FORM NO-F/TL/021

REV.00DATE20.03.2020

Dr. M.G.R.

EDUCATIONAL AND RESEARCH INSTITUTE

DEEMED TO BE UNIVERSITY

University with Graded Autonomy Status

(An ISO 21001 : 2018 Certified Institution)

Periyar E.V.R. High Road, Maduravoyal, Chennai-95. Tamilnadu, India.

RECORD NOTEBOOK

DESIGN AND ANALYSIS OF ALGORITHMS LAB

(EBCS22L03)

2024-2025(EVEN SEMESTER)

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

NAME	:	**R.SANJU**
REG NO	:	**231061101481**
COURSE	:	**B.TECH CSE**
YEAR/SEM/SEC	:	**II/IV/BE**

FORM NO-F/TL/021

REV.00DATE20.03.2020

Dr. M.G.R.
EDUCATIONAL AND RESEARCH INSTITUTE
DEEMED TO BE UNIVERSITY
University with Graded Autonomy Status
(An ISO 21001 : 2018 Certified Institution)
Periyar E.V.R. High Road, Maduravoyal, Chennai-95. Tamilnadu, India.

BONAFIDE CERTIFICATE

Register No	**:231061101481**
Name of Lab	: **DESIGN AND ANALYSIS OFALGORITHMS LAB (EBCS22L03)**
Department	**COMPUTERSCIENCEANDENGINEERING**

Certified that this is the bonafide record of work done by **R.SANJU 231061101481** of II Year B.Tech (CSE), Sec-‘BE’ in the **DESIGN AND ANALYSIS OF ALGORITHM’S LAB(EBCS22L03)** during the year 2024-2025

Signature of Lab-in-Charge **Signature of Head of Dept**

Submitted for the Practical Examination held on ------------------------

Internal Examiner **External Examiner**

INDEX

Ex.no:1 **Date:**

Quick Sort Algorithm

Aim:

Algorithm:

1. Input the number of elements(`n`).

2. Generate `n` random numbers** and store them in an array.

3. Start the timer to measure execution time.

4. Call Quick Sort function with the array and its bounds (`low = 0`, `high = n-1`).

5. Quick Sort function:

   - If `low < high`, partition the array and get the pivot index (`pi`).

   - Recursively sort the left sub-array (`low to pi-1`).

   - Recursively sort the right sub-array (`pi+1 to high`).

6. Partition function:

   - Select the pivot (last element of sub-array).

   - Rearrange elements so that smaller elements are before the pivot.

   - Swap the pivot to its correct position.

   - Return pivot index.

7. Stop the timer after sorting completes.

8. Display the sorted array.

9. Calculate and display execution time.

Flowchart:

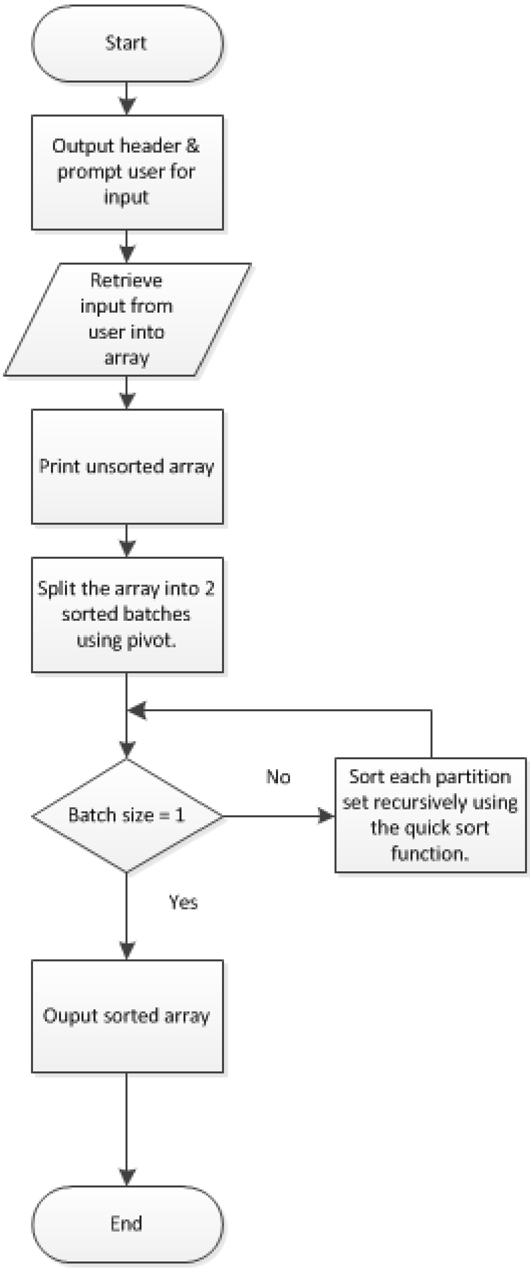

Fig.no 1a

Program:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
Void swap(int&a,int&b) {
    int temp = a;
    a = b;
    b = temp;
}
Int partition(std::vector<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;
}
Void quickSort(std::vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
int main() {
    int n;
    std::cout<<"231061101481 & R.SANJU\n";
    std::cout << "Enter number of integers: ";
    std::cin >> n;
    std::vector<int> arr(n);
    srand(static_cast<unsigned>(time(nullptr)));
    std::cout << "Generated random numbers: ";
    for (int &num : arr) {
        num = rand() % 100;
        std::cout << num << " ";
    }
    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();
    std::cout << "\nSorted array: ";
    for (const int &num : arr) {
        std::cout << num << " ";
    }
    double time_taken = static_cast<double>(end - start) / CLOCKS_PER_SEC;
    std::cout << "\nTime taken: " << time_taken << " seconds\n";
    return 0;
}
```

Program explanation:

1. Generate random numbers and store them in an array.
2. Select a pivot from the array and partition elements into smaller and larger groups.
3. Recursively apply QuickSort on the left and right sub-arrays.
4. Swap elements to ensure correct order.
5. Sort the array completely by continuing partitioning.
6. Display the sorted numbers and execution time.

Output:

```
231061101481 & R.SANJU
Enter number of integers: 5
Generated random numbers: 23 17 55 9 58
Sorted array: 9 17 23 55 58
Time taken: 2e-06 seconds
```

Fig.no:1b

RESULT:

Exp.no:2 **Date:**

Strassen matrix

Aim:

Algorithm:

1.Divide the matrices into four equal submatrices.

1.Compute seven intermediate matrices using recursive multiplication.

3.Calculate submatrix products using scalar additions and subtractions.

4. Recursively compute the seven matrix products for smaller submatrices.

5.Construct the final submatrices using computed values.

6. Merge the submatrices to form the final matrix.

7. Return the result matrix as the product of the original matrices.

Flow chart:

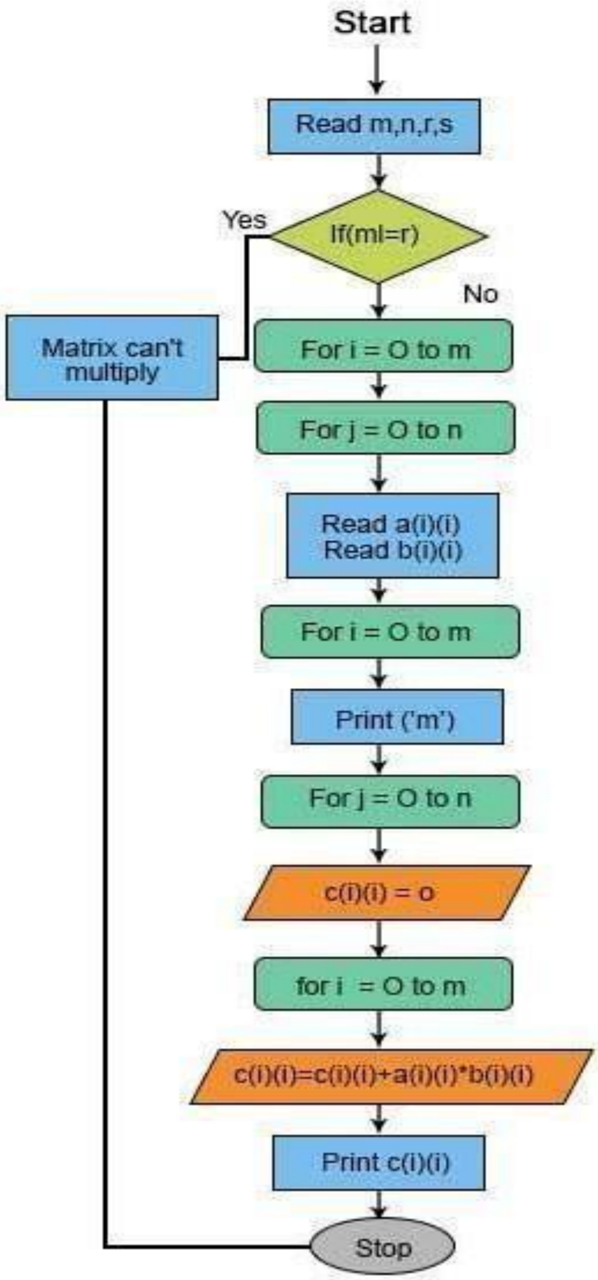

Fig.no:2

Program:

```
#include <iostream>
using namespace std;

void multiplyMatrices(int A[][10], int B[][10], int C[][10], int rowsA, int colsA, int rowsB, int
colsB) {
    if (colsA != rowsB) {
        cout << "Matrix multiplication not possible. Columns of A must equal rows of B." <<
endl;
        retur
    }
for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) {
            C[i][j] = 0;
            for (int k = 0; k < colsA; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
int main() {
    cout<<"231061101481 & R.SANJU\n";
    int rowsA, colsA, rowsB, colsB;
    int A[10][10], B[10][10], C[10][10];
    cout << "Enter number of rows and columns for first matrix: ";
    cin >> rowsA >> colsA;
    cout << "Enter number of rows and columns for second matrix: ";
    cin >> rowsB >> colsB;

    if (colsA = rowsB) {
        cout << "Matrix multiplication not possible. Columns of A must equal rows of B." <<
endl;
        return 1

    }
```

```
    cout << "Enter elements of first matrix:" << endl;
    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsA; j++) {
            cin >> A[i][j];
        }
    }
    cout << "Enter elements of second matrix:" << endl;
    for (int i = 0; i < rowsB; i++) {
        for (int j = 0; j < colsB; j++) {
            cin >> B[i][j];
        }
    }
    multiplyMatrices(A, B, C, rowsA, colsA, rowsB, colsB);
    cout << "Resultant matrix:" << endl;
    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsB; j++) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Program Explanation:

1. Initialize matrices `A` and `B` and store their elements.

2. Create a result matrix `C`, initially filled with zeros.

3. Multiply corresponding elements of matrices using nested loops.

4. Store computed values in matrix `C`.

5. Complete multiplication for all elements.

6. Print the final matrix `C` as output.

Output:

```
231061101481 & R.SANJU
Enter number of rows and columns for first matrix: 2 2
Enter number of rows and columns for second matrix: 2 2
Enter elements of first matrix:
4 5 3 2
Enter elements of second matrix:
7 5 9 4
Resultant matrix:
73 40
39 23
```

Fig.no:2b

RESULT:

Exp.no:3 **Date:**

Warshall Algorithm

Aim:

Algorithm:

1. Initialize the adjacency matrix representing the graph.

2. Set diagonal elements to 1 (each node is reachable from itself).

3. Iterate over all intermediate vertices (k from 1 to n).

4. For each pair of vertices (i, j), update the reachability:

5. If i can reach k and k can reach j, then i can reach j.

6. Repeat for all vertices to compute transitive closure.

7. Return the updated matrix, showing reachability between all pairs.

Flowchart

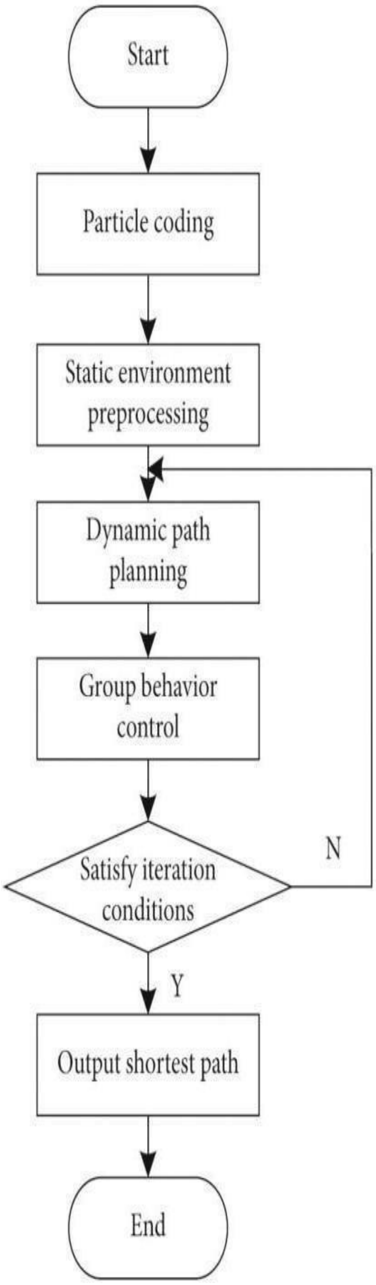

Fig.no:3a

Program:

```
#include <iostream>
using namespace std;
void warshallTransitiveClosure(int graph[][10], int V) {
    int tc[10][10];
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            tc[i][j] = graph[i][j];
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
            }
        }
    }
    cout << "Transitive Closure:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << tc[i][j] << " ";
        }
        cout << endl;
    }
    int main() {
    cout<<"231061101481 & R.SANJU\n";
    int V;
    int graph[10][10];
    cout << "Enter number of vertices (max 10): ";
    cin >> V;
    cout << "Enter adjacency matrix:\n";
    for (int i = 0; i < V; i++)
```

```
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];
    warshallTransitiveClosure(graph, V);
    return 0;
}
```

Program Expanation:

1. Initialize adjacency matrix with connectivity between vertices.

2. Copy initial values into the transitive closure matrix.

3. Iterate over all vertices as possible intermediate nodes.

4. Update reachability between vertex pairs using logical OR.

5. Process all vertices to compute transitive closure.

6. Display the final transitive closure matrix.

Output:

```
231061101481 & R.SANJU
Enter number of vertices (max 10): 2
Enter adjacency matrix:
5
6
1
3
Transitive Closure:
1 1
1 1
```

Fig.no:3b

RESULT:

.

Exp.no:4 **Date:**

Floyd-Warshall algorithm

Aim:

Algorithm:

1. Initialize the adjacency matrix with edge weights; set unreachable paths to infinity.

2. Set diagonal elements to zero (each node's shortest path to itself is zero).

3. Iterate over all intermediate vertices (`k` from `1` to `n`).

4. For each pair of vertices (`i`, `j`), update the shortest path:

   - If `i → k → j` is shorter than `i → j`, update the distance.

5. Repeat for all vertices to compute shortest paths between all pairs.

6. Store the final matrix, which contains the shortest distances between all nodes.

7. Return the matrix as the solution to the shortest path problem.

Flowchart:

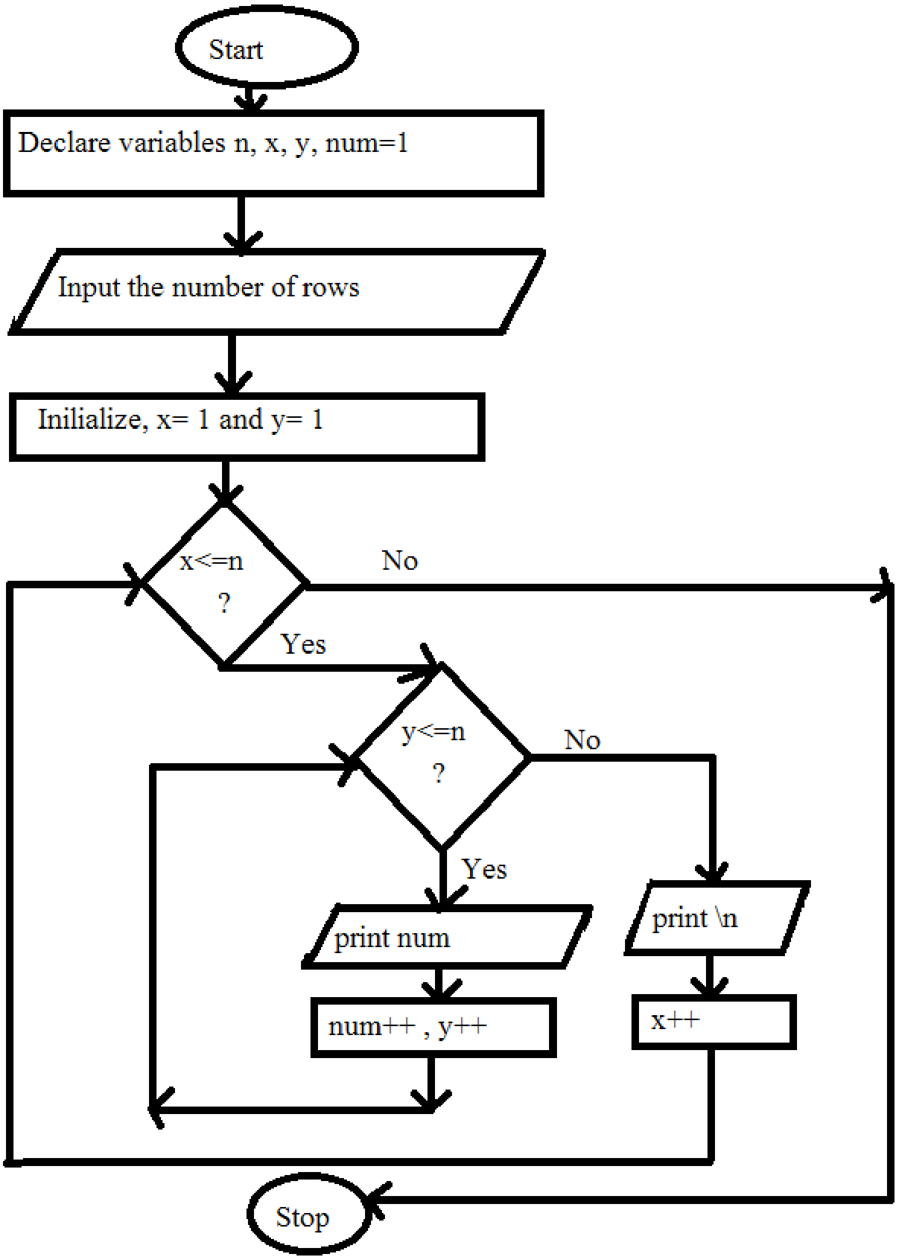

Fig.no:4a

Program:

```
#include <iostream>
#include <vector>
#include <climits>
#define V 4
#define INF INT_MAX
void floydWarshall(std::vector<std::vector<int>>& graph) {
for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (graph[i][k] != INF && graph[k][j] != INF) {
                    graph[i][j] = std::min(graph[i][j], graph[i][k] + graph[k][j]);
                }
            }
        }
    }
}
void printGraph(const std::vector<std::vector<int>>& graph) {
    std::cout << "All-Pairs Shortest Path Matrix:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (graph[i][j] == INF)
                std::cout << "INF ";
            else
                std::cout << graph[i][j] << " ";
        }
        std::cout << '\n';
    }
}
int main() {
    std::vector<std::vector<int>> graph(V, std::vector<int>(V));
    std::cout<<"231061101481 & R.SANJU\n";
    std::cout << "Enter adjacency matrix (" << V << "x" << V << ") for the weighted
```

```
graph:\n";
    std::cout << "(Use " << INF << " for no direct path)\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            std::cin >> graph[i][j];
        }
    }
    floydWarshall(graph);
    printGraph(graph);
    return 0;
```

Program Explanation:

1. Initialize adjacency matrix with edge weights.

2. Set unreachable paths to infinity (`INF`).

3. Consider each node as an intermediate vertex.

4. Update shortest paths using the formula `distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])`.

5. Complete shortest path computation for all vertex pairs.

6. Display the final shortest path matrix.

Output:

```
231061101481 & R.SANJU
Enter adjacency matrix (4x4) for the weighted graph:
(Use 2147483647 for no direct path)
8 3 4 7
7 3 2 4
6 2 3 7
2 1 4 8
All-Pairs Shortest Path Matrix:
8 3 4 7
6 3 2 4
6 2 3 6
2 1 3 5
```

Fig.no:4b

RESULT:

Exp.no:5 **Date:**

Travelling Salesman Problem

Aim:

Algorithm:

1. List all cities and their distances.
2. Generate all possible routes visiting each city exactly once.
3. Calculate the total distance for each route.
4. Select the route with the minimum total distance.
5. Use dynamic programming or branch-and-bound to optimize the search.
6. Store the optimal path and its cost.
7. Return the shortest route as the solution.

Flowchart:

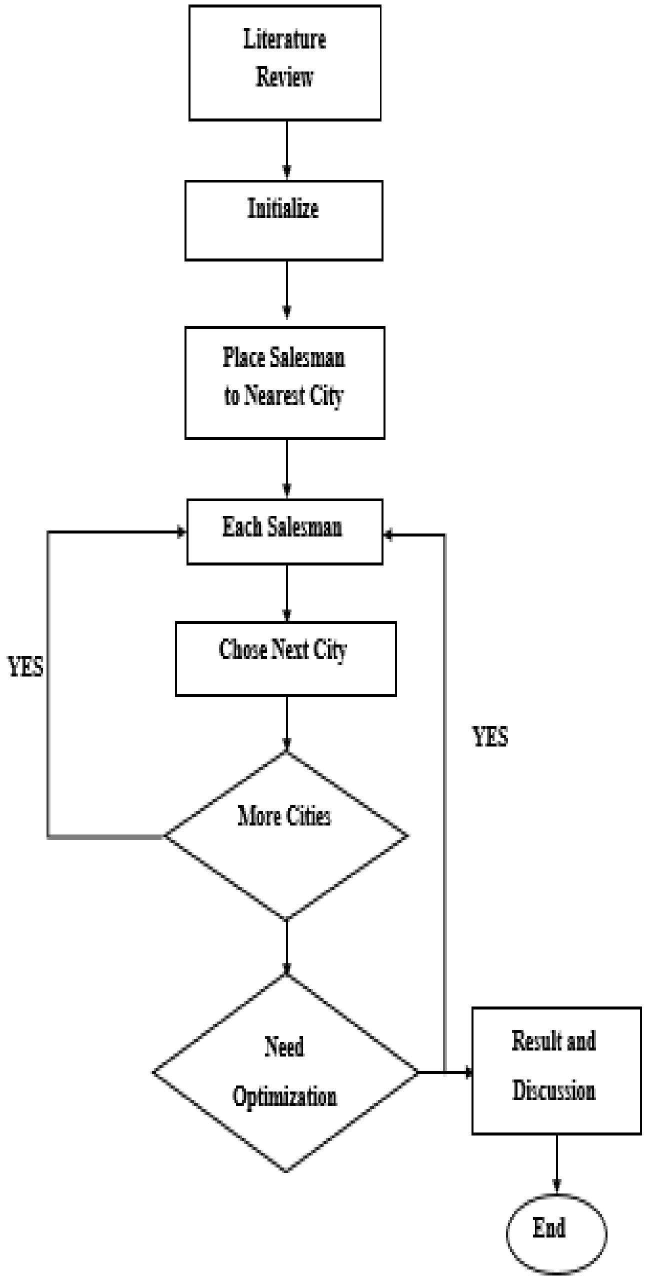

Fig.no:5a

Program:

```
#include <iostream>
#include <climits>

#define V 4
#define INF INT_MAX
int minCost = INF;
int optimalPath[V];
void branchAndBound(int graph[V][V], int path[], int visited[], int level, int cost) {
    if (level == V) {
        if (graph[path[V - 1]][path[0]] == INF) return; // Prevent overflow
        cost += graph[path[V - 1]][path[0]];
        if (cost < minCost) {
            minCost = cost;
            for (int i = 0; i < V; i++)
                optimalPath[i] = path[i];
        }
        return;
    }
    for (int i = 0; i < V; i++) {
        if (!visited[i] && graph[path[level - 1]][i] != INF) {
            visited[i] = 1;
            path[level] = i;
            branchAndBound(graph, path, visited, level + 1, cost + graph[path[level - 1]][i]);
            visited[i] = 0;
        }
    }
}
int nearestNeighbor(int graph[V][V], int start) {
    int cost = 0, current = start;
    bool visited[V] = {false};
    visited[current] = true;
    for (int count = 1; count < V; count++) {
        int nearest = -1, minDist = INF;
```

```
        for (int j = 0; j < V; j++) {
            if (!visited[j] && graph[current][j] < minDist) {
                minDist = graph[current][j];
                nearest = j;
            }
        }
        if (nearest == -1) return INF; // No valid path
        cost += minDist;
        current = nearest;
        visited[current] = true;
    }
    if (graph[current][start] == INF) return INF;
    cost += graph[current][start]; // Return to start
    return cost;
}
int main() {
    int graph[V][V];
    std::cout<<"231061101481 & R.SANJU\n";
    std::cout << "Enter adjacency matrix (" << V << "x" << V << ") for the weighted
graph:\n";
    std::cout << "(Enter -1 if no direct path exists between nodes)\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++) {
            int val;
            std::cin >> val;
            graph[i][j] = (val == -1) ? INF : val;
        }
    int path[V], visited[V] = {0};
    path[0] = 0;
    visited[0] = 1;
    branchAndBound(graph, path, visited, 1, 0);
    int approxCost = nearestNeighbor(graph, 0)
    std::cout << "\nOptimal TSP Cost (Branch & Bound): ";
    if (minCost == INF)
```

```
        std::cout << "No valid tour exists.\n";
    else {
        std::cout << minCost << "\nOptimal Path: ";
        for (int i = 0; i < V; i++)
            std::cout << optimalPath[i] << " ";
        std::cout << optimalPath[0] << "\n"; // Return to start
    }
    std::cout << "Approximate TSP Cost (Nearest Neighbor): ";
    if (approxCost == INF)
        std::cout << "No valid tour exists.\n";
    else {
        std::cout << approxCost << "\n";
        if (minCost != INF) {
            float error = ((float)(approxCost - minCost) / minCost) * 100;
            std::cout << "Approximation Error: " << error << "%\n";
        }
    }

    return 0;
}
```

Program Explanation:

1. Define cities and their distances in a matrix.

2. Use branch-and-bound method to explore possible routes.

3. Calculate cost of each possible route.

4. Track the minimum cost route as the best solution.

5. Use nearest neighbor heuristic for approximation.

6. Display optimal and approximate solutions with error percentage.

Output:

```
231061101481 & R.SANJU
Enter adjacency matrix (4x4) for the weighted graph:
(Enter -1 if no direct path exists between nodes)
9 6 9 8
8 6 7 9
7 8 9 7
9 5 8 9
ERROR!

Optimal TSP Cost (Branch & Bound): 27
Optimal Path: 0 3 1 2 0
Approximate TSP Cost (Nearest Neighbor): 29
Approximation Error: 7.40741%
```

Fig.no:5b

RESULT:

Exp.no:6 **Date:**

Knapsack Problem

Aim:

.

Algorithm:

1. Initialize a table for storing maximum values for different weights.

2. Iterate over items, checking if they fit in the knapsack.

3. For each item, decide whether to include it based on maximum value achievable.

4. Update the table with the best possible value at each weight.

5. Trace back to determine selected items.

6. Store the optimal selection of items.

7. Return the maximum value achievable within the weight limit.

Flowchart:

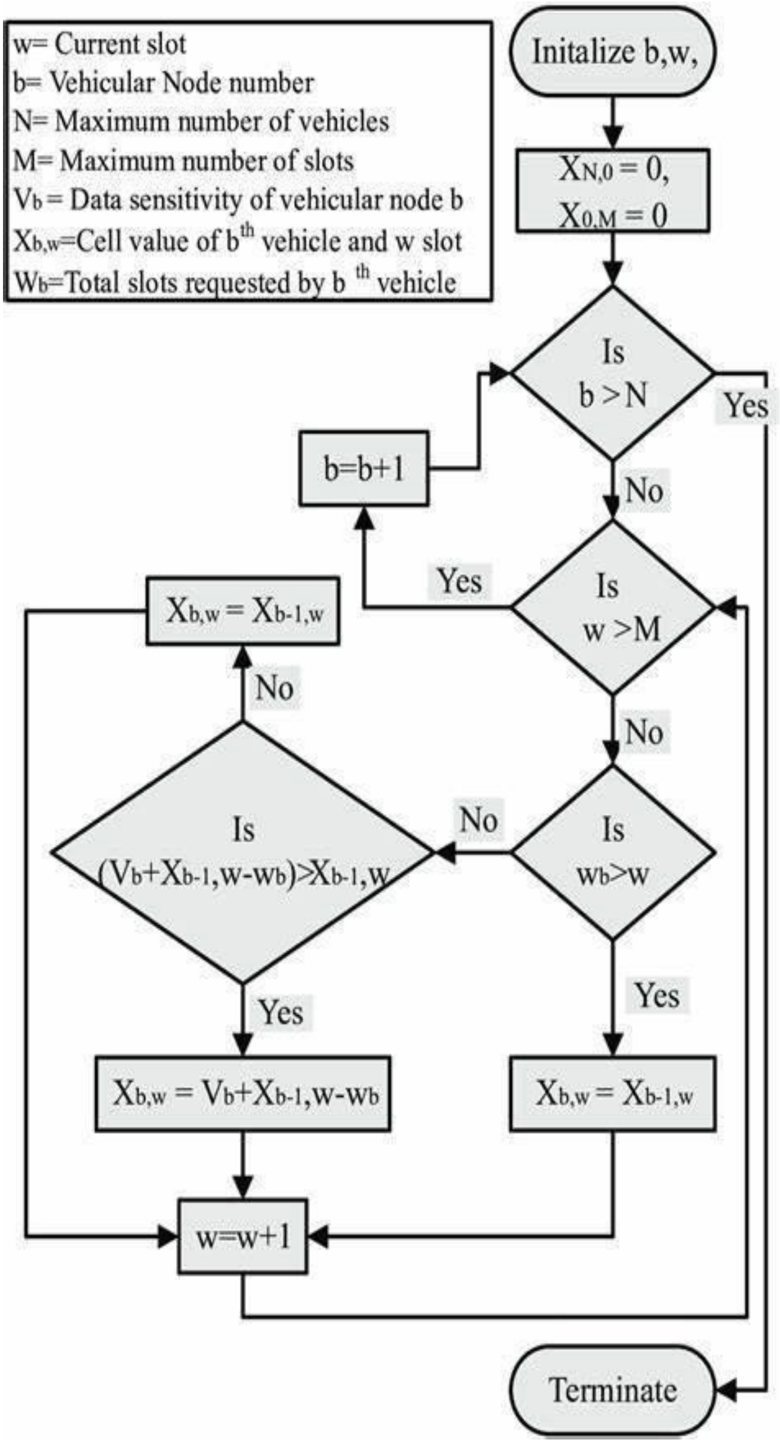

Fig.no:6a

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int knapsack(int W, const vector<int>& wt, const vector<int>& val, int n) {
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            if (wt[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], val[i - 1] + dp[i - 1][w - wt[i - 1]]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][W];
}
int main() {
    int n, W;
    cout<<"231061101481 & R.SANJU\n";
    cout << "Enter the number of items: ";
    cin >> n;
    vector<int> wt(n), val(n);
    cout << "Enter the capacity of the knapsack: ";
    cin >> W;
    cout << "Enter the weights and values of the items:\n";
    for (int i = 0; i < n; ++i) {
        cout << "Item " << i + 1 << " - Weight: ";
        cin >> wt[i];
        cout << "Item " << i + 1 << " - Value: ";
        cin >> val[i];
    }
```

```
    int maxProfit = knapsack(W, wt, val, n);
    cout << "\nMaximum value in Knapsack = " << maxProfit << endl;
    return 0;
}
```

Program Explanation:

1. Initialize a table for storing computed values.

2. Check each item for inclusion based on weight capacity.

3. Choose items that maximize value without exceeding weight limit.

4. Update values recursively to find optimal solutions.

5. Complete table computation for maximum profit.

6. Display the highest achievable value in knapsack.

Output:

```
231061101481 & R.SANJU
Enter the number of items: 3
Enter the capacity of the knapsack: 50
Enter the weights and values of the items:
Item 1 - Weight: 26
Item 1 - Value: 10
Item 2 - Weight: 40
Item 2 - Value: 35
Item 3 - Weight: 68
Item 3 - Value: 59

Maximum value in Knapsack = 35
```

Fig.no:6b

RESULT:

Exp.no:7 **Date:**

Dijkstra's Algorithm

Aim:

.**Algorithm:**

1. Initialize distance from the source to all vertices as infinity, except the source itself.

2. Set the source vertex as visited.

3. For each unvisited neighbor, update its shortest distance if a shorter path is found.

4. Select the next vertex with the smallest known distance.

5. Repeat until all vertices are visited and shortest paths are determined.

6. Store the shortest paths from the source to all vertices.

7. Return the shortest path distances for all nodes.

Flowchart:

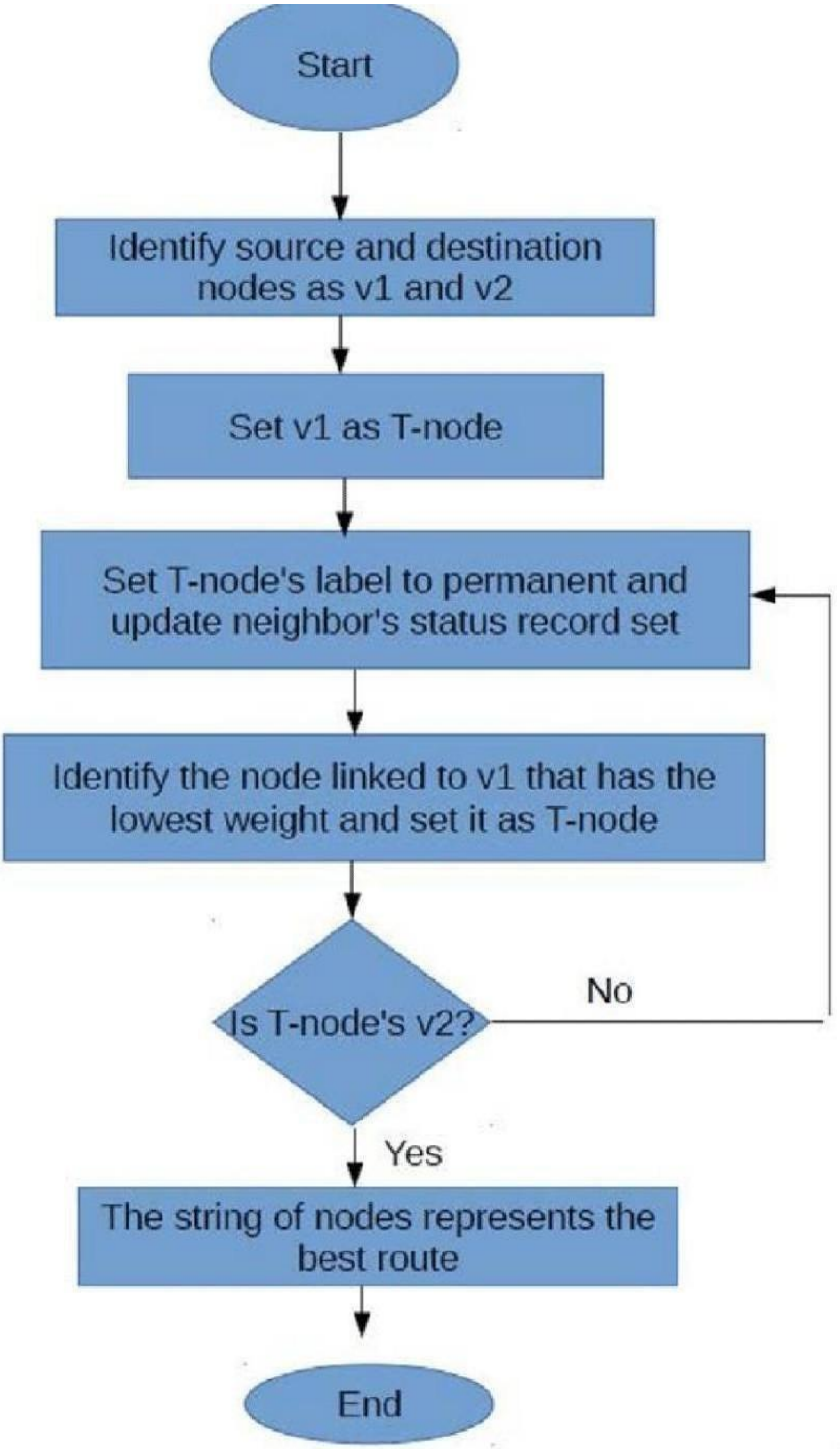

Fig.no:7a

Program:

```
#include <iostream>
#include <climits>
#include <vector>
using namespace std;
void dijkstra(const vector<vector<int>>& graph, int vertices, int start) {
    const int INF = INT_MAX;
    vector<int>  distance(vertices,  INF);
    vector<bool> visited(vertices, false);
    distance[start] = 0;
    for (int count = 0; count < vertices - 1; ++count) {
        int minDistance = INF, minIndex = -1;

        for (int i = 0; i < vertices; ++i) {
            if (!visited[i] && distance[i] < minDistance) {
                minDistance = distance[i];
                minIndex = i;
            }
        }
        if (minIndex == -1) break;  // All remaining vertices are unreachable
        visited[minIndex] = true;
        for (int j = 0; j < vertices; ++j) {
            if (!visited[j] &&
                graph[minIndex][j] != 0 &&
                distance[minIndex] != INF &&
                distance[minIndex] + graph[minIndex][j] < distance[j]) {
                distance[j] = distance[minIndex] + graph[minIndex][j];
            }
        }
    }
    cout << "\nVertex\tDistance from Source\n";
    for (int i = 0; i < vertices; ++i) {
        cout << i << "\t";
```

```
        if (distance[i] == INF)
            cout << "INF";
        else
            cout << distance[i];
        cout << "\n";
    }
}
int main() {
    cout<<"231061101481 & R.SANJU\n";
    int vertices, start;
    cout << "Enter number of vertices: ";
    cin >> vertices;
    vector<vector<int>> graph(vertices, vector<int>(vertices));

    cout << "Enter adjacency matrix (use 0 for no direct path):\n";
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < vertices; ++j) {
            cin >> graph[i][j];
        }
    }
    cout << "Enter starting vertex (0 to " << vertices - 1 << "): ";
    cin >> start;
    if (start < 0 || start >= vertices) {
        cout << "Invalid starting vertex.\n";
        return 1;
    }
    dijkstra(graph, vertices, start);
    return 0;
}
```

Program Explanation:

1. Initialize distances from source as infinite (`INF`).

2. Set source distance to zero and mark it as visited.

3. Select the nearest vertex and update distances of neighbors.

4. Repeat for all vertices to find shortest paths.

5. Complete path computation ensuring optimal distances.

6. Display shortest distances from source to all nodes.

Output:

```
231061101481 & R.SANJU
Enter number of vertices: 3
Enter adjacency matrix (use 0 for no direct path):
0 5 3
4 2 0
2 6 0
Enter starting vertex (0 to 2): 2

Vertex  Distance from Source
0   2
1   6
2   0
```

Fig.no:7b

RESULT:

Exp.no:8 **Date:**

Kruskal's Algorithm

Aim:

Algorithm:

1. Sort all edges by increasing weight.

2. Initialize a forest with each vertex as a separate tree.

3. Iterate through edges, adding them if they don’t form a cycle.

4. Use Union-Find to check for cycles.

5. Continue adding edges until all vertices are connected.

6. Store the minimum spanning tree with selected edges.

7. Return the minimum spanning tree as the solution.

Flowchart:

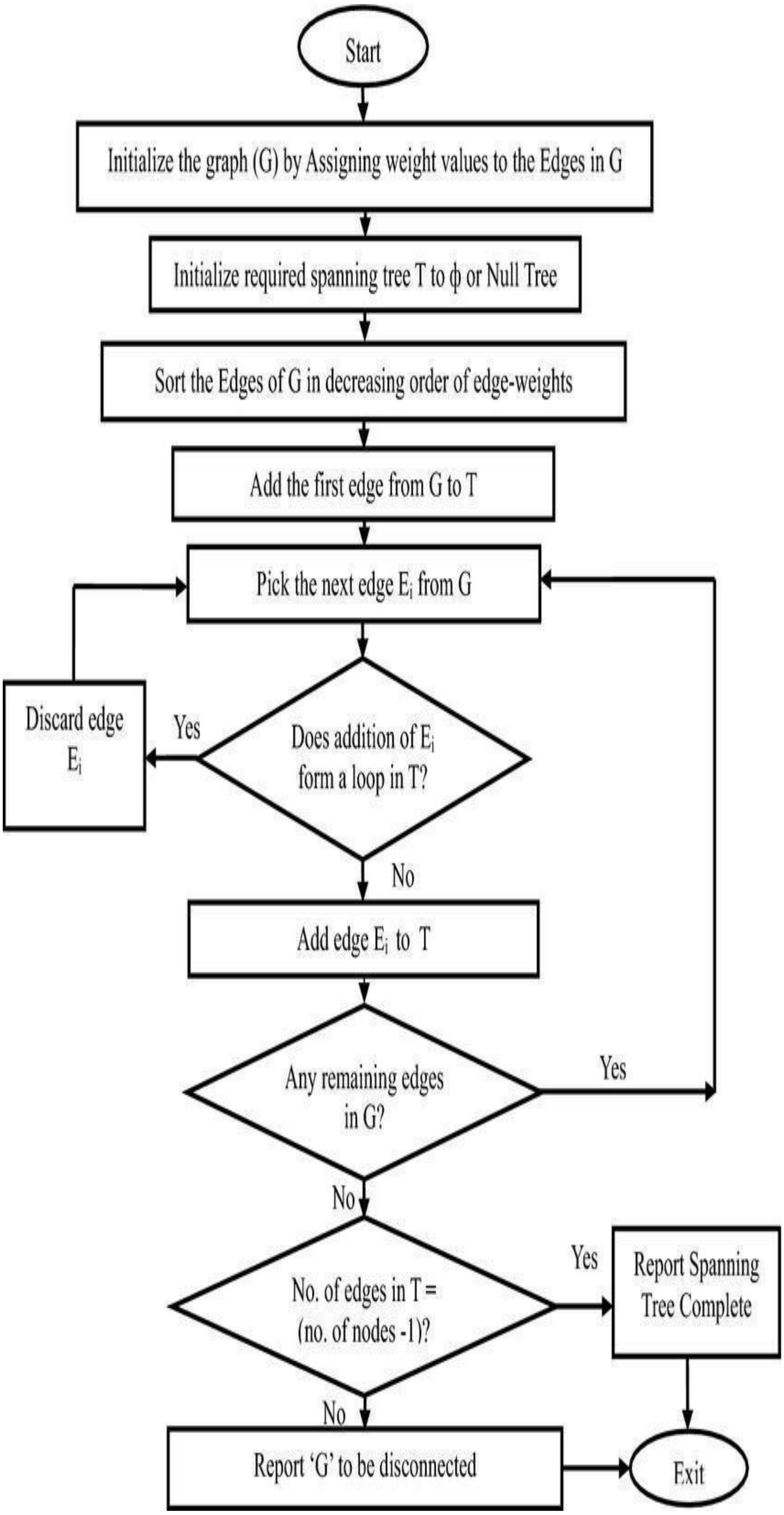

Fig.no:8a

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
#define MAX_VERTICES 10
struct Edge {
    int src, dest, weight;
};
class DisjointSet {
    std::vector<int> parent, rank;
public:
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }
    int find(int u) {
        if (parent[u] != u)
            parent[u] = find(parent[u]); // Path compression
        return parent[u];
    }
    void unite(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);
        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV])
                parent[rootV] = rootU;
            else if (rank[rootU] < rank[rootV])
                parent[rootU] = rootV;
            else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
```

```
        }
    }
};

void kruskalMST(std::vector<Edge>& edges, int V) {
    DisjointSet ds(V);
    int mstWeight = 0;
    std::vector<Edge> mst;
    std::sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;});

    for (const auto& edge : edges) {
        int rootSrc = ds.find(edge.src);
        int rootDest = ds.find(edge.dest);
        if (rootSrc != rootDest) {
            mst.push_back(edge);
            mstWeight += edge.weight;
            ds.unite(rootSrc, rootDest);
        }
    }
    std::cout << "\nEdges in the Minimum Spanning Tree:\n";
    for (const auto& edge : mst)
        std::cout << edge.src << " - " << edge.dest << " : " << edge.weight << "\n";
    std::cout << "Total MST Weight: " << mstWeight << "\n";
}

int main() {
    int V, E;
    std::cout<<"231061101481 & R.SANJU\n";
    std::cout << "Enter number of vertices: ";
    std::cin >> V;
    std::cout << "Enter number of edges: ";
    std::cin >> E;
    std::vector<Edge> edges(E);
```

```
    std::cout << "Enter edges (src dest weight):\n";
    for (int i = 0; i < E; i++)
        std::cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    kruskalMST(edges, V);
    return 0;
}
```

Program Explanation:

1. Sort edges based on increasing weight.

2. Initialize parent sets for all vertices.

3. Select edges one by one, ensuring no cycles.

4. Use Union-Find to check connectivity.

5. Continue adding edges until all vertices are connected.

6. Display minimum spanning tree edges and total cost.

Output:

```
231061101481 & R.SANJU
Enter number of vertices: 2
Enter number of edges: 2
Enter edges (src dest weight):
4
7
3
9
1
2

Edges in the Minimum Spanning Tree:
9 - 1 : 2
Total MST Weight: 2
```

Fig.no:8b

RESULT:

Exp.no:9 **Date:**

N-Queens Problem

Aim:

Algorithm:

1. Place the first queen in the first row.
2. Check for conflicts (same column, diagonal).
3. If a conflict occurs, backtrack and try a different position.
4. Repeat until all queens are placed or backtracking finds no solution.
5. Store the valid board configurations.
6. Return all possible solutions for placing `N` queens safely.

Flowchart:

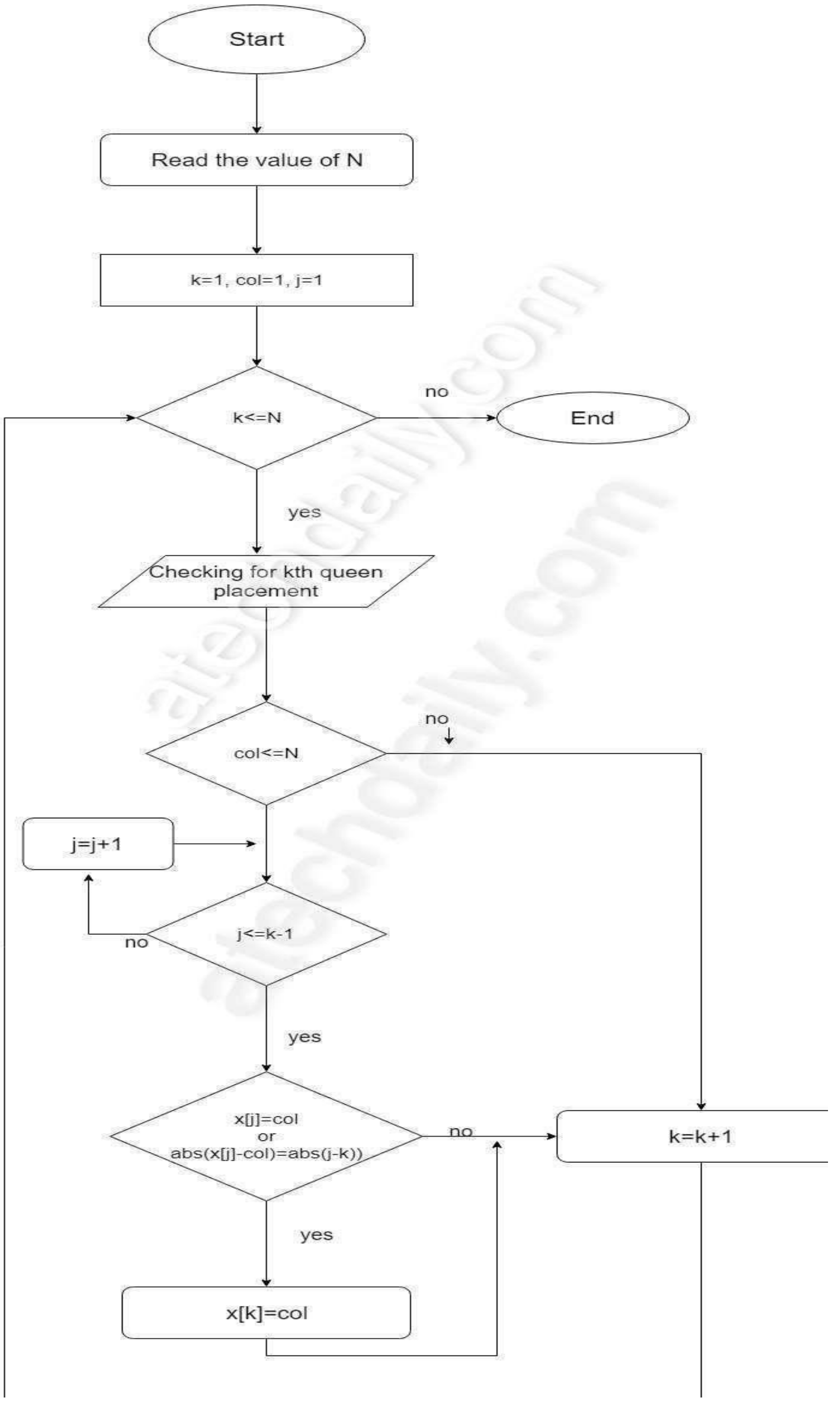

Fig.no:9a

Program:

```
#include <iostream>

#include <vector>

using namespace std;
int N;

vector<vector<int>> board;
bool isSafe(int row, int col)
{

    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;
for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
  if (board[i][j])

            return false;

    for (int i = row, j = col; i < N && j >= 0; i++, j--)
        if (board[i][j])
            return false;
    return true;

}
ool solveNQueens(int col) {
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(i, col)) {
            board[i][col] = 1;
if (solveNQueens(col + 1)

    return true;
```

```
        board[i][col] = 0; }

    }
    return false;       }
void printSolution() {

    cout << "Solution:\n";

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << (board[i][j] ? "Q " : "- ");
        cout << endl;
    } }
int main() {
     cout<<"231061101481 & R.SANJU\n";

    cout << "Enter the value of N: ";
    cin >> N;
    if (N <= 0) {

        cout << "N must be a positive integer.\n";
        return 1;  }
    board.assign(N, vector<int>(N, 0));
    if (solveNQueens(0))
        printSolution();
    else

        cout << "No solution exists\n";
    return 0;

}
```

Program Explanation:

1. Initialize an empty board for `N` queens.

2. Place first queen in the first row.

3. Move to the next row checking safe positions.

4. Backtrack if conflicts arise and try different placements.

5. Repeat until all queens are placed correctly.
   Print all valid board configura

Output:

```
231061101481 & R.SANJU
Enter the value of N: 4
Solution:
- - Q -
Q - - -
- - - Q
- Q - -
```

Fig.no:9b

RESULT: