Tanishka
Chauhan
ML/63

ASSIGNMENT-02

Ques-1.

```
int linear-search (int arr[], int n) {
    for (i = 0 ; i < length (array) ; i++)
    {
            if (arr[i] == key)
                return i
    }
    return -1
}
```

Ques-2 -

Iterative Insertion Sort

```
for i from 1 to length(array) - 1 :
        key = arr[i]
        j = i-1
        while (j >= 0 && arr[j] > key) :
                arr[j+1] = array[j]
                j = j-1
        array[j+1] = key

return array.
```

Recursive Insertion sort

~~for~~ :
    if n<=1:
        return array

recursive_insertion(array, n-1)
key = array[n-1]
j = n-2
while(j>=0 && array[j] >key):
        array[j+1] = array[j]
        j = j-1

array[j+1] = key
return array.

Insertion sort is a an online sorting algorithm because it can sort a list as it receives new elements one by one. It maintains a sorted sublist & inserts each new element into the correct position within that sublist making it efficient where new elements are continuously added to a list.

Other sorting algo's - merge, quick & heap sort are not inherently online sorting becoz they require the entire i/p to be available before sorting can begin.

Que 3-

1. Bubble sort

Best - $O(n)$
Avg - $O(n^2)$
Worst - $O(n^2)$

2. Selection Sort

Best - $O(n^2)$

Avg - $O(n^2)$

Worst - $O(n^2)$

3. Insertion Sort

Best - $O(n)$

Avg - $O(n^2)$

Worst - $O(n^2)$

4. Merge Sort

Best - $O(n \log n)$

Avg - $O(n \log n)$

Worst - $O(n \log n)$

5. Quick Sort

Best - $O(n \log n)$

Avg - $O(n \log n)$

Worst - $O(n^2)$

6. Heap Sort

Best - $O(n \log n)$

Avg - $O(n \log n)$

Worst - $O(n \log n)$

7. Count Sort

Best - $O(n+k)$

Avg - $O(n+k)$

Worst - $O(n+k)$

8. Radix Sort

Best - $O(nk)$

Avg - $O(nk)$

Worst - $O(nk)$

9. Bucket Sort

Best - $O(n+k)$

Avg - $(n+k)$

Worst - $O(n^2)$

Ques-4- In-place Sorting Algo's

→ Bubble Sort
→ selection Sort
→ Insertion Sort
→ Quick Sort
→ Heap Sort

Stable Sorting Algo's

→ insertion Sort.
→ Merge Sort

Online Sorting Algo's

→ Insertion Sort

Ques-5 Recursive

```
Binary (int arr[], int target, int low, int high).
if low > high :
        return NOT_FOUND
mid = (low + high)/2
if array[mid] == target:
        return mid
else if ( array [mid] > target):
        return binary (array, target, low, mid - 1);
else
        return binary (array, target, mid +1, high);
```

Iterative

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

        if (array [mid] == target):
            return mid

        else if (array [mid] < target):
            low = mid + 1;
        else

            high = mid - 1;

return NOT_found
```

$T.C = O(\log n)$ } Binary Search
$S.C = O(1)$

$T.C. = O(n^2)$ } Linear Search
$S.C = O(1)$

Ques - 6 -

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Ques-8 -

Quick Sort is often the prefared choice for sorting large datasets or arrays due to its average-case time complexity of $O(n\log n)$. Quick sort is especially useful for large datasets.

Ques-9 - Inversion occurs when there are 2 elements arr[i] & arr[j] such that $i<j$ but arr[i] > arr[j].

Inversion count provides insights into how "out of order" an array is.

Ques10 - Quick sorts performance depends on selection of pivot element. In best-case, the pivot selection leads to balanced partitions. In this T.C is $O(n\log n)$

Ques-11 - Merge Sort

- Best Case

Recurrence Relation :- $T(n) = 2T(n/2) + O(n)$

- Worst Case

Recurrence Relation :- $T(n) = 2T(n/2) + O(n)$

Quick Sort

- Best Case

Recurrence Relation :- $T(n) = T(k) + T(n-k-1) + O(n)$

- Worst Case

Recurrence Relation :- $T(n) = T(n-1) + O(n)$

Similarities

→ Both merge & quick sort have a T.C of $O(n \log n)$ in their best case

→ Both use divide & Conquer approach.

Differences

→ Quick Sort has worst T.C. of $O(n^2)$ while merge sort guarantees $O(n \log n)$ regardless of i/p data

→ Mergesort requires additional space for merging arrays whereas Quick sort is usually in-place

Ques12 - In this version of stable selection sort, when finding the minimum element in the unsorted portion of array, instead of directly swapping the minimum element with the element at the current position, we shift all elements to the right of minimum element one position to right. This maintains the relative order of equal elements, ensuring stability. Finally we place the minimum element at its correct position in the array.

Ques13 – Yes, we can modify Bubble sort by adding a flag that indicates whether any swaps were made during a pass through the array.

```
int n = len(arr);
flag = 0;
for(i=0; i<N; i++)
{   for(j=i+1; j<N-1-i; j++)
      {      if (arr[i] > arr[j])
               {   swap(arr[i], arr[j],
                   flag =1 ;
               }
          if( flag==0)
               break;
      }
}
```

Ques14 - When dealing with sorting tasks that exceeds the available memory (RAM), external sorting algorithms like Mergesort are essential for efficiently handling large datasets by utilizing disk for intermediate storage. It minimizes the number of disk accesses by sorting smaller chunks of data that can fit into memory (internal sorting) & then merging these sorted chunks.