selected = arr[min_index];
    for (int j= min_index ; j >i; --j)
    {
      arr[j] =arr[j-1];
    }
```

```
        arr[j] = selected;
    }
}

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

Q-13. Bubble sort scans 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?

Ans.

```
#include <iostream>
using namespace std;
void bubble_sort(int arr[], int n)
{
    bool swapped;
    for(int i=0; i<n-1; ++i)
    {
        swapped = false;
        for(int j=0; j<n-i-1; ++j)
        {
```

```
        if (arr[j] > arr[j+1])
        {
          swap(arr[j], arr[j+1]);
          swapped = true;
        }
      }
      if (!swapped)
      {
        break;
      }
    }
}
int main()
{
  int arr[] = {5, 4, 3, 2, 1};
  int n = sizeof(arr)/sizeof(arr[0]);
  bubble_sort(arr, n);
  cout << "sorted array:";
  for (int i=0; i<n; ++i)
  {
    cout << arr[i] << " ";
  }
  cout << endl;
  return 0;
}
```

Here, in this version of bubble sort:

① We introduce a boolean variable "swapped" to track whether any swap occurred in a pass.

② After each pass, if no swap occurred in the inner loop, it means that the array is already sorted. In such a case, we break out of the

order loop, as the remaining elements are already in their correct positions.

③ This optimization reduces the no. of iterations required, making the algorithm more efficient, especially for nearly sorted arrays or arrays where only a few elements are out of place.

Q-14 Your computer has a RAM of 2GB and you are given an array of 4GB of sorting. Which algorithm you are going to use for this purpose and why? Also explain the concept of External & Internal Sorting?

Ans. In this scenario, where the size of the array exceeds the available physical Memory (RAM) of the computer, we need to use an algorithm that performs external sorting. One of the most suitable algorithms for this purpose is the merge sort algorithm.

Merge Sort for External Sorting:

① Merge sort is well-suited for external sorting because it follows the divide & conquer approach & requires minimal memory during the sorting process.

② In external sorting, when the dataset is too large to fit entirely into RAM, the dataset is divided into smaller chunks that can fit into memory, & each chunk is sorted individually.

③ Once the chunks are sorted, the sorted chunks are merged together using the merge operations which is the hallmark of merge sort.

④ Merge Sorts merge operation efficiently merges sorted chunks of data from external storage (ex→ hard disk) in a single shared output.

classmate
Date
Page

Concepts of External & Internal Sorting:

① Internal Sorting:

- Internal sorting refers to sorting algorithms that can operate entirely within the computer's primary memory (RAM).
- In internal sorting, the entire dataset to be sorted can fit into the available memory.
- Algorithms like quick sort, merge sort, insertion sort, & selection sort are commonly used for internal sorting.
- These algorithms typically have better performance since accessing data from memory is much faster than accessing data from external storage.

② External Sorting:

- External sorting refers to sorting algorithms that are designed to handle datasets that are too large to fit entirely into memory and must be stored on external storage devices, such as hard drives or SSDs.
- In external sorting, the dataset is divided into smaller chunks, each of which can fit into memory.
- These chunks are individually sorted using internal sorting algorithms.
- Once sorted the sorted chunks are merged together to produce the final sorted output.

- Merge sort is one of the most commonly used algorithms for external sorting due to its efficiently merge operation, which can merge sorted chunks of data from external storage.

In summary, for sorting a large dataset that exceeds the available physical memory (RAM), we would use an external sorting algorithm like merge sort. Merge sort's divide & conquer approach & efficient merge operation make it well-suited for handling large datasets stored on external storage devices.