1. Explain the concept of algorithm efficiency and how it is measured.
- algorithm efficiency refers to how effectively an algorithm uses time and memory resources while solving a problem.
- it determines how the algorithm's performance scales as the size of the input increases.

Efficiency is measured using:
- time Complexity: Measures the total time an algorithm takes as a function of input size (n).
- Space Complexity: Measures the total memory used by the algorithm during execution.
- Efficiency is expressed using asymptotic notations such as Big O, Omega, and Theta.
- Example: Linear search has O(n) time complexity, while binary search has O(log n).
- Efficient algorithms save computational time, reduce resource consumption, and improve overall system performance.

2. Differentiate between time complexity and space complexity with

examples.?
ans) time Complexity:
- Refers to the total time taken by an algorithm to run completely.
- It is usually measured in terms of the number of basic operations performed.
- Example: Linear search takes O(n) time, while merge sort takes O(n log n).

Space Complexity:
- refers to the total amount of memory required by an algorithm, including input data, auxiliary space, and temporary variables.
- Example: Merge sort requires O(n) extra space for merging, while quick sort uses O(log n) space for recursive calls.

Difference: Time complexity focuses on speed, whereas space complexity focuses on memory usage. Both are important for evaluating the efficiency of algorithms, and a balance must often be achieved between them.

3. Derive the index formula for accessing elements in a 3-D array in column-major

order.?

ans) - Let A[L1..U1][L2..U2][L3..U3] be a three-dimensional array with base address BA and element size w bytes.

- in column-major order, elements of the first column are stored first, followed by the next column, and so on.

- the address of element A[i][j][k] is given by:

LOC(A[i][j][k]) = BA + [(k - L3) × (U2 - L2 + 1) × (U1 - L1 + 1) + (j - L2) × (U1 - L1 + 1) + (i - L1)] × w

Explanation:

- the offset is computed based on the number of full columns and planes that come before the desired element.

- Column-major storage is common in languages like Fortran, while C uses row-major order.

4. Write the algorithm for binary search and explain its steps.

- Concept: Binary search works on sorted arrays and repeatedly divides the array in half to locate a key.

Algorithm:

- Set low = 0 and high = n -

1.
- While low ≤ high:
a. mid = (low + high) // 2
b. If key == A[mid], return mid (found).
c. If key < A[mid], set high = mid - 1.
d. Else, set low = mid + 1.
- if the element is not found, return -1.
- Example: For A = [2, 4, 6, 8, 10, 12] and key = 10,
- Step 1: mid = 3 → A[mid] = 8 → key > 8 → low = 4.
- Step 2: mid = 4 → A[mid] = 10 → element found.

Complexity: O(log n) for average and worst cases.
Efficient for large sorted data.

5. Describe bubble sort and explain its working with an example.
ans) Concept: Bubble sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

Algorithm:
- For i = 0 to n - 1
- For j = 0 to n - i - 2
- if A[j] > A[j+1], swap them.
- Repeat until no more swaps are

needed.
- Example: Array [5, 3, 4, 1]
Pass 1: [3, 4, 1, 5]
Pass 2: [3, 1, 4, 5]
Pass 3: [1, 3, 4, 5]
Complexity: Best case O(n) (already sorted), worst $O(n^2)$.
- Properties: Simple and stable sorting algorithm, but inefficient for large datasets.

6. Write a recursive method for calculating factorial of a number and explain it.
ans) Algorithm:

```
fact(n):
if n == 0 or n == 1:
return 1
else:
return n * fact(n - 1)
```

- Example: For n = 4, fact(4) = 4 × 3 × 2 × 1 = 24.
- Explanation:
- Each recursive call multiplies the current number by the factorial of (n-1).
- Base condition (n==0 or n==1) stops the recursion.
- Complexity: Time O(n), space O(n) (due to

recursive stack).
- Applications: Mathematical computations, combinatorics, and recursion demonstration.
- 7. Explain circular linked lists and write the algorithm to traverse it.
- A circular linked list is a variation of a linked list where the last node points back to the first node instead of NULL, forming a circle.
- it can be singly or doubly circular.

Advantages:
- Traversal can start from any node.
- Useful for applications like round-robin scheduling and buffer management.

Algorithm to Traverse:

```
traverse(head):
if head == NULL:
return
temp = head
do:
print(temp->data)
temp = temp->next
while(temp =!
```

head)

Example: For nodes 10 → 20 → 30 → 10, traversal prints 10 20 30 repeatedly in a loop until stopped manually.

8. (a) Discuss the trade-offs between recursion and iteration with reference to memory usage.
(b) Write a recursive algorithm for the Fibonacci series.
ans)
(a) Trade-offs:
- Recursion uses function call stacks, which increases memory usage.
- Each call stores return address, parameters, and local variables.
- iteration uses loops and a fixed number of variables, making it more memory efficient.
- However, recursion makes code simpler and easier to understand for problems like tree traversal and divide-and-conquer.
- iteration is faster and avoids stack overflow but may make logic harder to write for naturally recursive

problems.

(b) Recursive Fibonacci Algorithm:

```
fib(n):
if n == 0:
return 0
else if n == 1:
return 1
else:
return fib(n-1) + fib(n-2
```

For n = 5 → sequence: 0, 1, 1, 2, 3, 5.

- Complexity: $O(2^n)$ due to repeated sub-problem calls; can be optimized using dynamic programming.

9. Case Study - Explain how merge sort sorts an unsorted array efficiently and discuss real-world applications where it is preferred.

- Concept: Merge sort is a divide-and-conquer sorting algorithm that divides the array into halves, sorts each half recursively, and merges them.

Algorithm Steps:

- Divide the array into two halves until each part has one element.
- Conquer by sorting the two halves

recursively.
- Combine both halves by merging them into a sorted array.
- Example: For [38, 27, 43, 3, 9, 82, 10]
- Divide into [38, 27, 43, 3] and [9, 82, 10].
- Recursively sort and merge → [3, 9, 10, 27, 38, 43, 82].
- Complexity: Best, average, and worst case = O(n log n).
- Space Complexity: O(n).

Applications:
- Sorting linked lists (since merge sort works efficiently without random access).
- External sorting (sorting data stored on disk).
- Large datasets that cannot fit entirely in memory.

Conclusion: Merge sort is stable, predictable, and ideal for cases where consistent performance and stability are required.

OR Case Study - Describe the implementation and application of singly linked lists in managing polynomial expressions.
ans) Concept: A polynomial such as $3x^3 + 5x^2 + 2x + 1$ can be efficiently represented using a singly linked

list.
- Structure: Each node contains three fields: coefficient, exponent, and a pointer to the next term.

implementation Steps:
- Create nodes for each term with coefficient and exponent.
- Link them in descending order of powers.
- For polynomial addition, traverse both lists term by term, add coefficients with equal exponents, and link results in a new list.
- For deletion, simply unlink and free the required node.

Advantages:
- Dynamic memory usage, no fixed size needed.
- Easy insertion and deletion of terms.
- Efficient polynomial addition, subtraction, and multiplication.
- Example:

P1: $3x^3 + 5x^2 + 4$

P2: $6x^3 + 2x^2 + 7$

Result (P1 + P2): $9x^3 + 7x^2 + 11$

Conclusion: Singly linked lists provide a flexible and