(Q.1) Algorithm → step-by-step procedure or a set of well-defined instructions designed to perform a specific task or solve a particular problem.

⇒ Importance of efficiency in algorithm design
It ensures that the algorithm runs fa
saving time
Uses fewer resources like memory & proces
power.
It can handle large inputs effectively, mak
them practical for real-world applications

(Q.2) Row Major and Column major representat
of arrays
In memory, 2D array is stored as 1D sequence of elements. The way elements are arranged depends on whether the system uses row or column major order.

☆ Row-Major representation
elements of each row are stored contiguou
in memory. (used in C, C++, Java)

Ex $\begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix}$ RMO ⇒ [2, 8, 3, 4]

2 → 8 → 3 → 4

☆ Column-Major representation
elements of each columns are stored contiguously in memory (used in MATLAB and Fortran]

$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 1 → 4 → 2 → 5 → 3 → 6

Date
Page No.

(Q.3) General index formula for Multi-Dimensional Arrays

Let's consider an n-dimensional array:

A [L1 ... U1] [L2 ... U2] ... [Ln ... Un]

Base address = Address of the first element A[L1][L2]...[Ln]

Element size (w) = Number of bytes per element

Li = Lower bound of the ith dimension

Vi = Upper bound of ith dimension

General Formula

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

(Q.4)

Insertion Sort	Selection Sort
★ Builds the sorted array by inserting elements into the correct position.	★ Repeatedly selects the smallest (or largest) element and places it in order.
★ Best case Time Complexity O(n)	★ Best case Time Complexity O(n²)
★ Avg case ⇒ O(n²)	★ Avg case ⇒ O(n²)
★ Worst case ⇒ O(n²)	★ Worst case ⇒ O(n²)
★ Stable	★ Not Stable
★ in-place	★ in-place
★ Insertion Sort is faster for nearly sorted data	★ Selection sort performs the same number of comparisons regardless of data order.

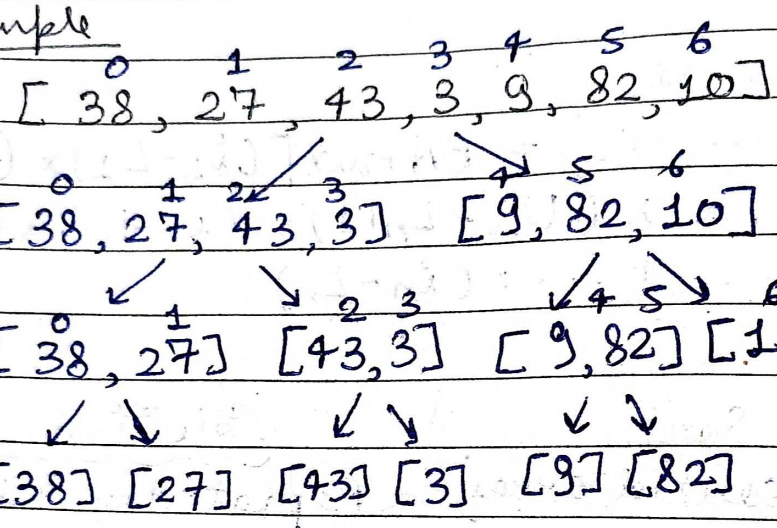
(Q.5) Merge Sort Algorithm

Divide – split the array into two halves until each subarray has only one element

Conquer – Recursively sort the two halves

Combine – Merge the two sorted halves into one sorted array.

For Example



Merge (sort each pair)

[27, 38] [3, 43] [9, 82] [10]

Merge again

[3, 27, 38, 43] [9, 10, 82]

Final Merge

[3, 9, 10, 27, 38, 43, 82]

⇨ Time Complexity

Best, Average, Worst case => $O(n \log n)$

Space complexity => $O(n)$

Merge sort is efficient & stable, ideal for large datasets but uses extra memory for merging.

Date
Page No.

6) Removal of Recursion

We convert the recursive process into an iterative one using loops, which saves memory & improves efficiency.

Recursive Factorial function

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

Equivalent iterative Algorithm

```
int num = 5;
int factorial = 1
for(int i=1 ; i<= num ; i++) {
    factorial *= i;
}
```

Advantages of Removing Recursion

Saves stack memory

Reduces function call overhead

Improves speed for large inputs.

7) Types of insertions in SLL

(A) At the beginning (head)

(a) Create a new node.

(b) Set newNode→next = head

(c) Update head = newNode

agram

10 → 20 → 30

insert 5 at the head = 5 → 10 → 20 → 30

newNode → [5|*] → [10|*] → [20|*] → [30|NULL]

(B) Insertion at the End

(a) Create a new Node & set newNode→next = NULL.

(b) Traverse the list until last node

(c) Set last→next = newNode.

Diagram

10 → 20 → 30

insert 5 at the end

[10|*] → [20|*] → [30|*] → [5|NULL]

(C) Insertion at a Specific Position

(a) Create a new Node

(b) Traverse to the node after which insertion is to be done

(c) Set newNode→next = temp→next.

Set temp→next = newNode.

Diagram

Insert 5 at the 2nd position

10 → 20 → 30

10 → 5 → 20 → 30

[10|*] → [5|*] → [20|*] → [30|*]

Advantages of LL insertion -

★ Efficient insertion (no shifting like arrays)

★ Dynamic memory allocation

Disadvantages -

★ Requires traversal to find insertion point (O(n) time).

★ Extra memory for pointers.

Date
Page No.

Q8) (a) Recursive solution to Tower of Hanoi

```
TOH (n, A, B, C)
  if n == 1;
  print "Move disk 1 from", A, "to", C
  return
TOH (n-1, A, C, B)  // Move n-1 disks from A to B
print "Move disk", n, "from", A, "to", C
TOH (n-1, B, A, C) // Move n-1 disks from B to C
```

Where A = source ; B = auxiliary ; C = destination

Logic =>

1. Move n-1 disks from A → B (C as aux)
   Move n^{th} (largest disk from A → C
   Move n-1 disks from B → C (A as aux)

Time Complexity ⇒ $T(n) = 2T(n-1)+1$

$\Rightarrow T(n) = 2^n - 1$

(b)

Advantages recursion	Disadvantages recursion
(i) Simplifies complex problems like TOH, Fibonacci Tree Traversal etc	(i) Uses more memory due to function call stack
(ii) Code is shorter & easier to understand	(ii) May cause stack overflow for large input values
(iii) useful for divide & conquer algorithms	(iii) Slower execution due to repeated function calls
(iv) Natural fits for hierarchical data structures like trees & graphs.	(iv) Can be harder to debug and trace.

(Q.9) Sparse Matrix Representation Using a 2D array

A sparse matrix contains more zero elements than non-zero elements.

Normal (2D Array) Representation

```
A[4][4] = {
            {0, 0, 3, 0},
            {0, 0, 5, 7},
            {0, 0, 0, 0},
            {0, 2, 6, 0}
          };
```

⇒ To store efficiently, we use a 3-column representation (also called Triplet form).

Row	Column	Value
0	2	3
1	2	5
1	3	7
3	1	2
3	2	6

⇒ We can store this data in a 2D array

```
Sparse[6][3] = {
                 {4, 4, 5},
                 {0, 2, 3},
                 {1, 2, 5},
                 {1, 3, 7},
                 {3, 1, 2},
                 {3, 2, 6}
               };
```

→ First row stores matrix dimensions & number of non-zero values

→ 4 rows, 4 columns, 5 non-zero elem

→ Remaining rows store (rows, columns, value) of each non-zero element.

Advantages of Sparse Matrix Representation:
(1) Memory Efficiency - saves space by storing only non-0 elements
(2) Faster computation - operations like add, or multiplication skip zero elements, making computation faster.
(3) Easier for data transmission
(4) Useful for Real-World applications
Ex ML, graph, algo & scientific computing

Limitations
(1) Complex implementation → requires additional logic to handle insertion, deletion, or search of elements.
(2) Extra Overhead → Must store row & column indices along with each value.
(3) Not suitable for Dense matrices → If many elements are non-zero, triplet form may use more space than a normal array.
(4) Difficult Random Access → direct access to an element is slower compared to normal 2D arrays.

OR

(P.T.O)

Doubly Linked Lists in Complex Data Management Applications

(1) Browser History Management :
Forward & backward navigation using next and prev pointers

(2) Undo / Redo Functionality in Editors :
Each action is stored as a node; backward traversal undoes actions, forward traversal redos them.

(3) Music / Video Playlist Management :
Allows skipping forward / backward in a playlist efficiently.

(4) Database Transaction Logs :
Easier insertion/deletion of records with previous/next links.

1. Traversal in DLL

⇨ Forward Traversal (10 → 20 → 30 → NULL)

```
Node temp = head;
while (temp != null) {
    SOP (temp.data + "→ ");
    temp = temp.next;
}
SOP ("NULL");
```

⇨ Backward Traversal (30 → 20 → 10 → NULL)

```
Node temp = tail;
while (temp != null) {
    SOP (temp.data + "→ ");
    temp = temp.prev;
}
SOP ("NULL");
```

Date
Page No.

2. Deletion in DLL

(i) Identify the node to delete (delNode)

(ii) Update the next pointer of delNode.prev to delNode.next.

(iii) Update the prev pointer of delNode.next to delNode.prev.

(iv) Delete delNode.

Code →

```
if (delNode.prev != null)
    delNode.prev.next = delNode.next;
if (delNode.next != null)
    delNode.next.prev = delNode.prev;
delNode = null;
```