TUTORIAL-3

Q.1) Write linear search pseudocode to search an element in a sorted array with minimum comparisions.

A.1) No. of comparisions can be reduced by copying the element to be searched (suppose x) to last location so that one last comparision when x is not present in arr[] - saved

pseudocode :-

```
search( arr, n, x)
if arr[n-1] == x  // 1 comparision
 return "true"
backup = arr[n-1].
arr[n-1] = x;

for i=0 ; i++  // No termination condition
 if arr[i] == x;   // execute at most n times
                   i.e at most n comparisions
arr[n-1] = backup
return(i<n-1) // 1 comparision
 return found;
else
return Not found;
```

Q.2) Write Pseudocode for Iterative & Recursive Insertion sort why Isort is also called Online sort.

Iterative Pseudocode:

```
// Sort an arr[] of size n
insertionsort(arr, n)
Loop from i = 1 to n-1
a→ pick element arr[i] & insert
it into sorted sequence arr[0 . . . i-1]
```

Recursive Pseudocode

```
Base case - if(n <= 1)
            return;        // If arr 1 or smaller, return.
sort first (n-1) elements
   i.e (arr, n-1).
last = arr[n-1];           // Insert last elem at correct
   j = n-2;                   position
Apply Isort algo.
print utility(f) of arr of size n.
```

- Insertion sort is called online sort because it does not need to know about what values it will sort and information is requested while algo runs.
- Where as selection is ~~off~~ insesort.

Q.3) Complexity of all sorting Algorithms :-

Selection sort - $\Omega(n^2)$
Bubble sort - $\Omega(n)$

Insertion	$\Omega(n)$
Heap	$\Omega(n \log(n))$
Quick	$\Omega(n \log(n))$
Merge	$\Omega(n \log(n))$
Count	$O(n+k)$
Radix	$O(dR)$

Q.4) Divide all sorting algorithm into Inplace / stable / online.

A.4) Inplace :- Bubble sort, selection sort, Insertion & Heap sort

Stable :- Merge sort, Count sort, Insertion sort & Bubble sort

Unstable :- Quick sort, Heap & selection sort

Q.5) Write recursive / Iterative pseudo code for binary search. What is Time & space complexity of linear & Binary search.

A.5)

```
low = 0, high = n-1;
while(low <= high)        // loop till search exhausts.
{
int mid = (low + high)/2;   // find mid value in search space & compare it with target.
if(target == num[mid])
return mid; }              // If target found
else
if(target <= nums[mid])
{       high = mid - 1;
}
```

```
else {                          // If target > middle element discard
    low = mid + 1;                 all element in left space
}
    return -1    // If no element exists in array
```

Recursive Pseudocode

```
Base condition:-
↳   if(low > high){
        return -1;
    }
    mid = (low + high)/2;   // find mid value
    if (target = num[mid])
{   return mid;             // If base condition met.
    }
    else if (target < num[mid])
{   return binarysearch(num, low, mid-1, target)
    }
    else
        return binarysearch(num, mid+1, high, target)
```

	Linear Search	Binary Search
Time(C)	$O(n)$	$O(\log_2 n)$
Space(C)	$O(1)$	$O(1)$ Iterative $O(\log_2 n)$ Recursively

Q.6) Recurrence relation for binary recursive Search.

As mean element is selected as pivot array divides in branches of equal size so that height of tree → minimum

In worst case time complexity it will be $O(n^2)$ and will happen when array - sorted and smallest / largest element is selected as pivot.

A6) Recurrence relation of binary search

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

A7) Quick sort is best sorting algo in practical use as it follows locality of reference & also it's best case complexity is $O(n \log n)$.

Q.8.) which sorting is best for practical uses, explain.

A.8) Quicksort is fastest general-purpose sort. It's more effective for datasets that fit in memory. It's an in-place sort (doesn't require extra storage) so it's appropriate for arrays

Q.9) what do you mean by number of inversions in an array?

count

A.9.) Inversion for an array indicates how far or close is the array from being sorted. If array already sorted inversion count is zero(0). but if array is sorted in reverse order inversion count = maximum

given array:-

arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}.

No. of inversions:- {(7,1), (7,6), (7,4), (7,5) (21,20)
(31,8), (31,10) (31,1) (31,20)
(31,6) (31,4) (31,5) (8,1)
(8,6) (8,5) (8,4) (10,1) (10,6) (10,4)
(10,5)}

= 20.

Q.10.) In which cases quicksort will give best & worst case time complexity.

A.10.) Best case time complexity of quicksort is O(NlogN) & that will be when pivot is selected as mean element

Q.11) Write recurrence relation of merge & quick sort in best & worst case. what are similarities & differences between the two?

A.11) Recurrence relation of merge sort in best case:-

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

on solving $T(n) = \Theta(n \log(n))$ Ans

" " quick sort

$$T(n) = T(n-1) + \Theta(n)$$

worst case: $\Theta(n^2)$ Ans

Merge sort	Quick sort
- array parted into 2 halves	- Array is parted into any ratio
- worst + average case has same complexity $\Theta(n \log n)$.	- worst case $T() = \Theta(n^2)$ requires lots of comparisions
- can work well with any type dataset irrespective of size	- Does not work well with large dataset
- Not in place, requires additional memory space to store auxillary arrays	- Quicksort is in place as it doesn't require any additional storage
- Stable	- Unstable
- linked lists	- Arrays

Q.12) selection sort can be made stable if instead of swapping, minimum element is placed on it's position without swapping.

eg:-

```
void SSort(int a[], int n)
for(int i=0; i<n-1; i++)   // Iteration

int min=i;          // find min element from arr[i] → arr[n-1]
for(int j=i+1; j<n; j++)
 if(a[min]>a[j])
    min=j;
 int key=a[min]; // Move min element at current i.
 while(min>i)
  {
  a[min]=a[min-1];
   min--;
  }
  a[i]=key;
 }
}
```

Q.13) Bubble sort scans whole array even when array is sorted. can you modify bubble sort so that it doesn't scan whole array once sorted.

A.13) Bubble sort can be optimized if inner loop doesn't cause any swap.

```
void bubbleSort(int arr[], int n)
```

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

Q.14) Your computer has RAM = 2GB & Array = 4GB - sorting which algorithm will be used & why? Explain external + internal sorting.

In such case merge sort will be efficient as it is an external sorting algorithm i.e data is divided into chunks & sorted using merge sort.

Q.14)

Internal Sorting

- All data is stored in memory at all times, while sorting - progress.
- shell sort applicable i.e access whatever array elements you wish to e whenever.

External Sorting

- Data stored outside memory & loaded to memory in small chunks.
- shell sort not applicable.
∵ are not entirely in memory
∴ random access not possible