1. Write a c program to perform matrix multiplication

Aim: To write a c program to perform multiplication of two matrices entered by user

main.c Share Run Output

```c
#include <stdio.h>
int main() {
    int a[3][3], b[3][3], c[3][3] = {0};
    int i, j, k;
    printf("Enter 3x3 matrix A:\n");
    for(i=0;i<3;i++) for(j=0;j<3;j++) scanf("%d", &a[i][j]);
    printf("Enter 3x3 matrix B:\n");
    for(i=0;i<3;i++) for(j=0;j<3;j++) scanf("%d", &b[i][j]);
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            for(k=0;k<3;k++)
                c[i][j] += a[i][k] * b[k][j];
    printf("Result:\n");
    for(i=0;i<3;i++) {
        for(j=0;j<3;j++) printf("%d ", c[i][j]);
```

```
Enter 3x3 matrix A:
1 2 3
4 5 6
7 8 9
Enter 3x3 matrix B:
9 8 7
6 5 4
3 2 1
Result:
30 24 18
84 69 54
138 114 90


=== Code Execution Successful ===
```

Result:to perform multiplication of two matrices entered by the user is successfully executed and output is verified

2. Write a C program to find Odd or Even number from a given set of numbers

Aim: To write a c program to determine the whether number from a given set of numbers are even or odd

```c
#include <stdio.h>
int main() {
    int n, i, a[100];
    printf("How many numbers? ");
    scanf("%d", &n);
    printf("Enter %d numbers:\n", n);
    for(i = 0; i < n; i++) scanf("%d", &a[i]);
    for(i = 0; i < n; i++)
        printf("%d is %s\n", a[i], a[i] % 2 ? "Odd" : "Even");
    return 0;
}
```

```
How many numbers? 4
Enter 4 numbers:
10 5 7 8
10 is Even
5 is Odd
7 is Odd
8 is Even


=== Code Execution Successful ===
```

Result: To determine whether number is odd or even are successfully executed and output is verified

3.Write a C program to find Factorial of a given number without using Recursion
Aim : To write a c program to calculate the factorial of a given number using iterations with out recurssion

main.c

Share Run Output

```c
#include <stdio.h>
int main() {
    int n, i;
    unsigned long long fact = 1;
    printf("Enter a number: ");
    scanf("%d", &n);
    if(n < 0)
        printf("Factorial not defined for negative numbers.\n");
    else {
        for(i = 1; i <= n; i++)
            fact *= i;
        printf("Factorial of %d = %llu\n", n, fact);
    }
    return 0;
}
```

```
Enter a number: 5
Factorial of 5 = 120


=== Code Execution Successful ===
```

Result: to calculate the factorial of a given number using iteration without recursion are successfully executed and outputs is verified

4.Write a C program to find Fibonacci series without using Recursion

Aim:To write a C program to generate the Fibonacci series up to n terms using iteration (no recursion).

main.c

Share Run Output

```c
#include <stdio.h>
int main() {
    int n, a = 0, b = 1, c, i;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci series: ");
    for(i = 0; i < n; i++) {
        printf("%d ", a);
        c = a + b;
        a = b;
        b = c;
    }
    return 0;
}
```

```
Enter number of terms: 6
Fibonacci series: 0 1 1 2 3 5

=== Code Execution Successful ===
```

Result:To write a C program to generate the Fibonacci series up to n terms using iteration (no recursion). Is successfully executed and output is verified

5.Write a C program to find Factorial of a given number using Recursion

Aim: To write a C program to calculate the factorial of a number using a recursive function.

```c
#include <stdio.h>
unsigned long long factorial(int n) {
    if(n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    if(n < 0)
        printf("Factorial is not defined for negative numbers
            .\n");
    else
```

```
Enter number of terms: 6
Fibonacci series: 0 1 1 2 3 5

=== Code Execution Successful ===
```

Result:To write a C program to calculate the factorial of a number using a recursive function. Output executed successfully and output is verified

6.Write a C program to find Fibonacci series using Recursion

Aim: To write a C program to print the Fibonacci series up to n terms using recursion.

main.c Share Run Output

```c
#include <stdio.h>
int fibonacci(int n) {
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int n, i;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci series: ");
    for(i = 0; i < n; i++)
```

```
Enter number of terms: 6
Fibonacci series: 0 1 1 2 3 5

=== Code Execution Successful ===
```

Result: To write a C program to print the Fibonacci series up to n terms using recursion. Code is successfully executed and output is verified

7.Write a C program to implement Array operations such as Insert, Delete and Display

Aim: To write a C program to implement basic array operations such as insertion, deletion, and display of elements.

main.c

Share Run

Output

```c
#include <stdio.h>
int arr[100], size = 0;
void display() {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
void insert(int val, int pos) {
    if (pos < 0 || pos > size) return;
    for (int i = size; i > pos; i--)
        arr[i] = arr[i - 1];
    arr[pos] = val;
    size++;
}
void delete(int pos) {
```

```
1.Insert 2.Delete 3.Display 4.Exit: 3

1.Insert 2.Delete 3.Display 4.Exit: 10


=== Code Execution Successful ===
```

Result: to implement basic array operations such as insertion, deletion, and display of elements. Is executed successfully output is verified

8.Write a C program to search a number using Linear Search method

Aim: To write a C program to search for an element in an array using the **Linear Search** method.

```c
#include <stdio.h>
int main() {
    int arr[100], n, key, i, found = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter element to search: ");
    scanf("%d", &key);
    for (i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Element found at index %d.\n", i);
            found = 1;
            break;
```

```
Enter number of elements: 5
Enter array elements:
10 20 30 40 50
Enter element to search: 30
Element found at index 2.


=== Code Execution Successful ===
```

Result: to search for an element in an array using the **Linear Search** method. Code executed successfully output is verified

9.Write a C program to search a number using Binary Search method
Aim: To write a C program to search an element in a sorted array using the Binary Search method.

main.c

Share Run

Output

```c
#include <stdio.h>
int binarySearch(int a[], int n, int key) {
    int l = 0, h = n - 1;
    while (l <= h) {
        int m = (l + h) / 2;
        if (a[m] == key) return m;
        else if (a[m] < key) l = m + 1;
        else h = m - 1;
    }
    return -1;
}
int main() {
    int a[] = {2, 4, 6, 8, 10}, key = 6;
    int n = sizeof(a) / sizeof(a[0]);
    int res = binarySearch(a, n, key);
```

```
Found at index 2


=== Code Execution Successful ===
```

Result : To search an element in a sorted array using the Binary Search method. Is executed successfully and output is verified

10. Write a C program to implement Linked list operations
Aim: To write a C program to perform basic operations on a **singly linked list**

main.c

Share Run

Output

```c
#include <stdio.h>
#include <stdlib.h>
struct Node { int data; struct Node* next; } *head = NULL;
void insert(int val) {
    struct Node* n = malloc(sizeof(struct Node)), *t = head;
    n->data = val; n->next = NULL;
    if (!head) head = n;
    else { while (t->next) t = t->next; t->next = n; }
}
void display() { for (struct Node* t = head; t; t = t->next)
    printf("%d -> ", t->data); printf("NULL\n"); }
int main() {
    insert(10); insert(20); insert(30);
    display();
    return 0;
}
```

```
10 -> 20 -> 30 -> NULL


=== Code Execution Successful ===
```

Result: To perform basic operations on a singly linked list is successfully executed output is **verified**

11.Write a C program to implement Stack operations such as PUSH,POP and PEEK.

Aim: To implement stack operations (PUSH, POP, PEEK) using arrays in C.

main.c

Share Run

```c
#include <stdio.h>
#define SIZE 5
int stack[SIZE], top = -1;
void push(int val) { if (top < SIZE-1) stack[++top] = val; else
    printf("Overflow\n"); }
void pop() { if (top >= 0) printf("Popped: %d\n", stack[top--]);
    else printf("Underflow\n"); }
void peek() { if (top >= 0) printf("Top: %d\n", stack[top]); else
    printf("Empty\n"); }
int main() {
    push(10); push(20); peek(); pop(); peek(); pop(); pop();
    return 0;
}
```

Output

```
Top: 20
Popped: 20
Top: 10
Popped: 10
Underflow

=== Code Execution Successful ===
```

Result: To implement stack operations (PUSH, POP, PEEK) using arrays in C. is successfully executed and output is verified

12. Write a C program to implement the application of Stack (Notations)

Aim: To implement infix to postfix conversion using stack data structure in c

main.c

Share Run

```c
#include <stdio.h>
#include <ctype.h>
char stack[100];
int top = -1;
void push(char c) { stack[++top] = c; }
char pop() { return stack[top--]; }
int prec(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
int main() {
    char infix[100], c;
    int i = 0;
    printf("Enter Infix: ");
    scanf("%s", infix);
    printf("Postfix: ");
    while ((c = infix[i++]) != '\0') {
```

Output

```
Enter Infix: (a+b)*c
Postfix: ab+c*

=== Code Execution Successful ===
```

Result: To implement infix to postfix conversion using stack data structure in c is executed successfully and output is verified

13.Write a C program to implement Queue operations such as ENQUEUE, DEQUEUE and Display

Aim:To implement basic queue operations — **ENQUEUE**, **DEQUEUE**, and **DISPLAY** — using arrays in C.

main.c

Share Run Output

```c
#include <stdio.h>
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;
void enqueue(int val) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1) front = 0;
    queue[++rear] = val;
    printf("%d enqueued.\n", val);
}
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return;
```

```
1.Enqueue 2.Dequeue 3.Display 4.Exit
Choice: 1
Enter value: 20
20 enqueued.

1.Enqueue 2.Dequeue 3.Display 4.Exit
Choice: 2
20 dequeued.

1.Enqueue 2.Dequeue 3.Display 4.Exit
Choice: 2
Queue Underflow

1.Enqueue 2.Dequeue 3.Display 4.Exit
Choice:
=== Session Ended. Please Run the code again ===
```

Result: To implement basic queue operations — **ENQUEUE**, **DEQUEUE**, and **DISPLAY** — using arrays in C. are executed successfully and output is verified

14. Write a C program to implement the Tree Traversals (Inorder, Preorder,postorder)

Aim:To implement and demonstrate **Inorder**, **Preorder**, and **Postorder** traversals of a binary tree in C.

main.c

Share Run Output

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *l, *r;
};
struct Node* n(int v) {
    struct Node* node = malloc(sizeof(struct Node));
    node->data = v; node->l = node->r = NULL;
    return node;
}
void inorder(struct Node* t) {
    if (!t) return;
    inorder(t->l);
    printf("%d ", t->data);
    inorder(t->r);
}
```

```
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1


=== Code Execution Successful ===
```

Result:To implement and demonstrate **Inorder**, **Preorder**, and **Postorder** traversals of a binary tree in C. is executed successfully and output is verified

15. Write a C program to implement hashing using Linear Probing method

Aim: To implement **hashing with linear probing** to resolve collisions in a hash table.

main.c

Share Run Output

```c
#include <stdio.h>
#define SIZE 10
int hashTable[SIZE];
void init() {
    for(int i=0; i<SIZE; i++) hashTable[i] = -1;
}
int hash(int key) { return key % SIZE; }
void insert(int key) {
    int i = hash(key);
    while(hashTable[i] != -1) i = (i+1) % SIZE;
    hashTable[i] = key;
}
int search(int key) {
    int i = hash(key), start = i;
    while(hashTable[i] != -1) {
        if(hashTable[i] == key) return i;
        i = (i+1) % SIZE;
        if(i == start) break;
```

```
Enter number of elements: 5
Enter elements:
12 22 32  42 52
Index Key
0   -1
1   -1
2   12
3   22
4   32
5   42
6   52
7   -1
8   -1
9   -1
Enter key to search: 42
Key found at index 5
```

Result: To implement **hashing with linear probing** to resolve collisions in a hash table. Is successfully implemented and output is verified

16.Write a C program to arrange a series of numbers using Insertion Sort
Aim: To write a C program that arranges a series of numbers in **ascending order** using the **Insertion Sort** algorithm.

main.c

Share Run Output

```c
#include <stdio.h>
int main() {
    int arr[100], n, i, j, key;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for(i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while(j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
```

```
Enter the number of elements: 6
Enter 6 integers:
4 5 7 8 4 2
Sorted array in ascending order:
2 4 4 5 7 8

=== Code Execution Successful ===
```

Result: arranges a series of numbers in **ascending order** using the **Insertion Sort** algorithm. Is successfully executed and output is verified

17.Write a C program to arrange a series of numbers using Merge Sort

Aim:To write a program that sorts a series of numbers in ascending order using the merge sort algorithm

```c
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int i = left, j = mid + 1, k = 0;
    int temp[100];
    while(i <= mid && j <= right) {
        if(arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }
    while(i <= mid)
        temp[k++] = arr[i++];
    while(j <= right)
        temp[k++] = arr[j++];
    for(i = left, k = 0; i <= right; i++, k++)
        arr[i] = temp[k];
}
void mergeSort(int arr[], int left, int right) {
```

```
Enter number of elements: 5
Enter 5 numbers:
4 5 1 6 2
Sorted array in ascending order:
1 2 4 5 6

=== Code Execution Successful ===
```

Result:sorts a series of numbers in ascending order using the merge sort algorithm successfully executed and output is verified

18. Write a C program to arrange a series of numbers using Quick Sort

Aim: To write a C program that arranges a series of numbers in **ascending order** using the **Quick Sort** algorithm.

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for(int j = low; j < high; j++) {
        if(arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i+1], &arr[high]);
    return i + 1;
}
```

```
Enter the number of elements: 6
Enter 6 integers:

7 2 1 6 8 5
Sorted array in ascending order:
1 2 5 6 7 8

=== Code Execution Successful ===
```

Result: arranges a series of numbers in **ascending order** using the **Quick Sort** algorithm. Is executed successfully output is verified

19. Write a C program to implement Heap sort

Aim: To write a C program that sorts a series of numbers in **ascending order** using the **Heap Sort** algorithm.

main.c

Share | Run

```c
#include <stdio.h>
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2
    if(left < n && arr[left] > arr[largest])
        largest = left;
    if(right < n && arr[right] > arr[largest])
        largest = right;
    if(largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}
void heapSort(int arr[], int n) {
    for(int i = n/2 - 1; i >= 0; i--)
```

Output

```
Enter the number of elements: 5
Enter 5 integers:
4 10 3 5 1
Sorted array in ascending order:
1 3 4 5 10

=== Code Execution Successful ===
```

Result:sorts a series of numbers in **ascending order** using the **Heap Sort** algorithm. Code executed successfully and output is verified

20.Write a program to perform the following operations:
a) Insert an element into a AVL tree
b) Delete an element from a AVL tree
c) Search for a key element in a AVL tree

Aim:To write a C program to perform the following operations on an **AVL Tree**:

1. Insert an element.
2. Delete an element.
3. Search for a key element.

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int key, height;
    struct Node *left, *right;
} Node;
int height(Node *n) {
    return n ? n->height : 0;
}
int getBalance(Node *n) {
    return n ? height(n->left) - height(n->right) : 0;
}
Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1;
    return node;
```

```
Menu:
1. Insert
2. Delete
3. Search
4. InOrder Traversal
5. Exit
Enter choice: 1
Enter key to insert: 50

Menu:
1. Insert
2. Delete
3. Search
4. InOrder Traversal
5. Exit
Enter choice: 1
Enter key to insert: 30
```

Result:to perform the following operations on an **AVL Tree** is executed successfully
And output is verified

21. Write a C program to Graph traversal using Breadth First Search

Aim: To implement **Breadth First Search (BFS)** traversal for a graph using C programming.

main.c Share Run

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int queue[MAX], front = -1, rear = -1;
int visited[MAX];

// Function to enqueue an element
void enqueue(int vertex) {
    if (rear == MAX - 1)
        printf("Queue Overflow\n");
    else {
        if (front == -1)
            front = 0;
        queue[++rear] = vertex;
    }
}
```

Output Clear

```
Enter number of vertices: 4
Enter adjacency matrix (4 x 4):
0 1 1 0
Enter starting vertex (0 to 3): BFS Traversal starting from vertex 0:
0

=== Code Execution Successful ===
```

Result: To implement **Breadth First Search (BFS)** traversal for a graph using C programming.
Are executed successfully and output is verified

22. Write a C program to Graph traversal using Depth First Search

Aim: To implement **Depth First Search (DFS)** traversal of a graph using C programming.

main.c Share Run Output

```c
#include <stdio.h>

#define MAX 100

int visited[MAX];

// DFS function
void dfs(int adj[MAX][MAX], int n, int start) {
    int i;
    visited[start] = 1;
    printf("%d ", start);

    for (i = 0; i < n; i++) {
        if (adj[start][i] == 1 && !visited[i]) {
            dfs(adj, n, i);
        }
    }
}
```

```
Enter number of vertices: 4
Enter adjacency matrix (4 x 4):
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter starting vertex (0 to 3): 0
DFS Traversal starting from vertex 0:
0 1 3 2

=== Code Execution Successful ===
```

Result:To implement **Depth First Search (DFS)** traversal of a graph using C programming.

Code executed successfully and output is verified

23.Implementation of Shortest Path Algorithms using Dijkstra's Algorithm

Aim:To implement **Dijkstra's Algorithm** in C to find the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

```c
#include <stdio.h>
#define MAX 100
#define INF 9999
void dijkstra(int adj[MAX][MAX], int n, int start) {
    int distance[MAX], visited[MAX], i, j, min, u;
    for (i = 0; i < n; i++) {
        distance[i] = INF;
        visited[i] = 0;
    }
    distance[start] = 0;
    for (i = 0; i < n - 1; i++) {
        min = INF;
        u = -1;
        for (j = 0; j < n; j++) {
            if (!visited[j] && distance[j] < min) {
                min = distance[j];
                u = j;
            }
```

```
Enter the number of vertices: 5
Enter the adjacency matrix (use 0 for no edge):
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 50
100 0 10 60 0
Enter the starting vertex (0 to 4): 0
Vertex  Distance from Source 0
0       0
1       10
2       50
3       30
4       60


=== Code Execution Successful ===
```

Result: code run successfully and out put is verified

24.Implementation of Minimum Spanning Tree using Prim's Algorithm

Aim:To implement **Prim's Algorithm** in C to find the **Minimum Spanning Tree (MST)** of a connected, undirected, and weighted graph.

main.c

Share Run Output

```c
#include <stdio.h>
#define MAX 100
#define INF 9999

int main() {
    int n, i, j;
    int adj[MAX][MAX];
    int visited[MAX] = {0};
    int parent[MAX], key[MAX];
    int min, u, totalCost = 0;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the adjacency matrix (use 0 for no edge):\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            scanf("%d", &adj[i][j]);
```

```
Enter the number of vertices: 5
Enter the adjacency matrix (use 0 for no edg
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0

Edges in the Minimum Spanning Tree:
0 - 1   Weight: 2
1 - 2   Weight: 3
0 - 3   Weight: 6
1 - 4   Weight: 5
Total weight of MST: 16


=== Code Execution Successful ===
```

Result:to implement **Prim's Algorithm** in C to find the **Minimum Spanning Tree (MST)** of a connected, undirected, and weighted graph is executed successfully and output is verified

25.Implementation of Minimum Spanning Tree using Kruskal Algorithm
Aim:To implement **Kruskal's Algorithm** in C for finding the **Minimum Spanning Tree (MST)** of a connected, undirected, and weighted graph.

main.c

Share Run Output

```c
#include <stdio.h>
#define MAX 100
struct Edge {
    int u, v, weight;
};
int parent[MAX];
int find(int i) {
    while (parent[i] != i)
        i = parent[i];
    return i;
}
void unionSets(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}

int main() {
```

```
Enter number of vertices: 4
Enter number of edges: 5
Enter edges (u v weight):
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4

Edges in the Minimum Spanning Tree:
2 - 3   Weight: 4
0 - 3   Weight: 5
0 - 1   Weight: 10
Total weight of MST: 19


=== Code Execution Successful ===
```

Result:To implement **Kruskal's Algorithm** in C for finding the **Minimum Spanning Tree (MST)** of a connected, undirected, and weighted graph. Code executed successfully and output is verified