Set-3

(20)

Q.1) Concept of Algorithm Efficiency & how it is measured (3 marks)

Algo Efficiency refers to how well an algorithm uses time & memory resources to solve a problem.

It helps determine which algo performs better for large inputs.

⇒ Measured using -

(1) Time Efficiency (2) Space Efficiency

⇒ Measurement method -

Efficiency is measured using Big O notation (O(n)), which shows how the algo's performance scales with input size n.

Q.2)	Time Complexity	Space Complexity
	Amount of time taken by an algorithm to run	Amount of memory used during execution
	Number of basic operations Linear Search → O(n)	Number of variables or storage locations used. using an extra array → O(n)
	Minimize execution time	Minimize memory usage.

Q.3) Index formula for Accessing Elements in a 3-D Array (Column Major Array

Let the array be A[L1...U1, L2...U2, L3...U3]

L1, L2, L3 = Lower bounds of dimensions

U1, L2, L3 = Upper

$N_1 = U_1 - L_1 + 1$, $N_2 = U_2 - L_2 + 1$, $N_3 = U_3 - L_3 + 1$
Base address = Base (A)

$LOC(A[i][j][k]) = Base(A) + (i - L_1) + (j - L_2) \times N_1 + (k - L_3) \times N_1 \times N_2$

Ex → A[0...2][0...2][0...2] Let B(A) = 100

LOC(A[1][1][1] = 100 + (1-0) + (1-0) x (2-0+1) + (1-0) x (2-0+1) x (2-0+1)
= 100 + 1 + 1(3) + (1) x (3) x (3)
= 100 + 1 + 3 + 9
= 113

Q. 4) Binary Search Algorithm
works on sorted array & repeatedly divides the search interval in half to find the target element efficiently

ALGO -

(1) Start with low index = 0 & upper index high = n-1
(2) Find mid index
mid = (low + high)/2
(3) Compare the mid element arr[mid] with target.
→ if arr[mid] == key, element found, return mid
→ if arr[mid] > key, search left, high = mid-1
→ if arr[mid] < key, search right, low = mid+1
(4) Repeats steps 2-3 until (low > high)
(5) If not found, return -1.

Time complexity :-
Best / Avg / Worst — O(log n)
Condition - Array must be sorted before apply Binary Search.

(Q.5) Bubble Sort

simple comparison-based sorting algorithm that repeatedly steps through the lists, compares adjacent elements & swap them if they are in the wrong order.

After each pass, the largest element "bubbles up" to its correct position.

Algorithm Steps –

(1) Start from first element & compare with the next one.

(2) If the current element isn't in the correct position, swap them.

(3) Continue this for entire array — one pass.

(4) Repeat until no swaps are needed.

Example Sort the array [5, 3, 4, 1, 2]

Pass	Compare & swap
1	[3, 5, 4, 1, 2] ↔ [3, 4, 1, 2, 5]
2	[3, 1, 2, 4, 5]
3	[1, 2, 3, 4, 5]
4	[1, 2, 3, 4, 5] Sorted.

Time Complexity —

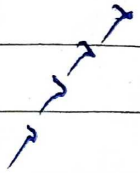
Best → $O(n)$

Avg/Worst → $O(n^2)$

Space Complexity — $O(1)$

Type → In-place, Stable sorting algorithm

```
void bubbleSort (int arr[]){
    int n = arr.length;
    for (int i=0 ; i<n-1; i++){
        for (int j=0; j<n-i-1; j++){
            if (arr[j] > arr[j+1]){
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
```



DRY RUN

0	1	2	3
2	3	1	0

i = 0 → 3
j = 0 → 3-i

i = 0 → 3 → j = 0 → 3-i = 0

arr[0] > arr[1]
temp = 2,
arr[0] = arr[1] = 3
arr[1] = 3

pass 1

2<3	Not swap	
3>1	swap	
3>0	swap	[2,1,0,3]

pass 2

2<3	Yes swap	[1,0,2,3]
2>0	swap	
2>3 ?	No swap	

pass 3

1>0	swap	
1<2	No swap	[0,1,2,3] Sorted
2<3	No swap	

(Q.6) Recursive method (factorial of a number)

Algo steps —

(1) Base case: if n=0 or 1 return 1

(2) Recursive case: Multiply n by factorial of (n-1)

```
int factorial(int n){
    if (n==0 || n==1){
        return 1;
    }
    return n * factorial(n-1);
}
```

Call factorial (5)

5 * factorial(4)

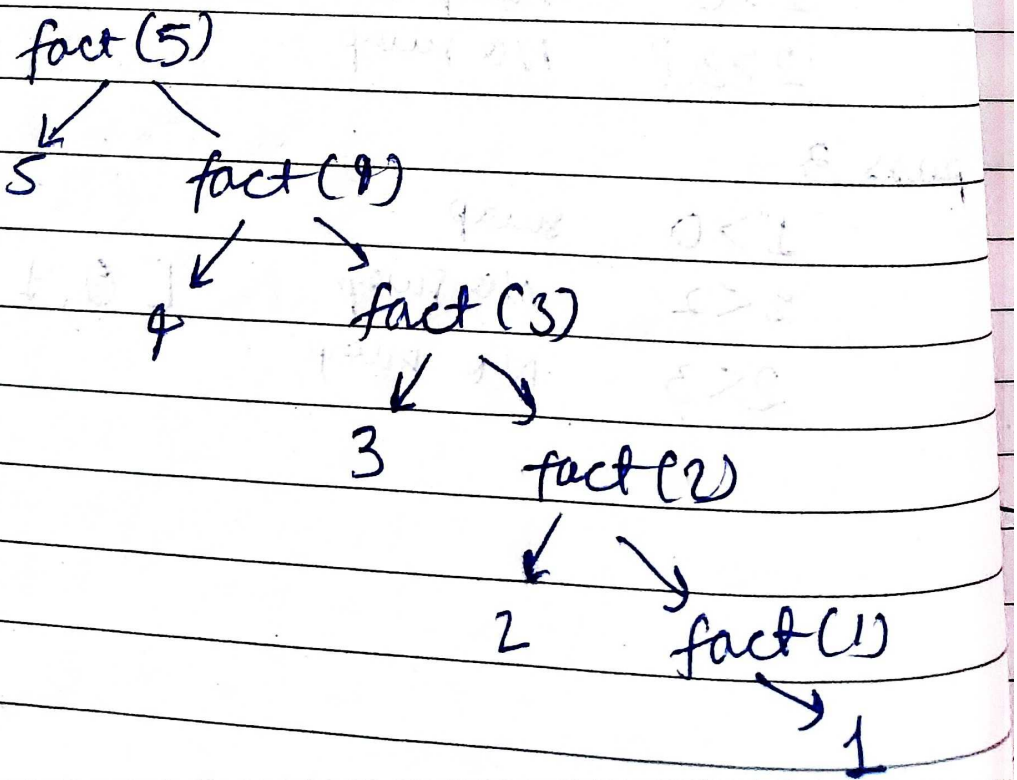
5 * 4 * factorial (3)

5 * 4 * 3 * factorial (2)

5 * 4 * 3 * 2 * factorial (1) ← Base case

5 * 4 * 3 * 2 * 1

T.C & S.C both = O(n)



2.7) Circular Linked List (CLL)

A variation of a linked list where the last node points back to the first node.

→ Can be singly circular (last node → first node)

→ Or doubly circular (last node → first node and first node → last node)

⇒ Advantage over regular linked list:-

→ Can traverse the list starting from any node.

→ Useful for buffered systems, round-robin scheduling and repeated tasks.

Structure (Singly CLL Node)

```
Class Node {
    int data;
    Node next;
    Node(int data){
        this.data = data;
        this.next = null;
    }
}
```

⇒ **ALGORITHM to traverse a CLL**

Objective- Print all nodes in the CLL.

Steps -

(1) Start with the head node

(2) Use a temp pointer and move through the list → Print temp.data
→ Move temp = temp.next.

(3) Stop when temp becomes equal to head again.

Pseudo code

```
void traverse(Node head){
    if (head == null) return;
    Node temp = head;
    do { SOP(temp.data + " ");
        temp = temp.next;
    } while (temp != head);
}
```

(Q.8) (a) Trade-offs between Recursion & iteration (Memory usage)

Recursion	Iteration
☆ Each recursive call adds a new stack frame in memory → O(n) space for n calls	☆ Iterative loops use constant memory → O(1) space (no stack overhead)
☆ Often simpler & more intuitive for problems	☆ Complex to implement.
☆ May cause stack overflow for large inputs	☆ Iteration is safer for large inputs

(b) Recursive Algorithm for Fibonacci Series

0, 1, 1, 2, 3, 5, 8, ...

⇨ Algo steps

Base case ⇒ if n=0, return 0

⇒ if n=1, return 1

Recursive case ⇒ Return fib(n-1) + fib(n-2)

```
int fib(int n) {
if (n==0) return 0;
if (n==1) return 1;
return fib(n-1) + fib(n-2);
}
```

Date
Page No.

2.9) Merge sort on an Unsorted Array
divide-and-conquer sorting algorithm
split the array into two halves
Recursively sort both halves
Merge these two sorted halves into a single sorted array.

Time Complexity → O(n log n) (even in worst case)
Space Complexity → O(n)
Type → Stable & suitable for large datasets & linked list

Real-world Applications

① External Sorting
Sorting data too large to fit in memory

② DBMS
Sorting records for merge-joins & query optimization.

③ Large-Scale Analytics
Used in big data frameworks like Hadoop to sort massive datasets.

④ Linked Lists Sorting
Merge sort works well because it does not require random access.

OR

OR Singly Linked List for Polynomial Expression

A polynomial is an expression —

$P(x) = 5x^3 + 4x^2 + 2x + 7$

coefficient (5, 4, 2, 7)

Exponent (3, 2, 1, 0)

⇨ SLL are ideal for storing polynomials because —

* Number of terms may vary dynamically
* Terms can be inserted or deleted easily

⇨ Node structure for polynomial terms

```
class PolyNode{
    int coeff, exp;
    PolyNode next;
PolyNode (int C, int e){
        coeff = C; exp = e; next = null;
```

⇨ Implementation

Op1 Create a Polynomial

each node stores one term.

Terms are linked in descending order of exp

Head → [5,3] → [4,2] → [2,1] → [7,0] → N

Op2 Traverse Polynomial

```
void display (PolyNode head){
    PolyNode temp = head;
while (temp != null){
    SOP(temp.coeff + "x^" + temp.exp);
if (temp.next != null){
        SOP(" + ");
    temp = temp.next
}
```

Step -3 Add Two Polynomials
→ Traverse both lists simultaneously
→ Compare exponents
* if equal → sum coefficients
* if one exponent is larger → insert that term

⇨ Advantage of Using SLL
Dynamic Size
Efficient insertion/Deletion
Memory Efficient.

⇨ Real-World Application
① Symbolic Mathematics → Storing and manipulating algebraic expressions.
② Computer Algebra Systems (CAS) → Mathematica, Maple, MATLAB.
③ Graphics & Engineering Computations → polynomials for curves, trajectories and simulations.