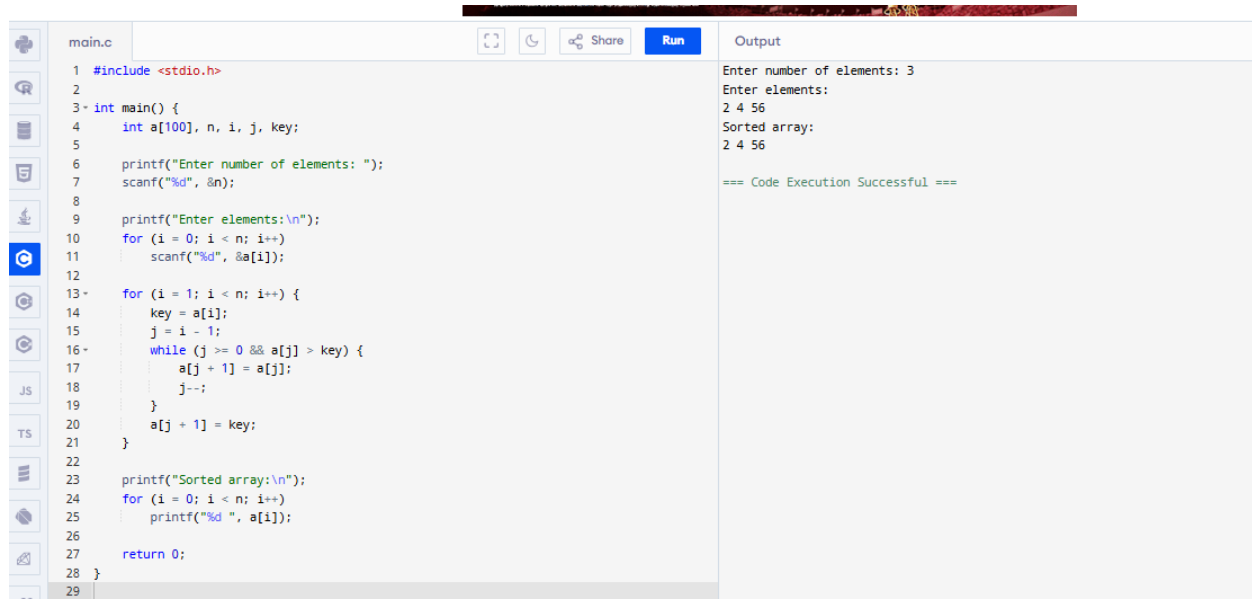


# Lab exercise

16. Write a C program to arrange a series of numbers using Insertion Sort

**Aim:** To sort a list of numbers using the Insertion Sort algorithm.



The screenshot shows a code editor with a C program for Insertion Sort. The code is as follows:

```
1 #include <stdio.h>
2
3 int main() {
4     int a[100], n, i, j, key;
5
6     printf("Enter number of elements: ");
7     scanf("%d", &n);
8
9     printf("Enter elements:\n");
10    for (i = 0; i < n; i++)
11        scanf("%d", &a[i]);
12
13    for (i = 1; i < n; i++) {
14        key = a[i];
15        j = i - 1;
16        while (j >= 0 && a[j] > key) {
17            a[j + 1] = a[j];
18            j--;
19        }
20        a[j + 1] = key;
21    }
22
23    printf("Sorted array:\n");
24    for (i = 0; i < n; i++)
25        printf("%d ", a[i]);
26
27    return 0;
28 }
29
```

The output window on the right shows the following text:

```
Enter number of elements: 3
Enter elements:
2 4 56
Sorted array:
2 4 56

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

## Result:

The program successfully sorted the numbers using Insertion Sort.

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

**Aim:**

To sort an array using Merge Sort.

```

main.c
1 #include <stdio.h>
2
3 void merge(int a[], int l, int m, int r) {
4     int i, j, k;
5     int n1 = m - l + 1;
6     int n2 = r - m;
7     int L[n1], R[n2];
8
9     for (i = 0; i < n1; i++)
10         L[i] = a[l + i];
11     for (j = 0; j < n2; j++)
12         R[j] = a[m + 1 + j];
13
14     i = 0; j = 0; k = l;
15     while (i < n1 && j < n2)
16         a[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
17     while (i < n1) a[k++] = L[i++];
18     while (j < n2) a[k++] = R[j++];
19 }
20
21 void mergeSort(int a[], int l, int r) {
22     if (l < r) {
23         int m = l + (r - l) / 2;
24         mergeSort(a, l, m);
25         mergeSort(a, m + 1, r);
26         merge(a, l, m, r);
27     }
28 }
29
30 int main() {

```

Output

```

Enter number of elements: 5
Enter elements:
23 45 76 43 21
Sorted array:
21 23 43 45 76

=== Code Execution Successful ===

```

```

27     }
28 }
29
30 int main() {
31     int a[100], n, i;
32
33     printf("Enter number of elements: ");
34     scanf("%d", &n);
35
36     printf("Enter elements:\n");
37     for (i = 0; i < n; i++)
38         scanf("%d", &a[i]);
39
40     mergeSort(a, 0, n - 1);
41
42     printf("Sorted array:\n");
43     for (i = 0; i < n; i++)
44         printf("%d ", a[i]);
45
46     return 0;
47 }

```

## Result:

Successfully sorted using Merge Sort.

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

**Aim:** To sort an array using the Quick Sort algorithm.

```
main.c
1 #include <stdio.h>
2
3 void swap(int* a, int* b) {
4     int t = *a;
5     *a = *b;
6     *b = t;
7 }
8
9 int partition(int a[], int low, int high) {
10     int pivot = a[high];
11     int i = low - 1;
12
13     for (int j = low; j < high; j++) {
14         if (a[j] <= pivot) {
15             i++;
16             swap(&a[i], &a[j]);
17         }
18     }
19     swap(&a[i + 1], &a[high]);
20     return i + 1;
21 }
22
23 void quickSort(int a[], int low, int high) {
24     if (low < high) {
25         int pi = partition(a, low, high);
26         quickSort(a, low, pi - 1);
27         quickSort(a, pi + 1, high);
28     }
29 }
30
```

Output

```
Enter number of elements: 5
Enter elements:
23 76 98 33 65
Sorted array:
23 33 65 76 98

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

```
29 }
30
31 int main() {
32     int a[100], n, i;
33
34     printf("Enter number of elements: ");
35     scanf("%d", &n);
36
37     printf("Enter elements:\n");
38     for (i = 0; i < n; i++)
39         scanf("%d", &a[i]);
40
41     quickSort(a, 0, n - 1);
42
43     printf("Sorted array:\n");
44     for (i = 0; i < n; i++)
45         printf("%d ", a[i]);
46
47     return 0;
48 }
```

**Result:** Program correctly sorts elements using Quick Sort.

19. Write a C program to implement Heap sort

**Aim:**

To sort an array using Heap Sort

```
main.c
1 #include <stdio.h>
2
3 void heapify(int a[], int n, int i) {
4     int largest = i;
5     int l = 2 * i + 1;
6     int r = 2 * i + 2;
7
8     if (l < n && a[l] > a[largest])
9         largest = l;
10
11    if (r < n && a[r] > a[largest])
12        largest = r;
13
14    if (largest != i) {
15        int temp = a[i];
16        a[i] = a[largest];
17        a[largest] = temp;
18        heapify(a, n, largest);
19    }
20 }
21
22 void heapSort(int a[], int n) {
23     for (int i = n / 2 - 1; i >= 0; i--)
24         heapify(a, n, i);
25
26     for (int i = n - 1; i > 0; i--) {
27         int temp = a[0];
28         a[0] = a[i];
29         a[i] = temp;
30         heapify(a, i, 0);
31     }
32 }
```

Output

```
Enter number of elements: 5
Enter elements:
34 87 65 34 22
Sorted array:
22 34 34 65 87

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

```
30     heapify(a, i, 0);
31 }
32 }
33
34 int main() {
35     int a[100], n, i;
36
37     printf("Enter number of elements: ");
38     scanf("%d", &n);
39
40     printf("Enter elements:\n");
41     for (i = 0; i < n; i++)
42         scanf("%d", &a[i]);
43
44     heapSort(a, n);
45
46     printf("Sorted array:\n");
47     for (i = 0; i < n; i++)
48         printf("%d ", a[i]);
49
50     return 0;
51 }
```

**Result:** Heap Sort implemented and executed successfully.

20. Write a program to perform the following operations:

- Insert an element into a AVL tree
- Delete an element from a AVL tree
- Search for a key element in a AVL tree

**Aim:** To write a C program to implement an AVL Tree with insertion, deletion, and search operations while maintaining balance.

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Node {
5     int key;
6     struct Node* left;
7     struct Node* right;
8     int height;
9 };
10
11 int max(int a, int b) {
12     return (a > b) ? a : b;
13 }
14
15 int height(struct Node* N) {
16     if (N == NULL)
17         return 0;
18     return N->height;
19 }
20
21 struct Node* newNode(int key) {
22     struct Node* node = (struct Node*)malloc(sizeof(struct Node));
23     node->key = key;
24     node->left = node->right = NULL;
25     node->height = 1;
26     return node;
27 }
28
29 // Right rotate
30 struct Node* rightRotate(struct Node* y) {
31     struct Node* x = y->left;
32     struct Node* T2 = x->right;
33
34     x->right = y;
35     y->left = T2;
```

Output

```
1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 1
Enter key to insert: 23

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 1
Enter key to insert: 43

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 23

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 3
Enter key to search: 23
Key 23 found in the AVL Tree.

1. Insert
2. Delete
```

```
main.c
36
37 y->height = max(height(y->left), height(y->right)) + 1;
38 x->height = max(height(x->left), height(x->right)) + 1;
39
40 return x;
41 }
42
43 // Left rotate
44 struct Node* leftRotate(struct Node* x) {
45     struct Node* y = x->right;
46     struct Node* T2 = y->left;
47
48     y->left = x;
49     x->right = T2;
50
51     x->height = max(height(x->left), height(x->right)) + 1;
52     y->height = max(height(y->left), height(y->right)) + 1;
53
54     return y;
55 }
56
57 // Get balance factor
58 int getBalance(struct Node* N) {
59     if (N == NULL)
60         return 0;
61     return height(N->left) - height(N->right);
62 }
63
64 // Insert a key into AVL tree
65 struct Node* insert(struct Node* node, int key) {
66     if (node == NULL)
67         return newNode(key);
68
69     if (key < node->key)
70         node->left = insert(node->left, key);
```

Output

```
1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 1
Enter key to insert: 23

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 1
Enter key to insert: 43

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 23

1. Insert
2. Delete
3. Search
4. Inorder Display
5. Exit
Enter your choice: 3
Enter key to search: 23
Key 23 found in the AVL Tree.

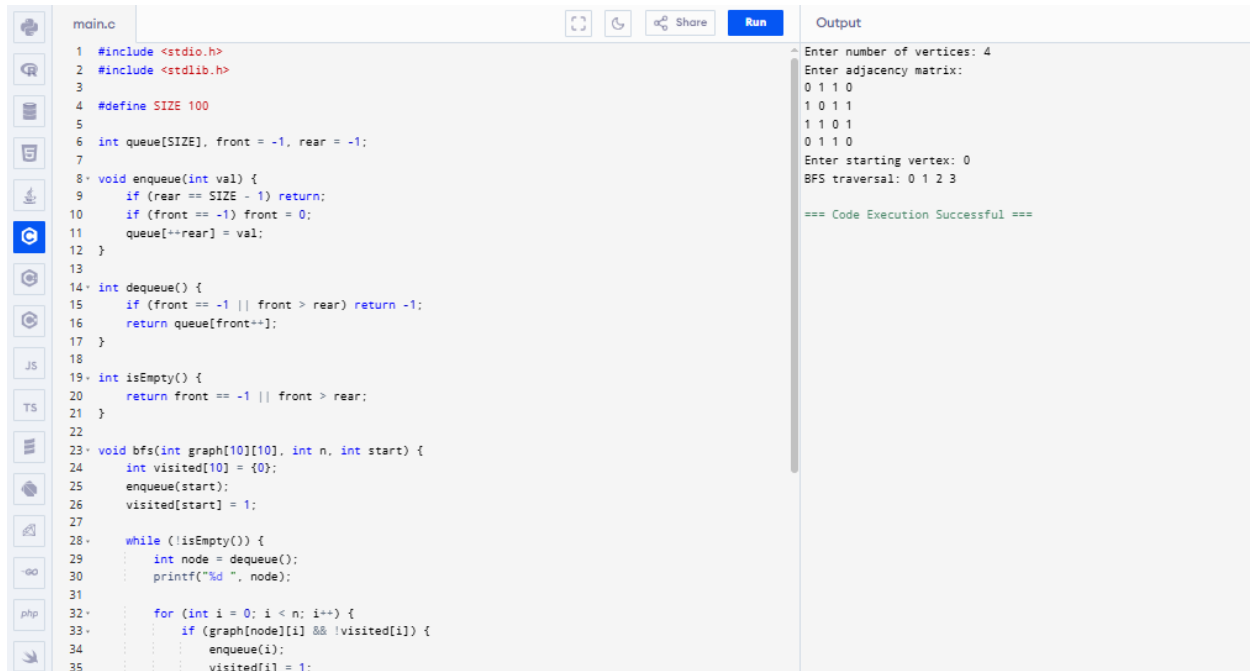
1. Insert
2. Delete
```



**Result:** The AVL Tree operations of insertion, deletion, and search were performed successfully with height balancing maintained.

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

**Aim:** To implement Breadth First Search (BFS) traversal on a graph using adjacency matrix.

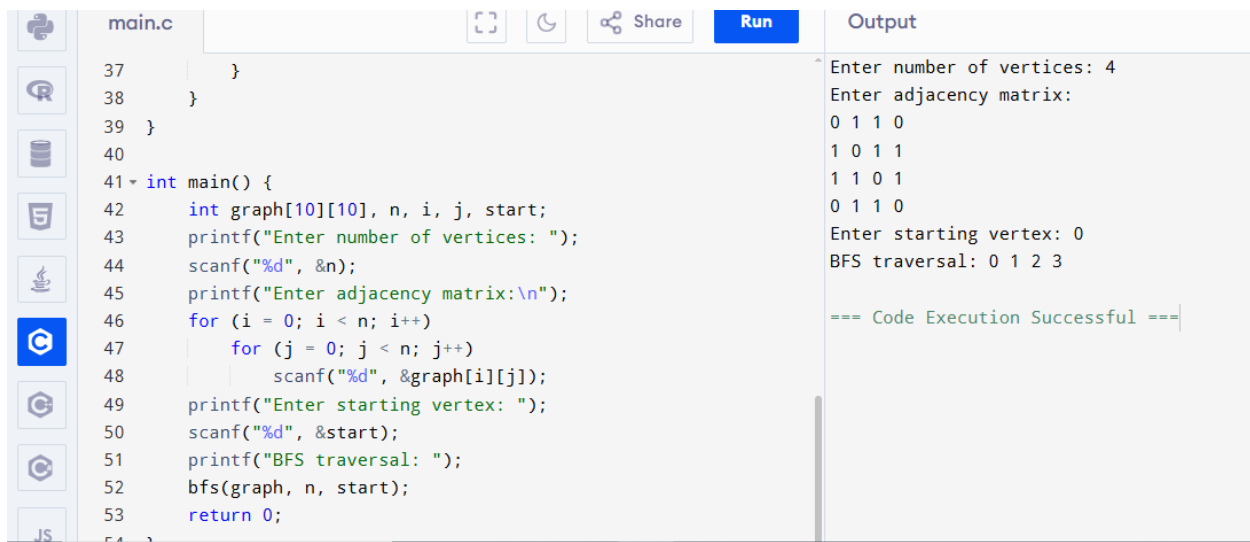


```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define SIZE 100
5
6 int queue[SIZE], front = -1, rear = -1;
7
8 void enqueue(int val) {
9     if (rear == SIZE - 1) return;
10    if (front == -1) front = 0;
11    queue[++rear] = val;
12 }
13
14 int dequeue() {
15     if (front == -1 || front > rear) return -1;
16     return queue[front++];
17 }
18
19 int isEmpty() {
20     return front == -1 || front > rear;
21 }
22
23 void bfs(int graph[10][10], int n, int start) {
24     int visited[10] = {0};
25     enqueue(start);
26     visited[start] = 1;
27
28     while (!isEmpty()) {
29         int node = dequeue();
30         printf("%d ", node);
31
32         for (int i = 0; i < n; i++) {
33             if (graph[node][i] && !visited[i]) {
34                 enqueue(i);
35                 visited[i] = 1;
36             }
37         }
38     }
39 }
```

Output

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter starting vertex: 0
BFS traversal: 0 1 2 3

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



```
main.c
37     }
38 }
39 }
40
41 int main() {
42     int graph[10][10], n, i, j, start;
43     printf("Enter number of vertices: ");
44     scanf("%d", &n);
45     printf("Enter adjacency matrix:\n");
46     for (i = 0; i < n; i++)
47         for (j = 0; j < n; j++)
48             scanf("%d", &graph[i][j]);
49     printf("Enter starting vertex: ");
50     scanf("%d", &start);
51     printf("BFS traversal: ");
52     bfs(graph, n, start);
53     return 0;
54 }
```

Output

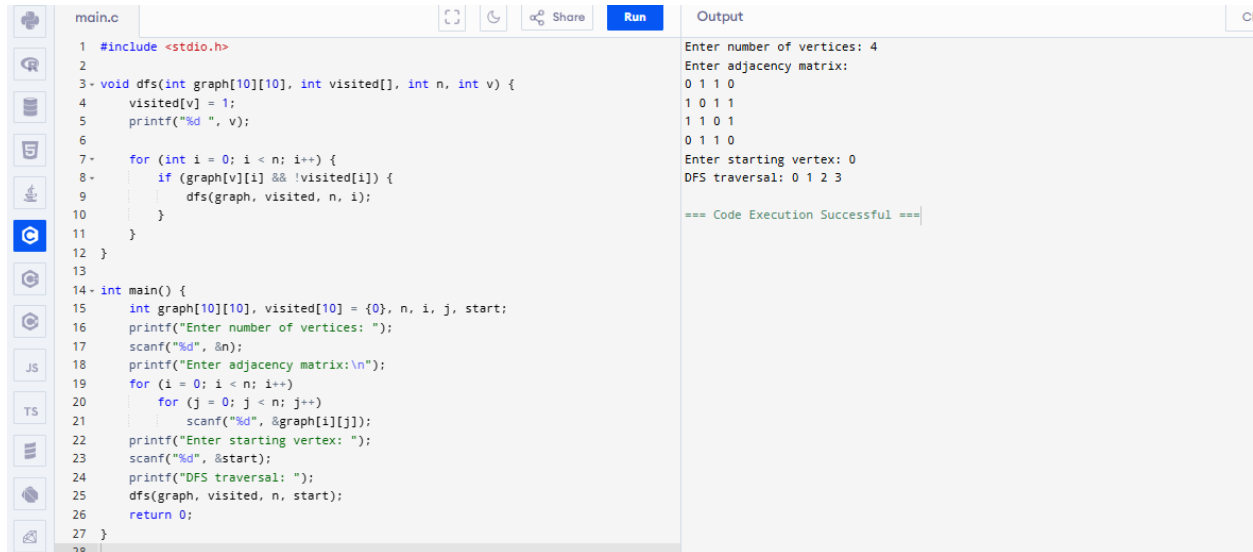
```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter starting vertex: 0
BFS traversal: 0 1 2 3

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

**Result:** Program successfully performs BFS traversal from a given start vertex.

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

**Aim:** To implement Depth First Search (DFS) traversal on a graph using recursion.



The screenshot shows a C program in a code editor with a 'Run' button and an 'Output' window. The code implements a Depth First Search (DFS) algorithm on a graph. It starts by including `<stdio.h>`. The `dfs` function is recursive, taking a graph, a visited array, the number of vertices, and the current vertex. It marks the current vertex as visited and prints its value. Then, it iterates through all vertices, and if a vertex is not visited and there is an edge between them, it recursively calls `dfs` on that vertex. The `main` function prompts the user for the number of vertices, the adjacency matrix, and the starting vertex, then calls `dfs` and prints the traversal result.

```
1 #include <stdio.h>
2
3 void dfs(int graph[10][10], int visited[], int n, int v) {
4     visited[v] = 1;
5     printf("%d ", v);
6
7     for (int i = 0; i < n; i++) {
8         if (graph[v][i] && !visited[i]) {
9             dfs(graph, visited, n, i);
10        }
11    }
12 }
13
14 int main() {
15     int graph[10][10], visited[10] = {0}, n, i, j, start;
16     printf("Enter number of vertices: ");
17     scanf("%d", &n);
18     printf("Enter adjacency matrix:\n");
19     for (i = 0; i < n; i++)
20         for (j = 0; j < n; j++)
21             scanf("%d", &graph[i][j]);
22     printf("Enter starting vertex: ");
23     scanf("%d", &start);
24     printf("DFS traversal: ");
25     dfs(graph, visited, n, start);
26     return 0;
27 }
```

Output:

```
Enter number of vertices: 4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter starting vertex: 0
DFS traversal: 0 1 2 3

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

**Result:** DFS traversal correctly visits all connected vertices from a starting node.

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

**Aim:** To find the shortest path from a source node to all other nodes using Dijkstra's algorithm.



The screenshot shows the Programiz Online Compiler interface. The code is a C program implementing Dijkstra's algorithm. The output window shows the following text:

```
Enter number of vertices: 5
Enter adjacency matrix (999 for no edge):
0 10 999 30 100
10 0 50 999 999
999 50 0 20 10
30 999 20 0 60
100 999 10 60 0
Enter starting vertex: 0
Shortest distances from vertex 0:
To 0: 0
To 1: 10
To 2: 50
To 3: 30
To 4: 60
=== Code Execution Successful ===
```

```
29 }
30
31 int main() {
32     int graph[10][10], n, i, j, start;
33     printf("Enter number of vertices: ");
34     scanf("%d", &n);
35     printf("Enter adjacency matrix (999 for no edge):\n");
36     for (i = 0; i < n; i++)
37         for (j = 0; j < n; j++)
38             scanf("%d", &graph[i][j]);
39     printf("Enter starting vertex: ");
40     scanf("%d", &start);
41     dijkstra(graph, n, start);
42     return 0;
43 }
```

**Result:** The program computes the minimum distance from the start vertex to all other vertices correctly.

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

**Aim:** To find the Minimum Spanning Tree of a graph using Prim's algorithm.

```

main.c
1 #include <stdio.h>
2
3 int main() {
4     int graph[10][10], visited[10] = {0}, n, i, j, edges = 0;
5     int min, x, y, totalCost = 0;
6
7     printf("Enter number of vertices: ");
8     scanf("%d", &n);
9     printf("Enter cost adjacency matrix (999 for no edge):\n");
10    for (i = 0; i < n; i++)
11        for (j = 0; j < n; j++)
12            scanf("%d", &graph[i][j]);
13
14    visited[0] = 1;
15
16    while (edges < n - 1) {
17        min = INF;
18        for (i = 0; i < n; i++) {
19            if (visited[i]) {
20                for (j = 0; j < n; j++) {
21                    if (!visited[j] && graph[i][j] < min) {
22                        min = graph[i][j];
23                        x = i;
24                        y = j;
25                    }
26                }
27            }
28        }
29
30        printf("Edge %d: (%d -> %d) cost = %d\n", edges + 1, x, y, min);
31        totalCost += min;
32        visited[y] = 1;
33        edges++;
34    }
35
36    printf("Total cost of MST: %d\n", totalCost);
37}

```

Output

```

Enter number of vertices: 5
Enter cost adjacency matrix (999 for no edge):
0 2 999 6 999
2 0 3 8 5
999 3 0 999 7
6 8 999 0 9
999 5 7 9 0
Edge 1: (0 -> 1) cost = 2
Edge 2: (1 -> 2) cost = 3
Edge 3: (1 -> 4) cost = 5
Edge 4: (0 -> 3) cost = 6
Total cost of MST: 16

=== Code Execution Successful ===

```

**Result:** The MST is generated with minimum cost using Prim's algorithm.

## 25. Implementation of Minimum Spanning Tree using Kruskal Algorithm.

**Aim:** To construct a Minimum Spanning Tree using Kruskal's algorithm.

```

main.c
1 #include <stdio.h>
2
3 int parent[10];
4
5 int find(int i) {
6     while (parent[i])
7         i = parent[i];
8     return i;
9 }
10
11 int uni(int i, int j) {
12     if (i != j) {
13         parent[j] = i;
14         return 1;
15     }
16     return 0;
17 }
18
19 int main() {
20     int cost[10][10], n, i, j, edges = 1, min, a, b, u, v, total = 0;
21
22     printf("Enter number of vertices: ");
23     scanf("%d", &n);
24     printf("Enter cost adjacency matrix (999 for no edge):\n");
25     for (i = 0; i < n; i++)
26         for (j = 0; j < n; j++)
27             scanf("%d", &cost[i][j]);
28
29     while (edges < n) {
30         min = 999;
31         for (i = 0; i < n; i++)
32             for (j = i + 1; j < n; j++)
33                 if (cost[i][j] < min) {
34                     min = cost[i][j];
35                     a = i;
36                     b = j;
37                 }
38
39         u = find(a);
40         v = find(b);
41         if (u != v) {
42             uni(a, b);
43             edges++;
44             total += min;
45         }
46     }
47
48     printf("Total cost of MST: %d\n", total);
49 }

```

Output

```

Enter number of vertices: 5
Enter cost adjacency matrix (999 for no edge):
0 2 999 6 999
2 0 3 8 5
999 3 0 999 7
6 8 999 0 9
999 5 7 9 0
Edge 1: (0 -> 1) cost = 2
Edge 2: (1 -> 2) cost = 3
Edge 3: (1 -> 4) cost = 5
Edge 4: (0 -> 3) cost = 6
Total cost of MST: 16

=== Code Execution Successful ===

```

```
main.c
42 printf("Enter number of vertices: ");
23 scanf("%d", &n);
24 printf("Enter cost adjacency matrix (999 for no edge):\n");
25 for (i = 0; i < n; i++)
26     for (j = 0; j < n; j++)
27         scanf("%d", &cost[i][j]);
28
29 while (edges < n) {
30     min = 999;
31     for (i = 0; i < n; i++)
32         for (j = 0; j < n; j++)
33             if (cost[i][j] < min) {
34                 min = cost[i][j];
35                 a = u = i;
36                 b = v = j;
37             }
38
39     u = find(u);
40     v = find(v);
41
42     if (uni(u, v)) {
43         printf("Edge %d: (%d -> %d) cost = %d\n", edges++, a, b, min);
44         total += min;
45     }
46
47     cost[a][b] = cost[b][a] = 999;
48 }
49
50 printf("Total cost of MST: %d\n", total);
51 return 0;
```

Output

```
Enter number of vertices: 5
Enter cost adjacency matrix (999 for no edge):
0 2 999 6 999
2 0 3 8 5
999 3 0 999 7
6 8 999 0 9
999 5 7 9 0
Edge 1: (0 -> 1) cost = 2
Edge 2: (1 -> 2) cost = 3
Edge 3: (1 -> 4) cost = 5
Edge 4: (0 -> 3) cost = 6
Total cost of MST: 16
=== Code Execution Successful ===
```

**Result:** All edges of MST were correctly selected using Kruskal's algorithm, and minimum total cost was computed.