1. Define algorithm and explain the importance of efficiency in algorithm design.
ans) an algorithm is a finite and ordered sequence of unambiguous steps designed to solve a particular problem or perform a specific task.
- it must have properties such as finiteness, input, output, definiteness, and effectiveness.
- Efficiency in algorithm design refers to how well the algorithm uses computational resources like time and memory.
- Efficient algorithms reduce processing time and memory usage, enabling faster execution and better scalability.
- Example: Binary search is more efficient than linear search for sorted data due to lower time complexity.

Hence, designing efficient algorithms is crucial for handling large datasets and improving system performance.

2. Explain row-major and column-major representation of arrays with suitable

diagrams.
ans) Arrays can be stored in memory either in row-major or column-major order.
- In row-major order, all elements of the first row are stored consecutively, followed by the next row. This is used in C/C++ languages.
- in column-major order, all elements of the first column are stored first, followed by the next column. This is used in languages like Fortran.
- Example: For matrix A with 2 rows and 3 columns, Row-major stores as A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2], while Column-major stores as A[0][0], A[1][0], A[0][1], A[1][1], A[0][2], A[1][2].
- Understanding these representations helps calculate addresses of elements and manage multidimensional data efficiently.

3. Derive the general index formula for multi-dimensional arrays.
ans) Let A[L1..U1][L2..U2]...[Ln..Un] be an n-dimensional array stored in row-major order with base address BA and each element of size w bytes.
- the general formula for locating an element A[i1][i2]...[in] is given by:
LOC(A[i1][i2]...[in]) = BA +

$[((i_1-L_1)\times(U_2-L_2+1)\times(U_3-L_3+1)\ldots\times(U_n-L_n+1)) +$
$((i_2-L_2)\times(U_3-L_3+1)\ldots\times(U_n-L_n+1)) + \ldots + (i_n-L_n)] \times w$

- this formula computes the offset of an element from the base address.
- it is used by compilers to access elements of multi-dimensional arrays directly in O(1) time.
- row-major and column-major differ only in the order of multiplication factors in the formula.

4. Compare insertion sort and selection sort with respect to their time complexity.
ans) insertion sort:

- Works by inserting elements from the unsorted part into their correct position in the sorted part.
- Best case O(n) (already sorted), average and worst case $O(n^2)$.
- Stable and adaptive.
- Selection sort:
- Works by repeatedly finding the smallest element from the unsorted part and placing it at the beginning.
- time complexity $O(n^2)$ in all cases (best, average, worst).
- Not stable but performs fewer

swaps.

Comparison: Insertion sort performs better on small or nearly sorted data, while selection sort performs a fixed number of comparisons regardless of data order.

5. Explain merge sort algorithm with an example.
ans)
Merge sort follows the divide-and-conquer approach to sorting.

Algorithm:
- Divide the array into two halves.
- Sort each half recursively using merge sort.
- Merge the two sorted halves to produce a single sorted array.
- Example: For array [5, 2, 4, 6, 1, 3], it is divided into [5, 2, 4] and [6, 1, 3]. After sorting both halves and merging, the result is [1, 2, 3, 4, 5, 6].
- Complexity: Best, average, and worst case = O(n log n).
- Space Complexity: O(n) due to temporary arrays.
- Merge sort is stable and suitable for large datasets and external

sorting.

6. Discuss removal of recursion and write an iterative algorithm equivalent to the recursive factorial function.

ans) recursion can be replaced by iteration to reduce memory usage and function call overhead.

- Recursive factorial function:

```
fact(n){
if(n<=1)
return 1;
else
return n*fact(n-1);
}
```

iterative Version:

```
fact(n){
result = 1;
for(i=1; i<=n; i++)
result = result*i;
return result;
}
```

the iterative version uses loops and avoids extra stack memory used in recursive

calls.
- Removing recursion improves efficiency but sometimes makes code less intuitive.
- Recursion is preferred for problems that are naturally hierarchical, like tree traversal.

7. Explain insertion operation in singly linked list with an example.
ans) a singly linked list consists of nodes where each node contains data and a pointer to the next node.
- insertion Steps:
- Create a new node with given data.
- If inserting at the beginning, point the new node's next to head and update head to the new node.
- if inserting in the middle or end, traverse to the required position.
- Adjust pointers such that the new node points to the next node and the previous node points to the new node.
- Example: Insert 25 between 20 and 30 in the list 10→20→30.
- New node (25) created.
- Set 25->next = 30 and 20->next = 25.
- Resulting list: 10→20→25→30.
- Time complexity is O(1) for beginning and O(n) for

general insertion.

9. Case Study - With an example, explain sparse matrix representation using a 2-D array and discuss its advantages and limitations.
ans) A sparse matrix has most of its elements as zero. Storing all elements wastes memory, so only non-zero values are stored efficiently.

2-D Array Representation:
- Non-zero elements are stored in a 2-D array with three columns: row, column, and value.
- The first row contains (total rows, total columns, number of non-zero elements).
- Example:
Matrix
[0 4 0]
[0 0 5]
[7 0 0]
Triplet form:
(3 3 3)
(0 1 4)
(1 2 5)
(2 0

7)

Advantages: Saves memory, easy to perform transpose and addition operations.
- Limitations: Accessing individual elements is slower compared to full matrix, and the structure is complex for frequent updates.

OR Case Study - Discuss the use of doubly linked lists in complex data management scenarios, with examples of traversal and deletion operations.

Doubly linked lists are used where frequent traversal, insertion, and deletion are required in both directions.
- Each node has three fields: previous pointer, data, and next pointer.

Applications:
- Undo/redo features in text editors.
- Browser forward/backward navigation.
- Managing playlists and memory allocation.

Traversal:
- Start from head and move using next pointers until

NLL.
- Can also traverse backward using previous pointers.

Deletion:
- Adjust next pointer of the previous node and previous pointer of the next node to bypass the node to be deleted.
- Free the node's memory.

Advantages:
= Efficient bidirectional traversal and easier node removal.
- No need to traverse from the beginning to access previous elements.
- Disadvantages:
- Requires extra memory for the previous pointer.
- More complex pointer manipulation compared to singly linked lists.