Date / /
Page No.

SET-(2)

Q1. Define time-space tradeoff with example :-

time space tradeoff means using more memory (i.e. space) to make an algorithm run in less time or using less memory but taking more time.

eg :- In mergesort we take extra space as we make a temp array but it works faster.

Q2. Application of multidimensional array :-

→ used to store data in table or matrix form.
→ used in mathematical operation like addition & multiplication
→ 2D arrays store pixel values of image.
→ 2D arrays represent grids or chess boards.

Q3. 1-D array index formula :-

In memory, array elements are stored continuously.

$$LOC(A[i]) = Base(A) + (i * w)$$

where Base(A) = Base address, i = index, w = size in bytes

Example → LOC[A(4)] = 1000 + (4 × 4)
Base(A) = 1000 = 1016
w = 4 bytes.

Significance :
- Enables random access to elements in O(1) time
- Helps compilers & system locate elements quickly

Q4. Algorithm (Linear Search)

1) Start from 1st element of the array.
2) Compare the target element with each element.
3) If match found, return index of that element.
4) If not found, even after checking all elements, return -1.

Pseudocode :-

```
linearSearch(int arr, target)
for(i to length of arr-1) {
    if (arr[i] = target) {
        return i;
    }
}
return -1;
```

Advantages	disadvantages.
- easy to understand & implement.	- slow for large dataset TC = O(n)
- works on both sorted and unsorted.	- inefficient when compared to Binary Search on sorted array.
- No extra memory required.	

Q5. Quick Sort Algorithm →
It's a divide and conquer algo that works & sorts an array by partitioning it into subarrays arrays around a pivot.

Date / /
Page No.

Steps :-

1) choose pivot element from the array
2) Partition the array so that,
   - all elements smaller than pivot go to it's left.
   - all elements greater than pivot go to it's right.
3) Recursively apply the same process to the left & right.
4) The process continues until each subarray has one element of partitioning.

Pseudocode :

```
int partition (int[] arr, int low, int high) {
    int pivot = arr[high]
    i = low - 1
    for (j = low ⟶ j < high) {
        if (arr[j] < pivot) {
            i++;
            swap (arr[i], arr[j])
        }
    }
    swap (arr[i+1], arr[high])
    return (i+1);
}
```

Q6. Discuss different types of Recursion with examples :-

(1) Direct Recursion - A function calls itself directly.

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

Date / /
Page No.

(2) Indirect Recursion – A function calls itself after calling another function first.

eg>
```
void funcA (int n) {
    if (n>0) funcB(n-1);
}
void funcB (int n) {
    if (n>0) funcA(n/2);
}
```

(3) Tail Recursion – the recursion call is the last statement in the function.

eg>
```
int Sum (int n, int acc) {
    if (n==0) return acc;
    return Sum(n-1, acc+n); // last call.
}
```

(4) Non-Tail Recursion – Recursion call is not the last statement.

eg> Factorial function in (1).

Q7. Double linked list (DLL)

each node contains
- data : value stored
- next : pointer to the next node.
- prev : pointer to the previous node.

→ Allows traversal in both directions.

(a) **Insertion in DLL :-**

1) Create a new node.
2) Adjust the prev & next pointers of the node & its neighbouring nodes.

Before ins. [10] ↔ [20] ↔ [30]

After ins. [10] ↔ [20] ↔ [25] ↔ [30]

} Insert 25 after 20.

3) If inserting at
   Beginning - update head pointer, set next to current head, set prev to NULL.
   End - update last node's next pointer, set prev to last node, set next to ~~last node~~. NULL
   Middle - Update pointers of previous & next nodes accordingly.

(b) **Deletion in DLL :-**

1) Find the node to delete.
2) Adjust the next of the previous node & prev of the next node to bypass the node.
3) prev / delete the node.

Before delete [10] ↔ [20] ↔ [25] ↔ [30]

[10] ↔ [20] ↔ [30]

Date / /
Page No.

Q8.(a). Iterative solution for calculating Fiboracci numbers 0, 1, 1, 2, 3, 5, 8, 13

```
int fib (int n) {
    if (n==0) {
        return 0;
    } if (n==1){
        return 1;
    int fib0=0, fib1=1 ; fibx=0 ;
    for (int i=2; i<=n; i++) {
        fibx = fib0 + fib1;
        fib0 = fib1;
        fib1 = fibx;
    }
    return fibx;
}
```

(b). Removal of Recursion

Recursion can be replaced by loops to save stack memory.

Recursive calls are converted to iterative constructs like 'for' or 'while'.

Example : $O(2^n)$

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

Iterative approach like above in part(a) avoids recursion overhead. TC = O(n)
SC reduces from O(n) stack space to O(1).

Date / /
Page No.

Q9. Sparse Matrix Storage using LinkedList.

1. Sparse matrix

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}_{4\times5}$$

2. LinkedList Representation
each node contains
row, col, data, next
↳ pointer to the next non-zero element.

Code :-

```
class Node {
    int row, col, data;
    Node next;
    Node (int r, int c, int d) {
        row = r ; col = c ; data = d ;
        next = NULL;
    }
}
```

LL representation of the example :

Node (row = 0 , col = 2 , data = 3) →
Node (row = 2 , col = 1 , data = 7) →
Node (row = 3, col = 7 , data = 1) → NULL.

[0, 2, 3] → [2, 1, 7] → [3, 4, 1] → NULL.

Advantages of LLR →

→ Space efficient → Reduces wasted storage
→ Dynamic Size → Easy traversal of non-zero elements.

Anshika Thakur (sec-34)
24SCSE1180664

Date / /
Page No.

Set (3).

Q1. Explain the concept of algorithm efficiency and how it is measured.

A1. Algo efficiency refers to how well an algo uses time and memory resources to solve a problem. It helps determine which algo performs better for large inputs.

measured using :-
1) Time 2) Space efficiency
Efficiency is measured by Big O notation [O(n)] which shows how the algo scales with input n.

Q2.

Time Complexity	Space Complexity
Amount of time taken by an algorithm.	Amount of memory taken or used during execution.
Number of basic operations	No. of variables / storage locations
Linear search → O(n)	Using an extra array → O(n)
Minimize execution time	Minimize memory usage

Date
Page No.

Q3. Derive index formula for accessing elements in a 3D-array in column-major order.

A3. Let the array be $A[L_1 \ldots U_1, L_2 \ldots U_2, L_3 \ldots U_3]$

L_1, L_2, L_3 = lower bounds of dimensions
U_1, U_2, U_3 = Upper bounds of dimensions

$N_1 = U_1 - L_1 + 1$ $\quad N_2 = U_2 - L_2 + 1$ $\quad N_3 = U_3 - L_3 + 1$

Base Address = Base(A)

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

Example ⇒ $A[0 \ldots 2][0 \ldots 2][0 \ldots 2]$ $\quad$ Let B(A) = 100

$$LOC(A[1][1][1]) = 100 + (1-0) + (1-0) \times (2-0+1) + (1-0) \times (2-0+1) \times (2-0+1)$$

$$= 100 + 1 + 1 \times 3 + 1 \times 3 \times 3$$

$$= 100 + 1 + 3 + 9$$

$$= 113$$ Ans.

Q4. Write the Algorithm for Binary Search & explain its steps:-

Binary Search Algorithm :-

Works on sorted array & repeatedly divides the search interval in half to find the target element efficiently.

Algorithm :-

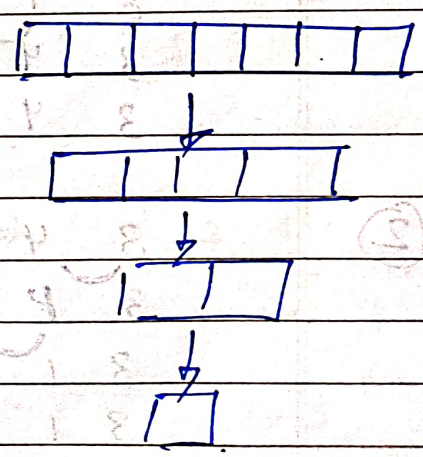
1) Start with low index = 0 & upper index high = n-1
2) Find mid index,
   mid = (low + high) / 2 or,
   (optimized) ⇒ mid = low + (high - low)/2
3) Compare the mid element arr[mid] with target
   → arr[mid] == Key, element found, return mid
   → arr[mid] > Key, search left, high = mid - 1
   → arr[mid] < Key, search right, low = mid + 1
4) Repeat step 2 and 3 until (low > high)
5) IF Key NOT found, return -1.

Time Complexity :-

$n/2^0$	n	$\frac{n}{2^k} = 1$
↓	↓	
$n/2^1$	$n/2$	$n = 2^k$
↓	↓	$\log n = k \log 2$
$n/2^2$	$n/4$	$\log_2 n = k$
⋮	⋮	
$n/2^k$	1	

k operations (from n to 1)

T.C. : O(log n)



Date / /
Page No.

Q5. Describe Bubble Sort & Explain it's working with example :-

Bubble Sort :-

Simple comparison - based sorting algorithm that repeatedly steps through the lists, compares adjacent elements & swap them if they are in wrong order.
After each pass, the largest element bubbles up to it's correct position.

Algorithm :-

1) Start from first element and compare with next one.
2) If current element isn't in the correct order, 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 5 1 2
3 4 1 5 2
3 4 1 2 5

(2) 3 4 1 2 5
3 1 4 2 5
3 1 2 4 5
3 1 2 4 5

(3) 1 3 2 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

⇒ [1, 2, 3, 4, 5]
Hence, Sorted

Date / /
Page No.

Time Complexity :-

Best : $O(n)$

Avg / Worst : $O(n^2)$

Space Complexity :- $O(1)$

Code :-

```
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 :-

			index
	0 1 2 3		
	[2, 3, 1, 0]		$i = 0 \to 3$
Pass 1 →	$2 < 3$	no swap	$j = 0 \to 3-i$
	$3 > 1$	swap	
	$3 > 0$	swap.	⇒ [2,1,0,3]
Pass 2 →	$2 > 1$	swap	
	$2 > 0$	swap	
	$2 < 3$	NO swap	⇒ [1,0,2,3]
Pass 3 →	$1 > 0$	swap	
	$1 < 2$	NO swap	
	$2 < 3$	No swap	⇒ [0,1,2,3]

↳ sorted

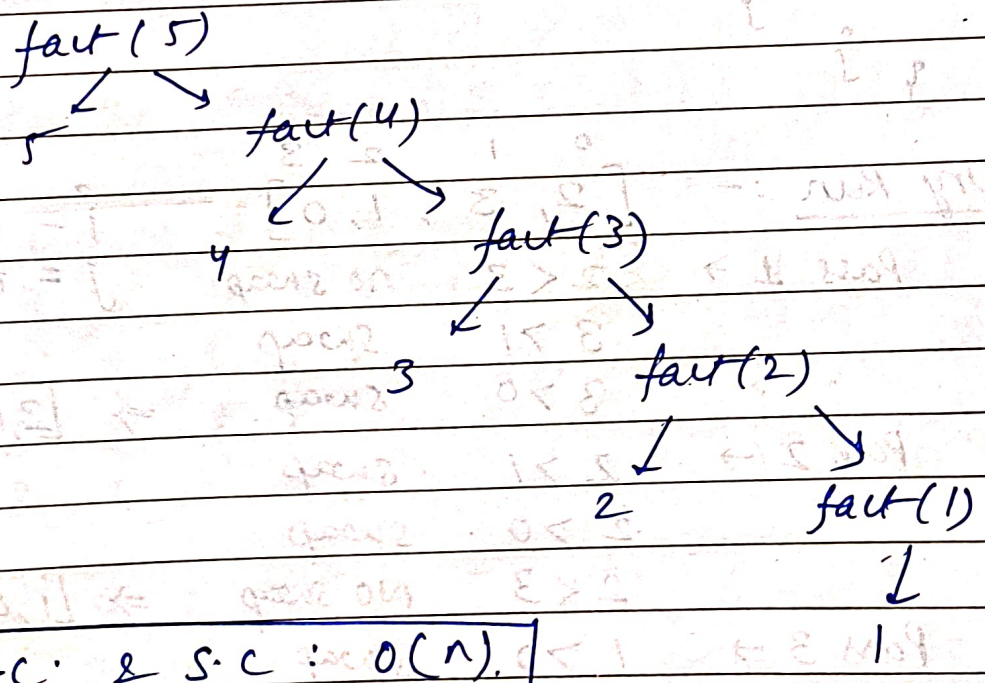
Date / /
Page No.

Q6. Write a recursive method for calculating factorial of a number and explain it.

Ans 6. **Algorithm :-**

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);
}
```



T.C. & S.C : O(n).

Date / /
Page No.

Q7. Explain circular linked list & write the algorithm to traverse it.

Circular linked List :-

A variation of a linked list where the last 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 →

- traverse the list starting from any node.
- Useful in buffered systems, round robin scheduling.

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 CLL)

1) Start with head node.
2) Use a temp pointer & move through the list
   Print temp.data
   move temp = temp.next
3) Stop when temp becomes equal to head again.

Date / /
Page No.

Pseudocode :-

```
int data ; Node * next ;
Void traverse ( Node* head) {
    if (head ==NULL) return ;

    Node * temp = head ;
    do {
        Cout << temp → data << " " ;
        temp = temp → next ;
    } while (temp != head) ;

    cout << endl ;
}
```

Q8. (a) Discuss the trade-offs between recursion & iteration with reference to 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)
* Simpler & more intuitive	* Complex to implement.
* May cause stack overflow for large inputs.	* Iteration is safer for large inputs.

Date / /
Page No.

(b) Write a recursive algorithm for the fibonacci series.

0, 1, 1, 2, 3, 5, 8, - - -

Algorithm :-

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);
}
```

Q9. Merge Sort on an Unsorted Array :-

- divide and conquer sorting algorithm.
- split the array into two halves
- recursively sort both halves
- merge those two sorted halves into a single array.

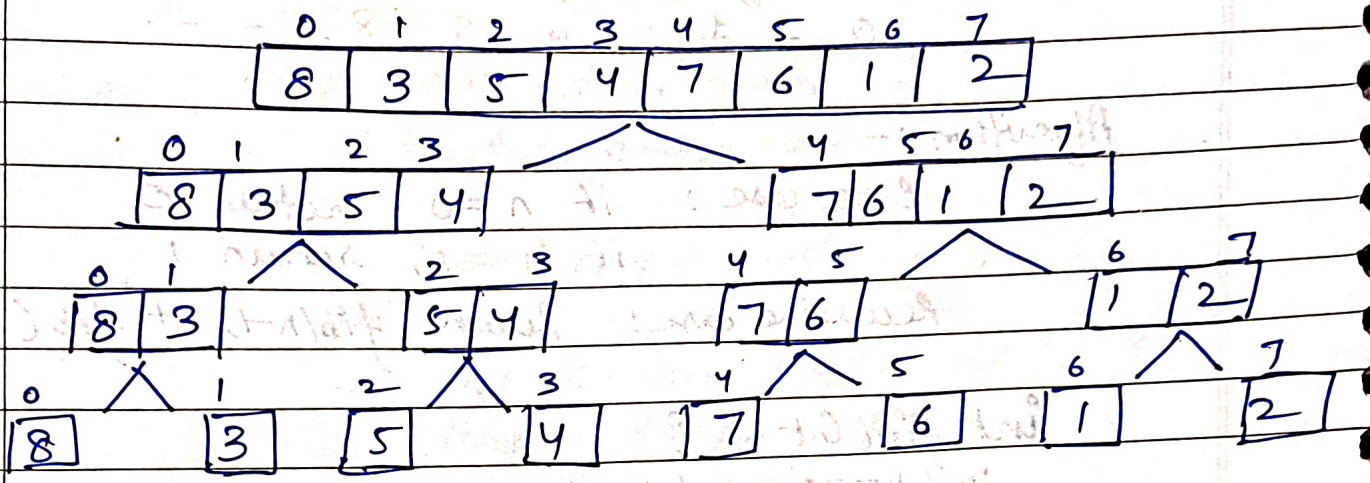
Time Complexity → O(nlogn) — even in worst case.
Space Complexity → O(n)

Real World Applications →

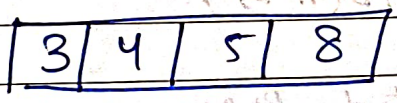
1. External sorting — sorting data too large to fit in memory
2. large scale analytics — Big data frameworks to sort massive datasets.
3. DBMS — sorting records for merge-joins & query optimizations.
4. Linked List sorting — works well as doesn't require random access.

Date / /
Page No.

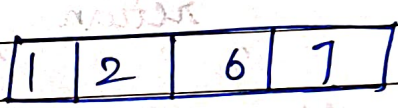
Example → arr = [8, 3, 5, 4, 7, 6, 1, 2]



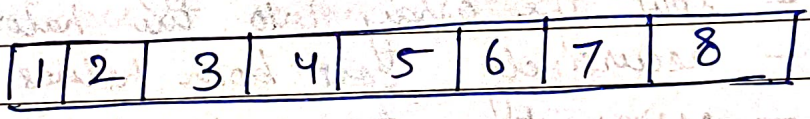
Left Half after sorting



Right half after sorting



Merge both halves :-



Algorithm :-

```
Mergesort(arr, left, right) {
  if (left < right) {
  mid = (left + right)/2
  Mergesort(arr, left, ~~right~~ mid)
  Merge sort(arr, mid+1, right)
  Merge(arr, left, mid, right)
```

PAGE No
DATE

Set-4

1. Define algorithm and explain importance of efficiency in algorithm design.

Ans An Algorithm is a finite set of well-defined, step-by-step instructions designed to perform a specific task or solve a particular problem.

Importance of Efficiency in Algorithm Design:

1) Time Optimization - Reduce the execution time
2) Space Optimization - Algorithm must be memory-efficient
3) Scalability - Efficient algorithms can handle large datasets without a significant increase in computation time.
4) Cost and Energy Saving - Lower computational costs & energy consumpⁿ.
5) USER Experience:

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

Ans **Row-major representation-**

All the elements of the first row are stored in consecutive memory locations, followed by the elements of the second row, & so-on.

Eg. Consider a 2D array A[3][3]

Row/col	0	1	2
0	1	2	3
1	4	5	6
2	9	8	9

→ Row-Major Storage Sequence:
→ 1,2,3,4,5,6,9,8,9

Memory Mapping Formula (Row-major):

$$LOC(A[I][J]) = Base(A) + [I \times N + J] \times W$$

Column-Major Repr^n:

All the elements of the first column are stored in consecutive memory location, followed by the elements of the second column & so-on.

Eg Array [3][5]:

Row/col	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

→ Col^m-Major Storage Seq:
→ 1, 4, 7, 2, 5, 8, 3, 6, 9

Memory Mapping formula:

$$LOC(A[i][j]) = Base(A) + [j \times M + i] \times W$$

Q3] Derive the general index formula for multi-dimensional arrays?

Ans Let A be a k-dimensional array with dimension size.

$$D_1, D_2, \ldots, D_k \quad (0 \le j_j < D_j \text{ for } j = 1 \ldots k)$$

PAGE No
DATE

4] Compare insertion sort and Selection sort with respect to their time complexity?

Ans

	Insertion Sort	Selection Sort
Best Case Time	$O(n)$	$O(n^2)$
Average Case Time	$O(n^2)$	$O(n^2)$
Worst Case Time	$O(n^2)$	$O(n^2)$
Space Complexity	$O(1)$	$O(1)$
Stability	Stable	Not-stable
Adaptiveness	Adaptive	Non-Adaptive
No of Comparison	Varies with input order	Always fixed $=n(n-1)/2$
No of Swaps	Up to $O(n^2)$	Atmost $(n-1)$
Practical	Efficient for small or nearly sorted data.	Used when memory writes are costly.

5] Explain merge sort algorithm with an example.

Ans Merge sort is a divide and conquer sorting algorithm that:

1) Divide the array into two halves.
2) Sorts each half recursively, and then
3) Merges the two sorted halved into one sorted array

Algorithm steps

1) Divide - Split the array into two halves - left and right
2) Conquer - Recursively sort the two halves using merge sort.
3) Combine (Merge) - Merge the two sorted halves into a single sorted array.

PAGE No
DATE

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

Ans Drawback of Recursion-

i) Extra memory usage - Each recursive call add a new stack-frame.
ii) Functional call overhead. - (consume time during each call)
iii) Risk of stack overflow - Deep recursion can exceed the system's call stack limit.

Removal Of Recursion-

- Use loops (iteration) instead of recursive calls.
- Use an explicit stack or queue (data structure).

Recursive f^n Pseudocode

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

Equivalent Iterative algorithm.

```
Algorithm Iterative_factorial (n)
1. fact ← 1
2. for i ← 1 to n do
3.     fact ← fact * i
4. return fact
```

Ex Execution :

fact = 1
i=1 ⟶ fact = 1 x 1 = 1
i=2 ⟶ fact = 1 x 2 = 2
i=3 ⟶ fact = 2 x 3 = 6
i=4 ⟶ fact = 6 x 4 = 24
i=5 ⟶ fact = 24 x 5 = 120.

PAGE No
DATE

7] Explain insertion operation in singly linked list with an example

Ans A singly linked list is a linear data structure where each element (called a node) consist two part:
1) Data - The value stored in the node
2) Next Pointer - A reference (or address) to the next node in the list.

Types of Insertion Operations-

a) Insertion at the beginning
b) Insertion at the end
c) Insertion after a given node (in the middle)

(a) Insertion at the beginning

Steps
1) Create a new node.
2) Set its next pointer to the current head.
3) Move the head pointer to the new node.

Pseudocode
```
Create newNode
newNode.data = value
newNode.next = head
head = newNode
```

(b) Insertion at the End

Steps
1) Create a new node & set its next to NULL.
2) Traverse the list to find the last node.
3) Set last node's next pointer to new node.

Pseudocode
```
Create newNode
newNode.data = value
newNode.next = NULL
if head == NULL:
    head == newNode:
else: temp = head
while temp.next != NULL
temp = temp.next
temp.next = newNod
```

PAGE No
DATE

(c) Insertion After a Given Node (Middle).

Steps:

1. Transverse: find node after which you want to insert.
2. Create a new node
3. Set newNode.next = target.next
4. Set target.next = newNode

Pseudocode:

```
Create newNode
newNode.data = value
newNode.next = target.next
target.next = newNode
```

Q9 With an example, explain sparse matrix representation using a 2-D array & discuss it advantage & limitation

Ans A sparse matrix is a matrix that contains mostly zero elements and only a few non-zero elements.

Eg Consider a 4x5 matrix

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

Advantage:

i) Memory Efficient
ii) Faster Computation
iii) Easy to store & Transfer
iv) Useful for Large Data

Limitation

1) complex Implementation
2) Slower Random access
3) Overhead for Dense Matrices