

G UDAY SIVA SAI

192425259

AI&ML

CSA0315

DATA STRUCTURE

DATA STRUCTURE OBSERVATION

# 1.Matrix Multiplication

## Aim

To write a C program to perform matrix multiplication.

## Observation

Matrix multiplication of A[m][n] and B[n][p] gives C[m][p].

## Algorithm

1. Start.
2. Read the dimensions and elements of matrix A and B.
3. Check if multiplication is possible (columns of A = rows of B).
4. Perform multiplication using triple nested loop.
5. Store result in matrix C.
6. Display the resultant matrix.
7. Stop.

A screenshot of a computer

AI-generated content may be incorrect.

## Result

The program was successfully executed and the product of two matrices was obtained.

# 2.Odd or Even Numbers

## Aim

To write a C program to find odd or even numbers from a given set of numbers.

## Observation

Odd numbers leave remainder 1 when divided by 2; even numbers leave remainder 0.

## Algorithm

1. Start.
2. Read the number of elements.
3. Read the set of numbers.
4. Check each number using modulus operator %2.
5. Print whether it is Odd or Even.
6. Stop.

A screenshot of a computer

AI-generated content may be incorrect.

## Result

The program was successfully executed and the given set was classified as odd or even.

# 3.Factorial without Recursion

## Aim

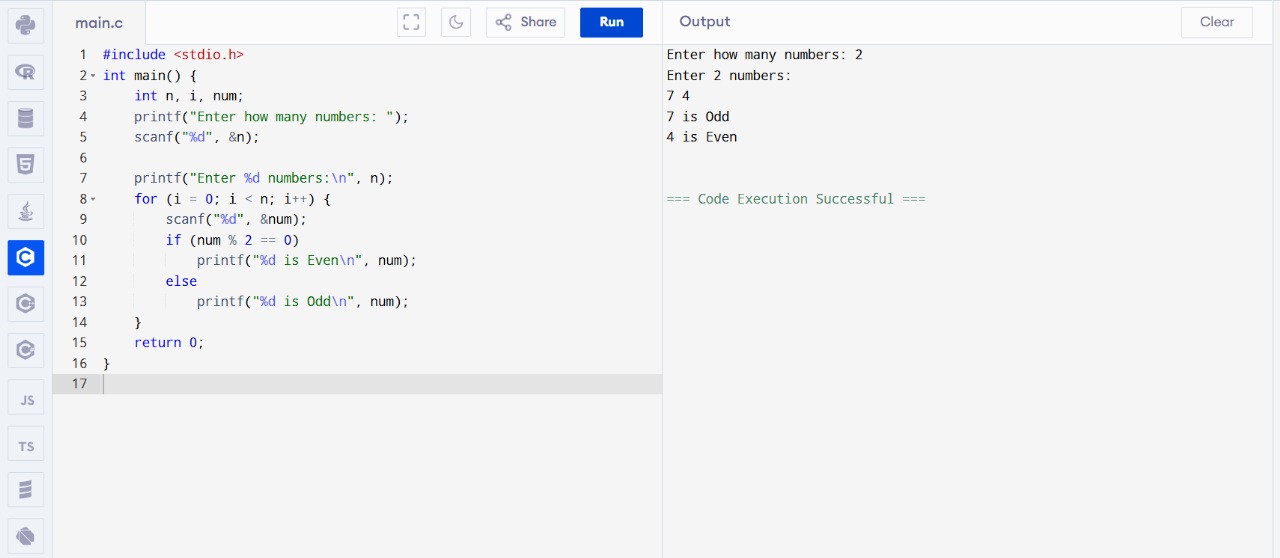
To write a C program to find the factorial of a given number without using recursion.

## Observation

Factorial n! = n × (n-1) × ... × 1.

## Algorithm

1. Start.
2. Read integer number n.
3. Initialize fact=1.
4. Multiply fact by numbers from 1 to n.
5. Print fact.
6. Stop.



## Result

The program was successfully executed and factorial was found without recursion.

# 4.Fibonacci without Recursion

## Aim

To write a C program to print Fibonacci series without recursion.

## Observation

Fibonacci sequence: 0,1,1,2,3,... where f(n)=f(n-1)+f(n-2).

## Algorithm

1. Start.
2. Read number of terms n.
3. Initialize first two terms 0 and 1.
4. Use loop to compute next terms.
5. Display series.
6. Stop.

A screenshot of a computer

AI-generated content may be incorrect.

## Result

The program was successfully executed and Fibonacci series was generated without recursion.

# 5.Factorial using Recursion

## Aim

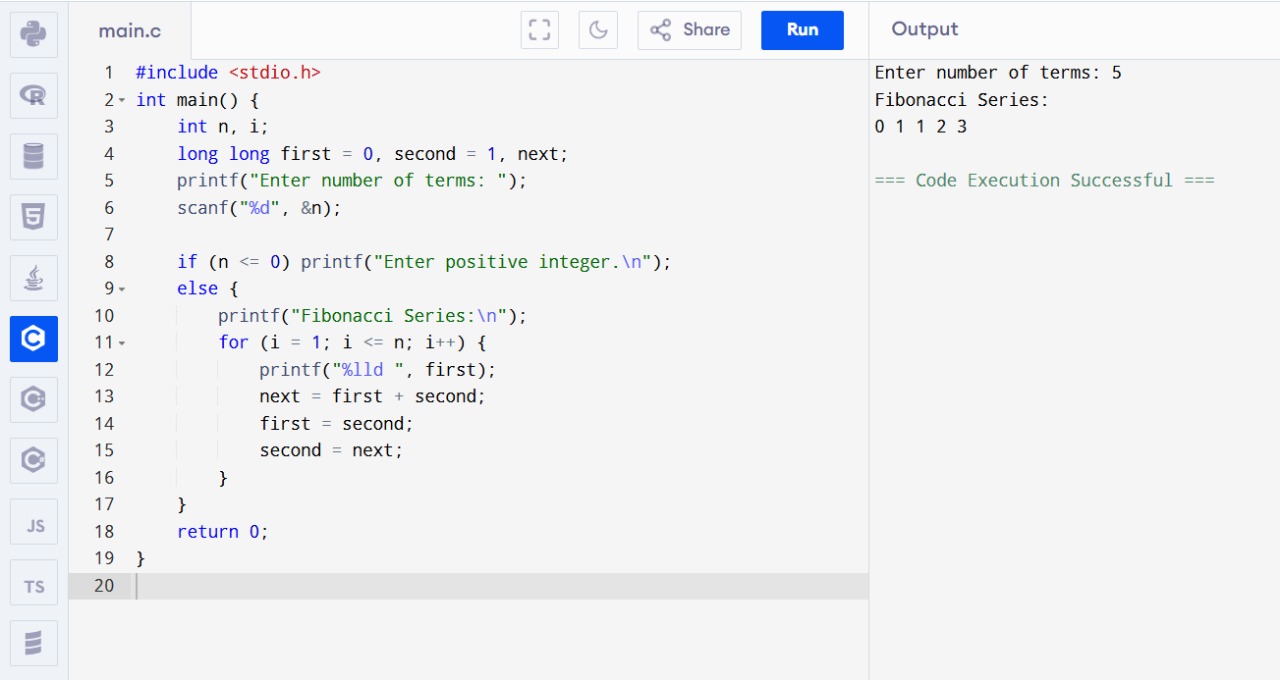
To write a C program to find the factorial of a given number using recursion.

## Observation

Factorial of a number n is defined as n! = n × (n-1) × ... × 1.

## Algorithm

1. Start.
2. Read integer number n.
3. Define recursive function fact(n).
4. If n==0 or n==1 return 1 else return n\*fact(n-1).
5. Call function and print result.
6. Stop.



## Result

The program was successfully executed and factorial was found using recursion.

# 6.Fibonacci using Recursion

## Aim

To write a C program to print Fibonacci series using recursion.

## Observation

Fibonacci sequence is generated using recursive relation f(n)=f(n-1)+f(n-2).

## Algorithm

1. Start.
2. Read n.
3. Define recursive function fib(n).
4. If n==0 return 0, if n==1 return 1.
5. Else return fib(n-1)+fib(n-2).
6. Call function for first n terms.
7. Display result.

A screenshot of a computer

AI-generated content may be incorrect.

## Result

The program was successfully executed and Fibonacci series was generated using recursion.

# 7.Array Operations (Insert, Delete, Display)

## Aim

To write a C program to implement array operations such as insert, delete, and display.

## Observation

Arrays are fixed-size collections. We can insert, delete and display elements using index-based operations.

## Algorithm

1. Start.
2. Declare array and size.
3. Insert element at given position.
4. Delete element from given position.
5. Display array elements.

**A screenshot of a computer

AI-generated content may be incorrect.**

## +

## Result

The program was successfully executed and array operations were performed.

# 8.Linear Search

## Aim

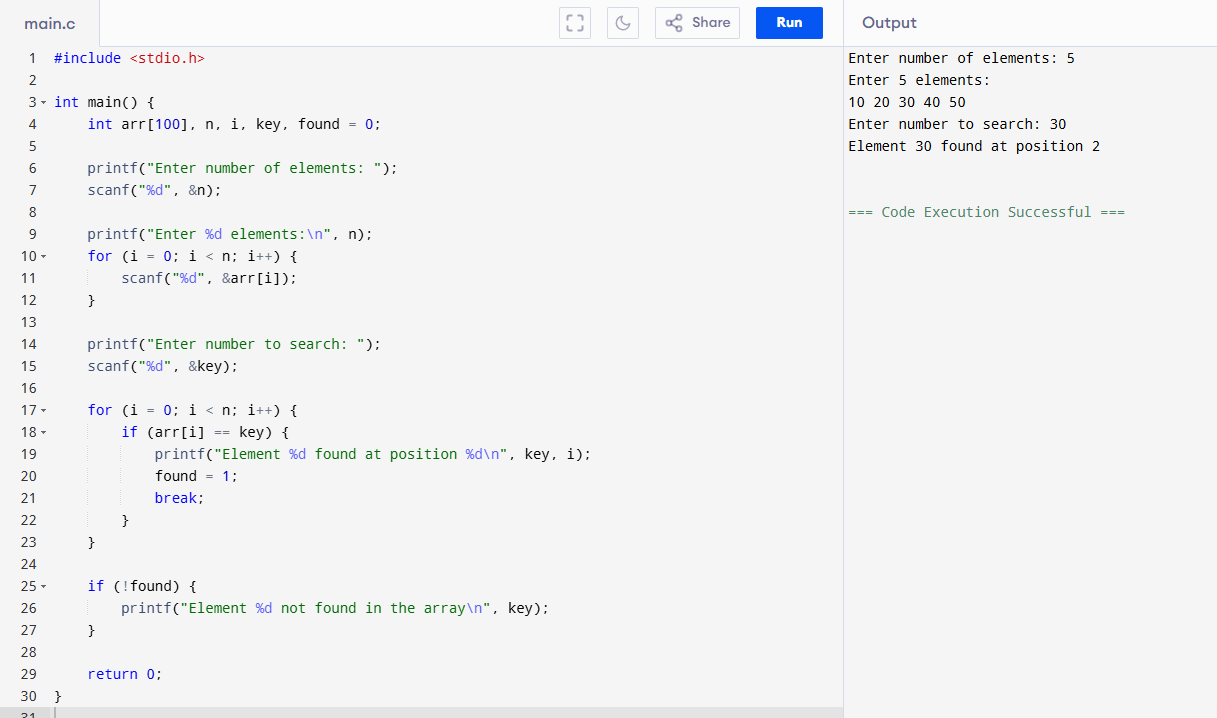
To write a C program to search a number using Linear Search method.

## Observation

Linear search compares each element until key is found.

## Algorithm

1. Start.
2. Read array elements and key.
3. Scan array sequentially.
4. If element matches key, print position.
5. If not found, print not found.
6. Stop.



## Result

The program was successfully executed and the element was searched using Linear Search.

# 9.Binary Search

## Aim

To write a C program to search a number using Binary Search method.

## Observation

Binary search works on sorted array by repeatedly dividing the search interval in half.

## Algorithm

1. Start.
2. Read sorted array and key.
3. Initialize low=0, high=n-1.
4. While low<=high compute mid.
5. If key==a[mid], found.
6. Else if key<a[mid], high=mid-1 else low=mid+1.
7. Stop.

## A screenshot of a computer program AI-generated content may be incorrect.

## Result

The program was successfully executed and the element was searched using Binary Search.

## ****10. Stack Operations (PUSH, POP, PEEK)****

**Aim:** To implement stack operations using arrays.

**Observation:** The stack follows **LIFO (Last In First Out)** principle.  
  
**Algorithm:**

1. Initialize top = -1.
2. For PUSH: increment top, insert element.
3. For POP: remove element at top, decrement top.
4. For PEEK: display element at top.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully implemented stack operations.

## ****11.. Application of Stack (Notations)****

**Aim:** To evaluate postfix/prefix expressions using stack.

**Observation:** Stack helps in expression evaluation.  
**Algorithm:**

1. Read expression from left to right.
2. If operand, push onto stack.
3. If operator, pop operands, evaluate, push result.
4. At end, top of stack gives final answer.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully evaluated expressions using stack.

## ****12. Queue Operations (ENQUEUE, DEQUEUE, Display)****

**Aim:** To implement queue operations using arrays.

**Observation:** Queue follows **FIFO (First In First Out)** principle.  
  
**Algorithm:**

1. Initialize front=0, rear=-1.
2. ENQUEUE: Increment rear, insert element.
3. DEQUEUE: Remove element at front, increment front.
4. Display: Print elements between front and rear.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully implemented queue operations.

## ****13. Tree Traversals (Inorder, Preorder, Postorder)****

**Aim:** To implement different tree traversal techniques.

**Observation:** Different traversal orders produce different sequences.  
  
**Algorithm:**

* **Inorder (L, Root, R)**
* **Preorder (Root, L, R)**
* **Postorder (L, R, Root)**

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully performed tree traversals.

## ****15. Hashing using Linear Probing****

**Aim:** To implement hashing with linear probing.

**Observation:** Collisions are resolved by linear probing.  
  
**Algorithm:**

1. Compute hash = key % table\_size.
2. If slot empty, insert key.
3. If occupied, move linearly until empty slot found.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully implemented hashing with collision handling.

## ****16. Insertion Sort****

**Aim:** To sort a list using insertion sort.

**Observation:** Works well for small datasets.  
**Algorithm:**

1. Start from second element.
2. Compare with previous elements.
3. Insert into correct position by shifting.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully sorted numbers using insertion sort.

## ****17. Merge Sort****

**Aim:** To sort numbers using merge sort

**Observation:** Efficient sorting using divide & conquer..  
**Algorithm:**

1. Divide array into halves recursively.
2. Sort each half.
3. Merge sorted halves.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully sorted numbers using merge sort.

## ****18. Quick Sort****

**Aim:** To sort numbers using quick sort.

**Observation:** Works efficiently for large datasets.  
**Algorithm:**

1. Choose pivot element.
2. Partition array into left < pivot, right > pivot.
3. Recursively sort partitions.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully sorted numbers using quick sort.

## ****19. Heap Sort****

**Aim:** To sort numbers using heap sort.

**Observation:** Sorting done using heap property.  
  
**Algorithm:**

1. Build max heap.
2. Swap root with last element.
3. Heapify remaining elements.
4. Repeat until sorted.

A screenshot of a computer program

AI-generated content may be incorrect.

**Result:** Successfully sorted numbers using heap sort.

## ****20. AVL Tree Operations (Insert, Delete, Search)****

**Aim:** To implement insertion, deletion, and search in AVL tree.

**Observation:** Tree remains balanced.  
**Algorithm:**

1. Insert node like in BST.
2. Update balance factor.
3. If imbalance occurs, perform rotations.
4. For delete, adjust and rebalance tree.
5. For search, traverse based on key.  
   A screenshot of a computer

   AI-generated content may be incorrect.

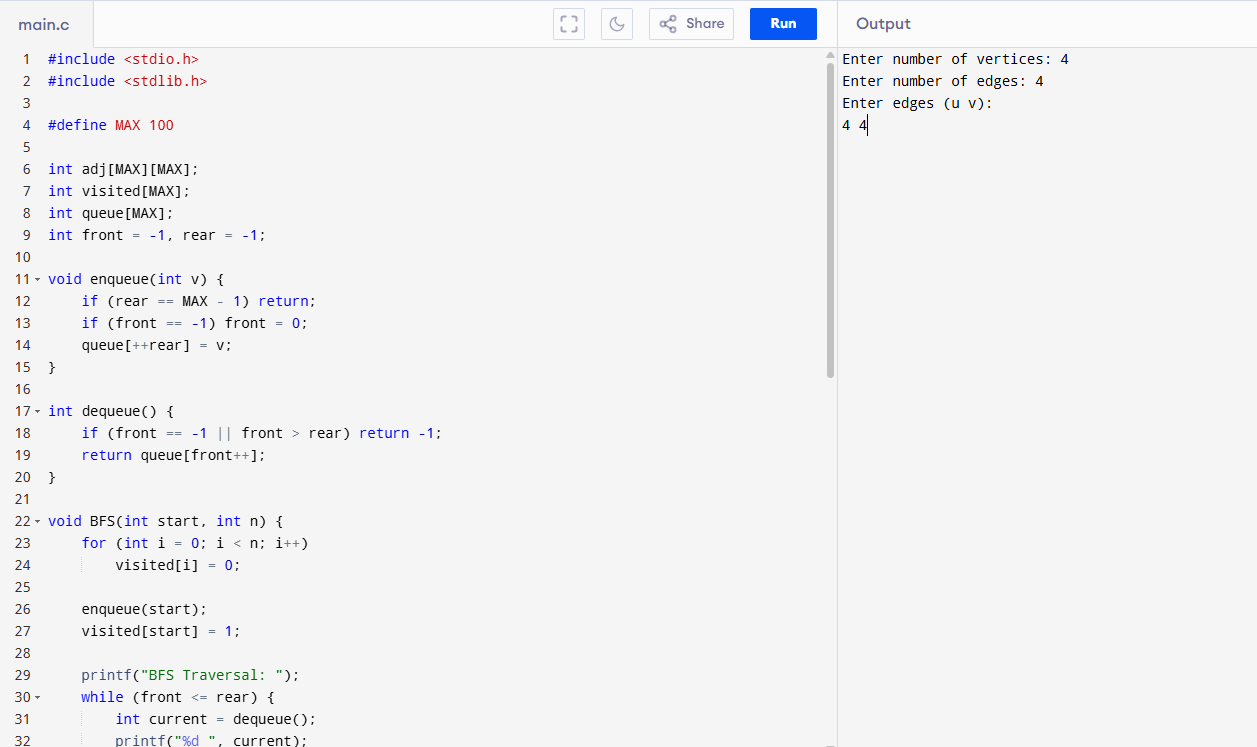
**Result:** Successfully implemented AVL tree operations.

## ****21. Graph Traversal (BFS)****

**Aim:** To traverse graph using Breadth First Search.

**Observation:** Visits nodes level by level.  
  
**Algorithm:**

1. Initialize queue.
2. Visit source node, mark visited.
3. Enqueue neighbors, dequeue front.
4. Repeat until queue empty.

****

**Result:** Successfully traversed graph using BFS.

## ****22. Graph Traversal (DFS)****

**Aim:** To traverse graph using Depth First Search.

**Observation:** Traverses deeper before backtracking.  
**Algorithm:**

1. Use stack/recursion.
2. Visit source node, mark visited.
3. Recursively visit unvisited neighbors.
4. Backtrack when no unvisited neighbor exists.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully traversed graph using DFS.

## ****23. Dijkstra’s Algorithm (Shortest Path)****

**Aim:** To find shortest path in weighted graph using Dijkstra’s algorithm.

**Observation:** Always finds shortest path in weighted graphs.  
**Algorithm:**

1. Initialize distance array.
2. Start from source, set distance=0.
3. Update neighbors with minimum distance.
4. Mark visited nodes.
5. Repeat until all nodes processed.

A screenshot of a computer

AI-generated content may be incorrect.

**Result:** Successfully implemented shortest path algorithm.

## ****24. Prim’s Algorithm (MST)****

**Aim:** To find minimum spanning tree using Prim’s algorithm.

**Observation:** Builds MST by adding one edge at a time.  
**Algorithm:**

1. Start from any node.
2. Add smallest weight edge connecting to MST.
3. Repeat until all vertices included

A screenshot of a computer

AI-generated content may be incorrect.

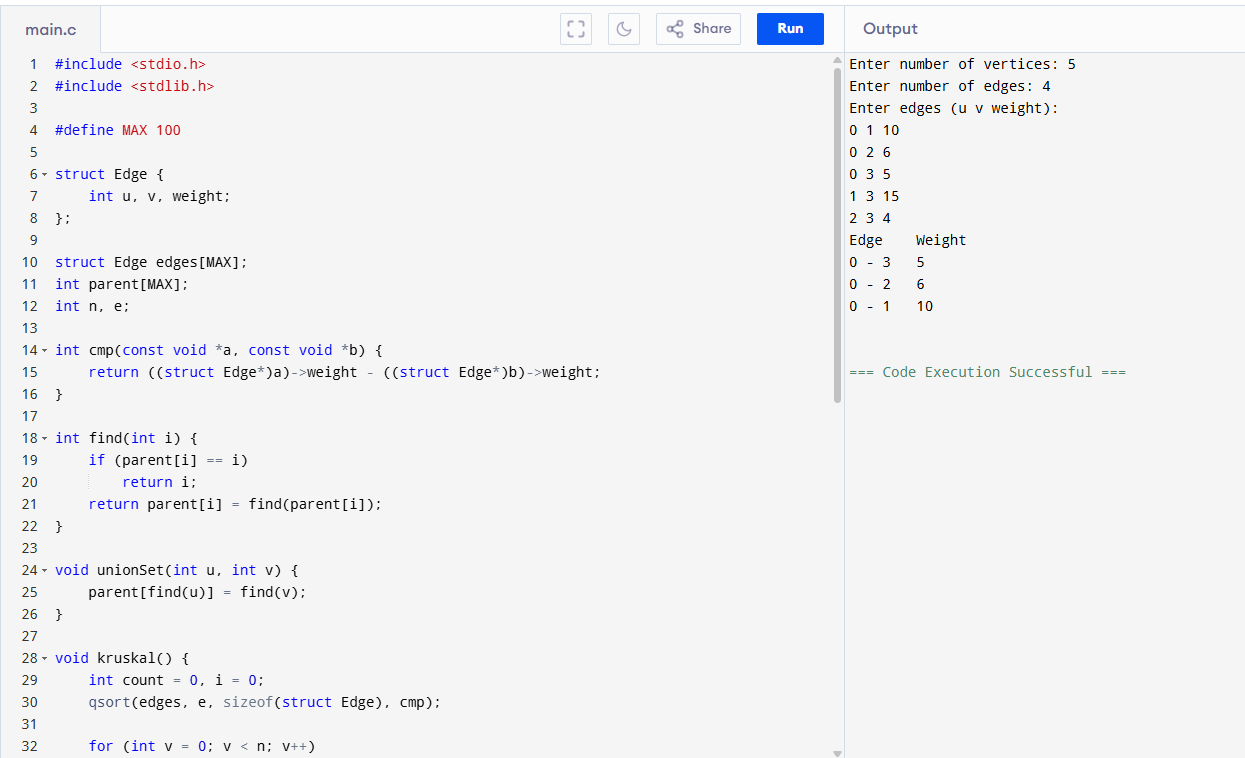
**Result:** Successfully found MST using Prim’s algorithm.

## ****25. Kruskal’s Algorithm (MST)****

**Aim:** To find minimum spanning tree using Kruskal’s algorithm.

**Observation:** MST built by adding edges in increasing weight order.  
**Algorithm:**

1. Sort edges by weight.
2. Pick smallest edge, check cycle using union-find.
3. If no cycle, add to MST.
4. Repeat until MST complete.

****

**Result:** Successfully found MST using Kruskal’s algorithm.