1. Define time-space trade-off with an example.
ans) The time-space trade-off principle states that a program can often be made to run faster by using more memory, or use less memory by allowing it to run slower.
- in algorithm design, we sometimes store pre-computed results or additional data to save computation time.

Example: Using a lookup table for storing results of a factorial or sine function avoids recomputation and saves time but consumes extra memory.

Conclusion: The goal is to find a balance where both time and memory usage are optimized according to system limitations.

2. Discuss the applications of multidimensional arrays with suitable examples.
ans) Multidimensional arrays are used to represent data in more than one dimension, such as 2-D or 3-D

structures.

Applications include:
- Matrices and mathematical computations: 2-D arrays store rows and columns of a matrix.
- graphics and image processing: Images are stored as 2-D arrays of pixels.
- game boards: Chess or tic-tac-toe grids are represented as 2-D arrays.
- Scientific simulations: 3-D arrays represent space coordinates or physical fields.
- Example: int matrix[3][3] represents a 3×3 matrix for mathematical operations.
- Hence, multidimensional arrays simplify data representation in multiple dimensions.

3. Derive the index formula for a 1-D array and explain its significance?
ans) Let an array A[L..U] contain n elements where each element occupies w bytes and the base address is B.
- the address of element A[i] is given by:
LOC(A[i]) = B + (i - L) × w
- Derivation: The difference (i L) represents the number of elements from the first element,

multiplied by the size of each element w to get the offset.
- Example: If base = 1000, L = 0, w = 2, then LOC(a[3]) = 1000 + (3-0)×2 = 1006.
- Significance: The formula allows direct access to any element in O(1) time without sequential traversal.

4. Write and explain the linear search algorithm. What are its advantages and limitations?
ans) Algorithm:
- Start from the first element of the array.
- Compare each element with the key to be searched.
- if a match is found, return the position of the element.
- if the end of the array is reached without finding the key, return "not found."

Example: For array [3, 7, 1, 9] and key 7, comparison succeeds at position 2.
- Advantages: Simple to implement and works on unsorted data.
- Limitations: Inefficient for large arrays since every element must be checked.
- time Complexity: Best O(1), worst

O(n).

5. Explain quick-sort algorithm and give an example of partitioning.
ans) Concept: Quick-sort uses the divide-and-conquer method. It selects a pivot element, partitions the array into two parts (smaller and larger than pivot), and sorts each part recursively.

Algorithm:
- Choose a pivot element.
- Rearrange elements so that all elements smaller than the pivot are to its left and greater ones to its right.
- Recursively apply quick-sort on the left and right subarrays.
- Example: For array [9, 3, 7, 6, 2, 8], pivot = 6.
After partition: [3, 2, 6, 9, 7, 8].
Left part [3, 2] and right part [9, 7, 8] are sorted recursively.
- Complexity: Best/average O(n log n), worst $O(n^2)$.
- Advantages: Efficient and in-place with small auxiliary space.

6. Discuss the different types of recursion with

examples.
ans)1. Direct recursion: A function calls itself directly. Example: fact(n){ if(n<=1) return 1; else return n*fact(n-1); }.
2. Indirect recursion: Function A calls B and B calls A. Example: A(){B();} B(){A();}.
3. Tail recursion: Recursive call is the last statement, can be converted into iteration. Example: sum(n){ if(n==0)return 0; else return n+sum(n-1); }.
4. Head recursion: Recursive call is made first, then computation is done on return.
5. Tree recursion: A function makes more than one recursive call, common in divide-and-conquer algorithms. Example: Fibonacci series.
6. Nested recursion: The argument of the recursive call itself contains a recursive call, e.g., func(n) = func(func(n-1)).

Recursion simplifies logic but increases memory use due to function stack calls.

7. Explain insertion and deletion operations in doubly linked lists with diagrams.
ans) a doubly linked list consists of nodes where each node contains data, a pointer to the next node,

and a pointer to the previous node.

Insertion:
- Create a new node.
- Adjust pointers of the previous and next nodes to link the new node.
- Update the previous and next links accordingly.
Example: To insert x after node p, set x->prev = p, x->next = p->next, p->next->prev = x, and p->next = x.

Deletion:
- Adjust the next pointer of the previous node and the previous pointer of the next node to bypass the node being deleted.
- Free the memory of the deleted node.
- advantages traversal possible in both directions, and insertion or deletion is easier than in singly linked lists.

8. (a) Write an iterative solution for calculating Fibonacci numbers.
(b) Explain the removal of recursion with an example.
ans)
(a) Iterative Fibonacci

Algorithm:
fib(n):
a = 0, b = 1
if n == 0 return a
for i = 2 to n:
  c = a + b
  a = b
  b = c
return b
For n = 5, sequence = 0, 1, 1, 2, 3, 5.

(b) Removal of Recursion:
Every recursive function can be rewritten as an iterative version using loops and stacks.
Example: Recursive factorial fact(n){ if(n<=1) return 1; else return n*fact(n-1); } can be converted to iteration:

fact(n):
result = 1
for i = 1 to n:
result = result * i
return result
Removing recursion saves memory and avoids function call overhead but may make code less

intuitive.

9. Case Study - Given a sparse matrix, demonstrate how it can be stored using linked-list representation and explain the benefits of this approach?
ans) Concept: A sparse matrix has mostly zero values. Instead of storing all elements, only non-zero elements are stored using nodes linked together.

Linked-List Representation:
- Each node stores three fields: row index, column index, value, and a pointer to the next non-zero element.
- The linked list can be row-wise or column-wise.
- Example:
Matrix
[0 5 0]
[7 0 0]
[0 0 9]
- can be stored as three nodes:
(row 0, col 1, val 5) → (row 1, col 0, val 7) → (row 2, col 2, val 9).

Benefits:
- Saves memory by storing only non-zero

elements.
- Dynamic representation—easy to insert or delete elements.
- Useful for applications like graph adjacency lists and large scientific data where matrices are sparse.
- time Complexity: Traversal O(number of non-zero elements), which is better than scanning all m×n entries.

OR Case Study - Discuss the role of quick sort in real-time applications and compare it with merge sort in terms of efficiency.
ans)
Quick sort is widely used where fast in-memory sorting is required, such as in compiler design, database query optimization, and system libraries like C's qsort().

Advantages:
- Very efficient for small to medium data sets.
- Works in place without extra memory.
- Average complexity O(n log n).
- Comparison with Merge Sort:
- Merge sort guarantees O(n log n) in all cases but -