NRI INSTITUTE OF INFORMATION SCIENCE

& TECHNOLOGY BHOPAL

DEPARTMENT OF

INFORMATION TECHNOLOGY

LAB MANUAL

ANALYSIS AND DESIGN OF ALGORITHM

(IT-403)

BACHELOR OF ENGINEERING (C.B.G.S)

NIIST BHOPAL		NRI INSTITUTE OF INFORMATION SCIENCE & TECHNOLOGY DEPT NAME: Information technology	FORM NO	NIIST/A/10
BRANCH	IT	LIST OF EXPERIMENT	REV. NO	0
SEMESTER	IV		REV. DT	30/06/2011
SUBJECT/CODE :- ANALYSIS AND DESIGN OF ALGORITHM / IT 403				

S. NO.	LIST OF EXPERIMENT
1	Write a program for Iterative and Recursive Binary Search.
2	Write a program for Merge Sort.
3	Write a program for Quick Sort.
4	Write a program for Strassen's Matrix Multiplication.
5	Write a program for optimal merge patterns.
6	Write a program for Huffman coding.
7	Write a program for minimum spanning trees using Kruskal's algorithm.
8	Write a program for minimum spanning trees using Prim's algorithm.
9	Write a program for single sources shortest path algorithm.
10	Write a program for Floyd-Warshall algorithm.
11	Write a program for traveling salesman problem.
12	Write a program for Hamiltonian cycle problem.

Subject Name: ANALYSIS AND DESIGN OF ALGORITHM

Subject Code: IT-403

Course Outcomes:

After successful completion of course, students will be able to:

CO1: Implement and analyze sorting and searching algorithms such as binary search, merge sort, quick sort, and heap sort.

CO2: Apply divide-and-conquer, greedy, and dynamic programming techniques to solve optimization problems including knapsack, shortest paths, and matrix multiplication.

CO3: Design and evaluate algorithms using backtracking and branch & bound for problems like 8-Queens, Hamiltonian cycle, and traveling salesman.

CO4: Utilize advanced tree and graph structures (BST, AVL, 2-3 trees, B-trees) for efficient searching and traversal operations.

CO5: Differentiate between NP-hard and NP-complete problems and understand their implications in algorithmic design.

CO6: Demonstrate the ability to apply parallel algorithms and appreciate their role in improving computational efficiency.

CO7: Translate theoretical algorithmic concepts into practical programming experiments and projects for real-world applications.

EXPERIMENT NO. 1

AIM:

To implement and compare **iterative and recursive binary search algorithms** for locating an element in a sorted array.

INTRODUCTION:

Binary search is a classic divide-and-conquer algorithm used to efficiently find the position of a target element in a sorted array. By repeatedly dividing the search interval in half, binary search achieves logarithmic time complexity, making it significantly faster than linear search for large datasets. The algorithm can be implemented both iteratively and recursively, each with its own trade-offs in terms of space and clarity.

Algorithm (Iterative):

```
BINARY_SEARCH_ITERATIVE(A, n, key)
1. Initialize low = 0, high = n - 1
2. Repeat while low <= high:
   a. mid = (low + high) / 2
   b. If A[mid] == key:
        return mid
   c. Else if A[mid] < key:
        low = mid + 1
   d. Else:
        high = mid - 1
3. If loop ends, return -1 (not found)
```

Algorithm (Recursive):

```
BINARY_SEARCH_RECURSIVE(A, low, high, key)
1. If low > high:
      return -1
2. mid = (low + high) / 2
3. If A[mid] == key:
      return mid
4. Else if A[mid] < key:
      return BINARY_SEARCH_RECURSIVE(A, mid+1, high, key)
5. Else:
      return BINARY_SEARCH_RECURSIVE(A, low, mid-1, key)
```

SOURCE CODE:

SAMPLE OUTPUT:

```
Enter number of elements: 5
Enter sorted elements:
1 2 3 4 5
Enter key to search: 4
Iterative result: Found at index 3
Recursive result: Found at index 3
```

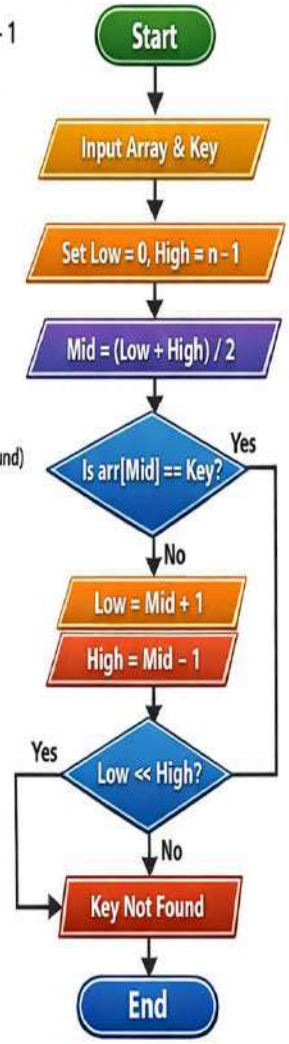
Flowchart :

Iterative Binary Search Algorithm

Algorithm (iterative):

BINARY_SEARCH_ITERATIVE(A, n, key)

1. Initialize low = 0, high = n – 1
2. Repeat while low <= high:
   a. mid = (low + high) / 2
   b. If A[mid] == key:
      return mid
   c. Else if A[mid] < key:
      low = mid + 1
   d. Else:
      high = mid – 1
3. If loop ends, return -1 (not found)

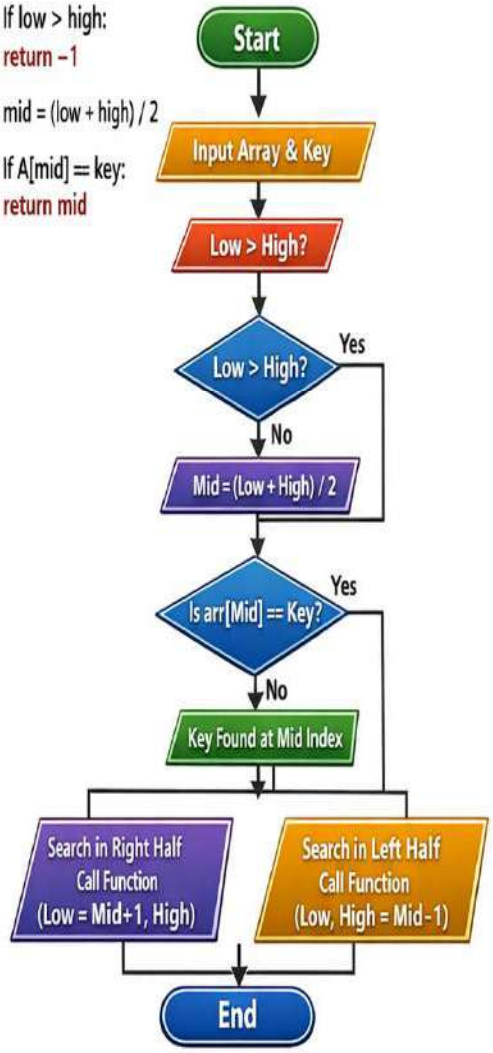


Recursive Binary Search Algorithm

Algorithm (Recursive):

BINARY_SEARCH_RECURSIVE(A, low, high, key)

1. If low > high:
   return –1
2. mid = (low + high) / 2
3. If A[mid] == key:
   return mid



EXPERIMENT NO. 2

AIM:

To implement the merge sort algorithm for sorting an array of integers.

INTRODUCTION:

Merge sort is a stable, comparison-based sorting algorithm that employs the divide-and-conquer paradigm. It recursively divides the array into halves, sorts each half, and then merges the sorted halves to produce the final sorted array. Merge sort guarantees O(n log n) time complexity in all cases and is widely used for external sorting and large datasets.

ALGORITHM :

```
MERGE_SORT(A, l, r)
1. If l < r:
   a. Find middle m = (l + r) / 2
   b. Call MERGE_SORT(A, l, m)
   c. Call MERGE_SORT(A, m+1, r)
   d. Call MERGE(A, l, m, r)

MERGE(A, l, m, r)

1. Create arrays L and R for left and right halves
2. Copy elements into L and R
3. Initialize i = 0, j = 0, k = l
4. While i < size(L) and j < size(R):
   a. If L[i] <= R[j]:
        A[k] = L[i], i++
   b. Else:
        A[k] = R[j], j++
   c. k++
5. Copy remaining elements of L if any
6. Copy remaining elements of R if any
```

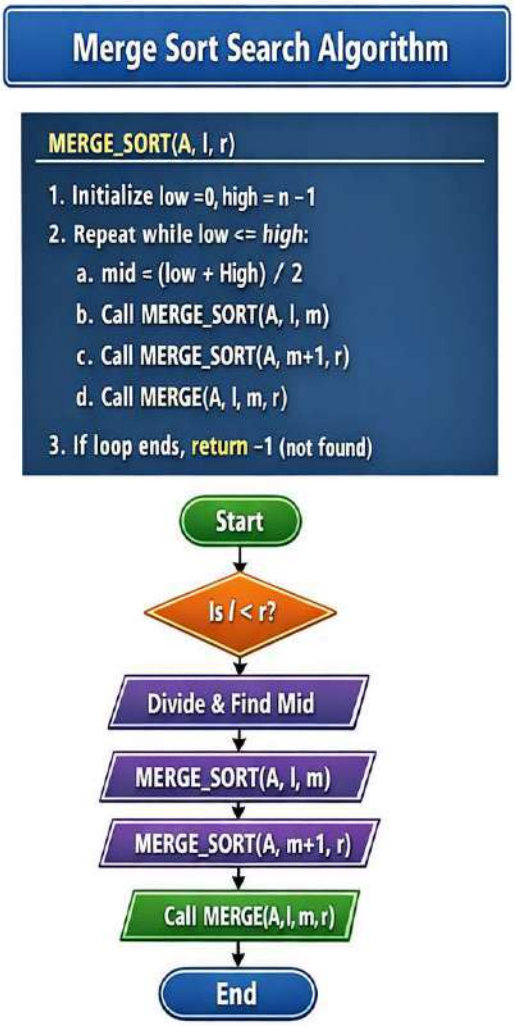
SOURCE CODE:

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

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for(int i=0;i<n1;i++) L[i]=arr[l+i];
    for(int j=0;j<n2;j++) R[j]=arr[m+1+j];
    int i=0,j=0,k=l;
    while(i<n1 && j<n2) {
        if(L[i]<=R[j]) arr[k++]=L[i++];
        else arr[k++]=R[j++];
    }
    while(i<n1) arr[k++]=L[i++];
    while(j<n2) arr[k++]=R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if(l<r) {
        int m=(l+r)/2;
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r);
    }
}

int main() {
    int n; cin>>n;
    int arr[n];
    for(int i=0;i<n;i++) cin>>arr[i];
    mergeSort(arr,0,n-1);
    for(int i=0;i<n;i++) cout<<arr[i]<<" ";
}
```

SAMPLE OUTPUT:

```
Input: 5
5 2 4 1 3
Output: 1 2 3 4 5
```

FLOWCHART:


Merge Sort Search Algorithm
MERGE_SORT(A, l, r)
1. Initialize low =0, high = n -1
2. Repeat while low <= high:
a. mid = (low + High) / 2
b. Call MERGE_SORT(A, l, m)
c. Call MERGE_SORT(A, m+1, r)
d. Call MERGE(A, l, m, r)
3. If loop ends, return -1 (not found)
Start
Is l < r?
Divide & Find Mid
MERGE_SORT(A, l, m)
MERGE_SORT(A, m+1, r)
Call MERGE(A,l,m,r)
End


Merge Procedure
MERGE(A, l, m, r)
1. If low > high:
return -1
2. mid = (low + High) / 2
3. If A[mid] == key:
return mid:
4. Else if A[mid] < key:
return BINARY_SEARCH RECURSIVE(A, mid-1, Hgh, key)
5. Else:
return BINARY_SEARCH RECURSIVE(A, low, mid-1, key)
Start
Create Arrays L & R
Copy Elements into L & R
Initialize l = 0; j ≈ 0, k = l
i < size(L) and j < size(R)?
L[i] < R[i]: Ak] = L[i], i++
A[k = R[], j++
k = k + 1
End

EXPERIMENT NO.3

AIM:

To implement the quick sort algorithm for sorting an array of integers

INTRODUCTION:

Quick sort is an efficient, in-place, divide-and-conquer sorting algorithm. It selects a pivot element, partitions the array into elements less than and greater than the pivot, and recursively sorts the subarrays. Quick sort has an average-case time complexity of O(n log n), but its worst-case is $O(n^2)$, which can be mitigated by good pivot selection strategies.

ALGORITHM:

```
QUICK_SORT(A, low, high)
1. If low < high:
   a. pi = PARTITION(A, low, high)
   b. QUICK_SORT(A, low, pi-1)
   c. QUICK_SORT(A, pi+1, high)


PARTITION(A, low, high)
1. pivot = A[high]
2. i = low - 1
3. For j = low to high-1:
   a. If A[j] <= pivot:
        i++
        swap A[i] and A[j]
4. swap A[i+1] and A[high]
5. return i+1
```

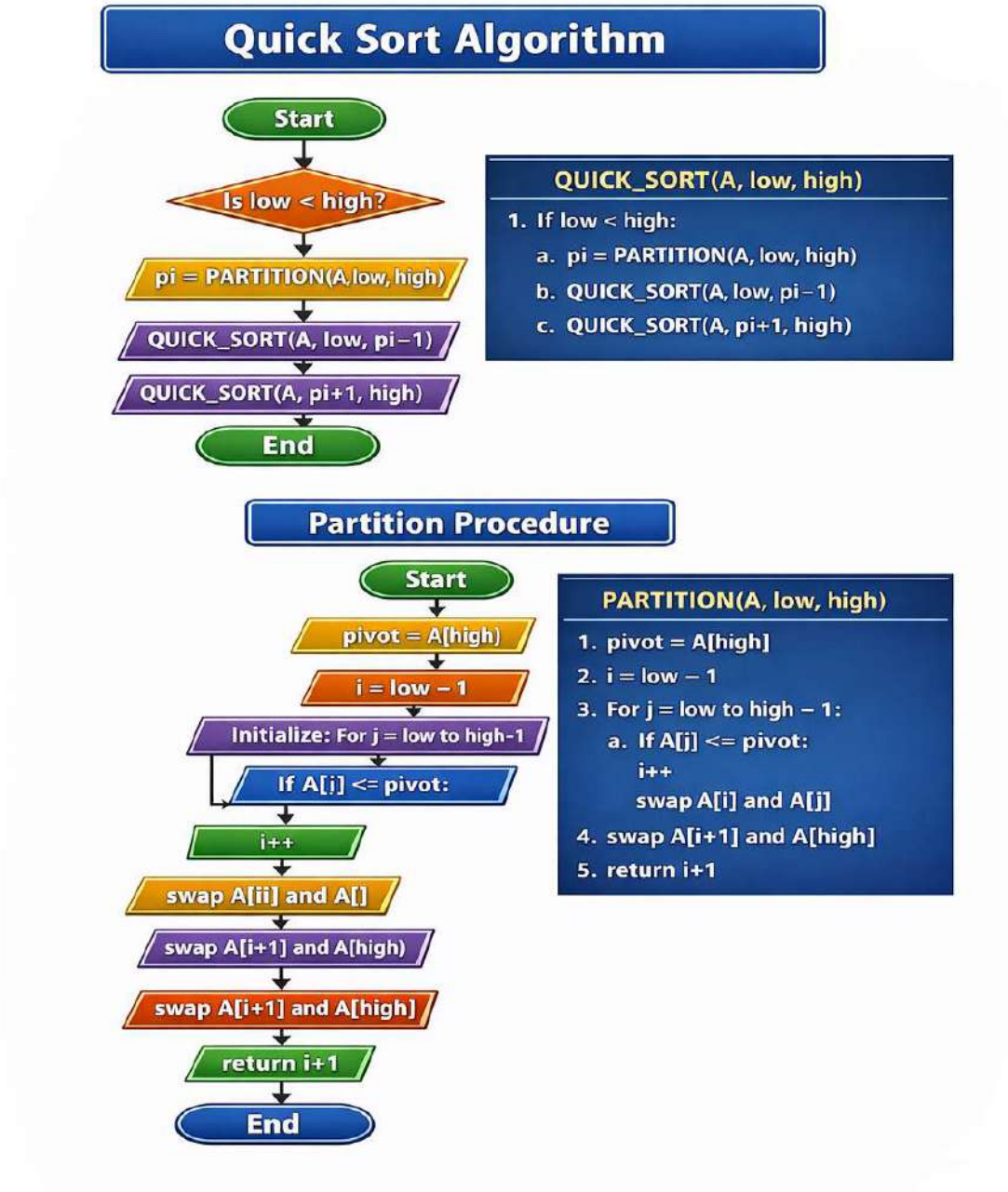
SOURCE CODE:

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

int partition(int arr[], int low, int high) {
    int pivot=arr[high], 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(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; cin>>n;
    int arr[n];
    for(int i=0;i<n;i++) cin>>arr[i];
    quickSort(arr,0,n-1);
    for(int i=0;i<n;i++) cout<<arr[i]<<" ";
}
```

SAMPLE OUTPUT:

```
Input: 6
10 7 8 9 1 5
Output: 1 5 7 8 9 10
```

FLOWCHART:


Quick Sort Algorithm
Start
Is low < high?
pi = PARTITION(A,low, high)
QUICK_SORT(A, low, pi−1)
QUICK_SORT(A, pi+1, high)
End
QUICK_SORT(A, low, high)
1. If low < high:
a. pi = PARTITION(A, low, high)
b. QUICK_SORT(A, low, pi−1)
c. QUICK_SORT(A, pi+1, high)
Partition Procedure
Start
pivot = A[high]
i = low − 1
Initialize: For j = low to high-1
If A[j] <= pivot:
i++
swap A[ii] and A[]
swap A[i+1] and A[high]
swap A[i+1] and A[high]
return i+1
End
PARTITION(A, low, high)
1. pivot = A[high]
2. i = low − 1
3. For j = low to high − 1:
a. If A[j] <= pivot:
i++
swap A[i] and A[j]
4. swap A[i+1] and A[high]
5. return i+1

EXPERIMENT NO.4

AIM:

To implement Strassen's algorithm for matrix multiplication.

INTRODUCTION:

Strassen's algorithm is a divide-and-conquer method for multiplying two matrices, reducing the number of multiplications required compared to the standard algorithm. For n × n matrices, it achieves a time complexity of approximately O(n^2.81), making it faster for large matrices. The algorithm recursively divides matrices into submatrices and combines results using seven multiplications and several additions/subtractions.

ALGORITHM:

```
STRASSEN_MULTIPLY(A, B)
1. Compute:
   M1 = (A11 + A22) * (B11 + B22)
   M2 = (A21 + A22) * B11
   M3 = A11 * (B12 - B22)
   M4 = A22 * (B21 - B11)
   M5 = (A11 + A12) * B22
   M6 = (A21 - A11) * (B11 + B12)
   M7 = (A12 - A22) * (B21 + B22)
2. Compute result matrix:
   C11 = M1 + M4 - M5 + M7
   C12 = M3 + M5
   C21 = M2 + M4
   C22 = M1 - M2 + M3 + M6
3. Return matrix C
```

SOURCE CODE:

```cpp
#include <iostream>
#include <vector>
using namespace std;
// Helper function to add two matrices
```

```
vector<vector<int>> add(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}
// Helper function to subtract two matrices
vector<vector<int>> subtract(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] - B[i][j];
    return C;
}
// Strassen's algorithm for matrix multiplication
vector<vector<int>> strassen(const vector<vector<int>>& A, const
vector<vector<int>>& B) {
    int n = A.size();
    if (n == 1) {
        return {{A[0][0] * B[0][0]}};
    }
    int k = n / 2;
    vector<vector<int>> A11(k, vector<int>(k)), A12(k, vector<int>(k)),
A21(k, vector<int>(k)), A22(k, vector<int>(k));
    vector<vector<int>> B11(k, vector<int>(k)), B12(k, vector<int>(k)),
B21(k, vector<int>(k)), B22(k, vector<int>(k));
    for (int i = 0; i < k; ++i)
```

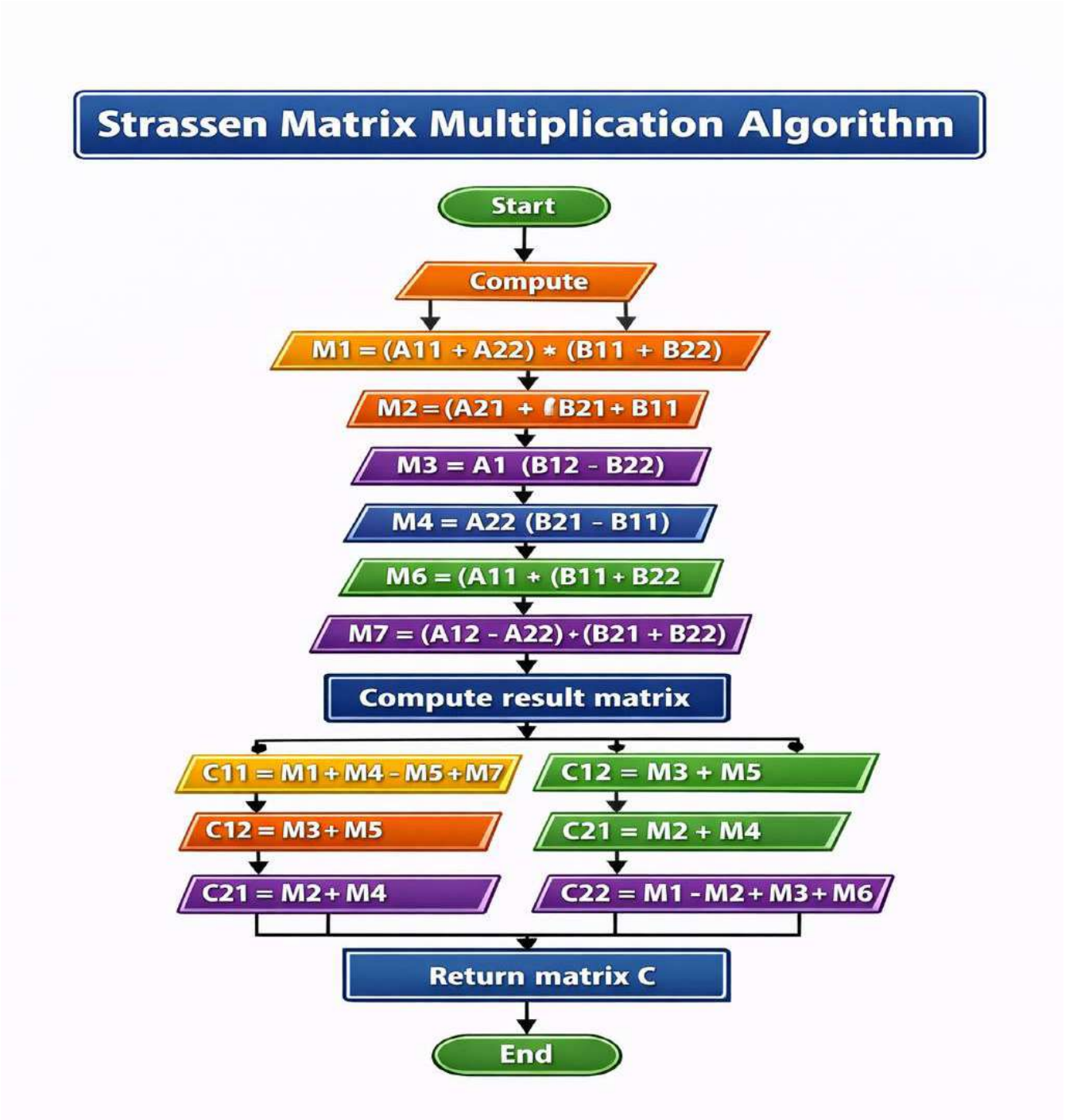
```
        for (int j = 0; j < k; ++j) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }
    auto M1 = strassen(add(A11, A22), add(B11, B22));
    auto M2 = strassen(add(A21, A22), B11);
    auto M3 = strassen(A11, subtract(B12, B22));
    auto M4 = strassen(A22, subtract(B21, B11));
    auto M5 = strassen(add(A11, A12), B22);
    auto M6 = strassen(subtract(A21, A11), add(B11, B12));
    auto M7 = strassen(subtract(A12, A22), add(B21, B22));
    vector<vector<int>> C(n, vector<int>(n));
    for (int i = 0; i < k; ++i)
        for (int j = 0; j < k; ++j) {
            C[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
            C[i][j + k] = M3[i][j] + M5[i][j];
            C[i + k][j] = M2[i][j] + M4[i][j];
            C[i + k][j + k] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
        }
    return C;
}

int main() {
    int n = 2;
    vector<vector<int>> A = {{1, 2}, {3, 4}};
    vector<vector<int>> B = {{5, 6}, {7, 8}};
```

```
    auto C = strassen(A, B);
    cout << "Product matrix:\n";
    for (const auto& row : C) {
        for (int val : row) cout << val << " ";
        cout << endl;
    }
    return 0;
}
```

SAMPLE OUTPUT:

```
Enter the 4 elements of first matrix: 1 2
3
4
Enter the 4 elements of second matrix: 5 6
7
8
The first matrix is

1    2
3    4
The second matrix is

5    6
7    8
After multiplication
19   22
43   50
```

FLOWCHART:


Strassen Matrix Multiplication Algorithm
Start
Compute
M1 = (A11 + A22) * (B11 + B22)
M2 = (A21 + B21 + B11
M3 = A1 (B12 - B22)
M4 = A22 (B21 - B11)
M6 = (A11 + (B11 + B22
M7 = (A12 - A22) + (B21 + B22)
Compute result matrix
C11 = M1 + M4 - M5 + M7
C12 = M3 + M5
C12 = M3 + M5
C21 = M2 + M4
C21 = M2 + M4
C22 = M1 - M2 + M3 + M6
Return matrix C
End

EXPERIMENT NO. 5

AIM:

Write a program for optimal merge patterns.

INTRODUCTION:

The optimal merge pattern problem seeks the most efficient way to merge multiple sorted files into a single file, minimizing the total computation cost. The greedy solution involves repeatedly merging the two smallest files, a process efficiently implemented using a min-heap (priority queue). This approach is widely used in external sorting and file management systems.

ALGORITHM:

```
OPTIMAL_MERGE(files)
1. Insert all file sizes into a min-heap
2. total_cost = 0
3. While heap size > 1:
   a. Extract two smallest sizes f1, f2
   b. cost = f1 + f2
   c. total_cost += cost
   d. Insert cost back into heap
4. Return total_cost
```

SOURCE CODE:

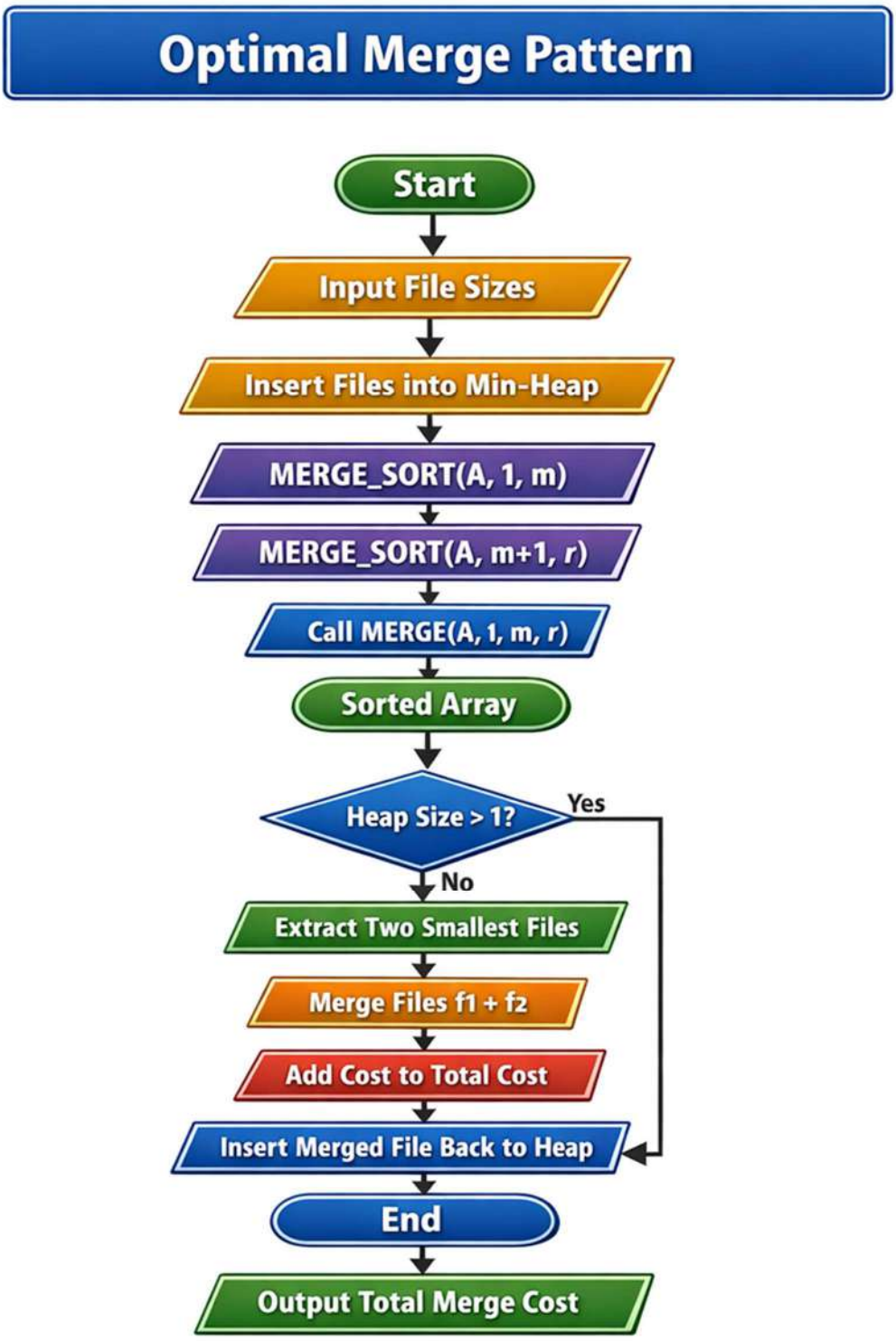
```cpp
#include <iostream>
#include <queue>
using namespace std;
```

```
int optimalMerge(int files[], int n) {
    priority_queue<int, vector<int>, greater<int>> pq(files,
files+n);
    int cost=0;
    while(pq.size()>1) {
        int f1=pq.top(); pq.pop();
        int f2=pq.top(); pq.pop();
        int merge=f1+f2;
        cost+=merge;
        pq.push(merge);
    }
    return cost;
}

int main() {
    int n; cin>>n;
    int files[n];
    for(int i=0;i<n;i++) cin>>files[i];
    cout<<"Optimal merge cost: "<<optimalMerge(files,n);
}
```

SAMPLE OUTPUT:

```
 Input: 4
20 30 10 5
Output: Optimal merge cost: 115
```

FLOWCHART:


Optimal Merge Pattern
Start
Input File Sizes
Insert Files into Min-Heap
MERGE_SORT(A, 1, m)
MERGE_SORT(A, m+1, r)
Call MERGE(A, 1, m, r)
Sorted Array
Heap Size > 1?
Yes
No
Extract Two Smallest Files
Merge Files f1 + f2
Add Cost to Total Cost
Insert Merged File Back to Heap
End
Output Total Merge Cost

EXPERIMENT NO.6

AIM:

To implement Huffman coding for lossless data compression

INTRODUCTION:

Huffman coding is a greedy algorithm used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes for more frequent characters. The algorithm constructs a binary tree (Huffman tree) based on character frequencies, ensuring optimal prefix codes and efficient encoding/decoding.

ALORITHM:

```
HUFFMAN(symbols, frequencies)
1. Create a min-heap of nodes with (symbol, frequency)
2. While heap size > 1:
   a. Extract two nodes with smallest frequencies
   b. Create new node with frequency = sum of both
   c. Set extracted nodes as left and right children
   d. Insert new node back into heap
3. Remaining node is root of Huffman tree
4. Traverse tree to assign codes:
   - Left edge = 0
   - Right edge = 1
```

SOURCE CODE:

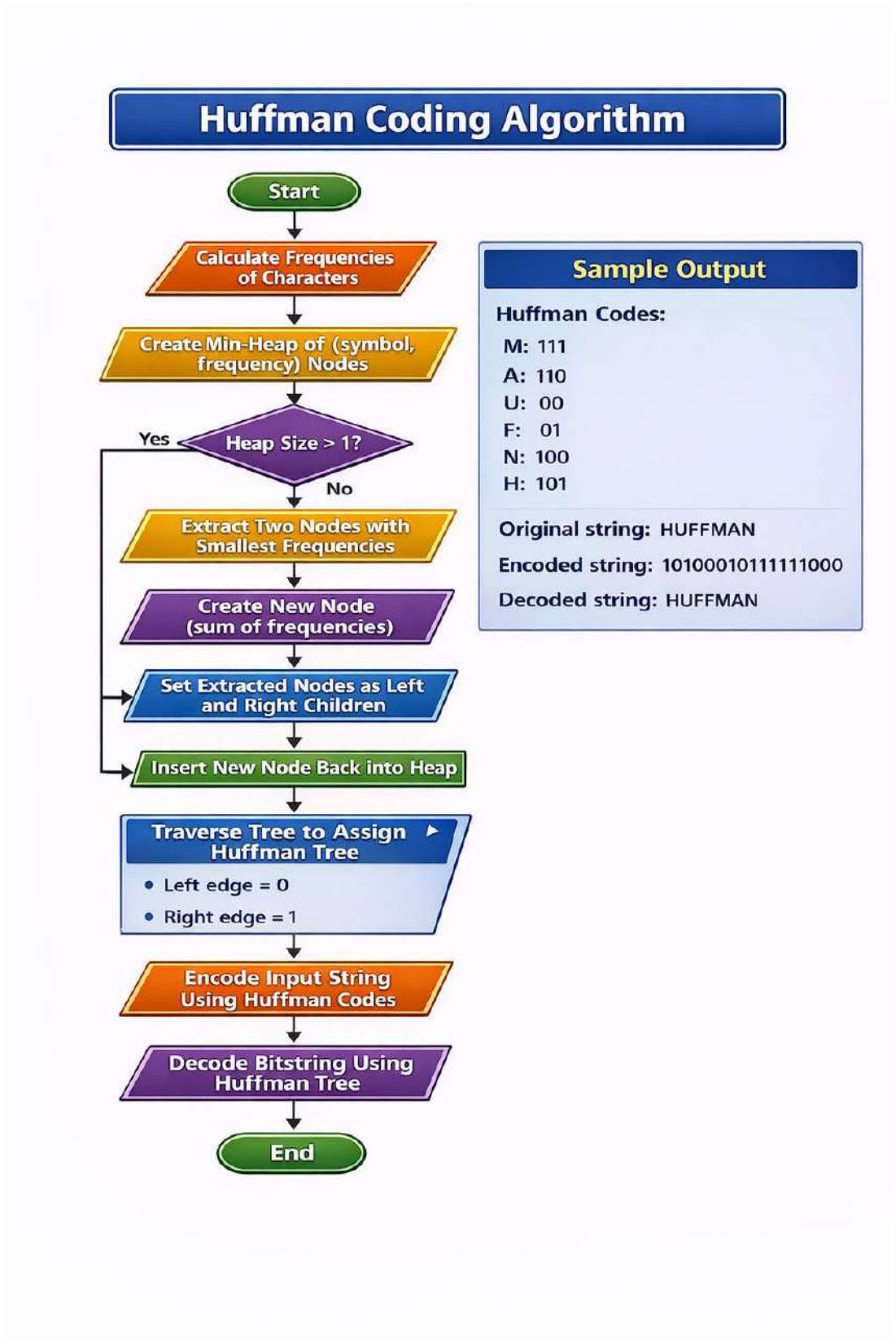
```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
using namespace std;

struct Node {
    char ch;
    int freq;
    Node* left;
    Node* right;
    Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
    Node(char c, int f, Node* l, Node* r) : ch(c), freq(f), left(l), right(r) {}
};

struct Compare {
    bool operator()(Node* l, Node* r) { return l->freq > r->freq; }
};

void buildCodes(Node* root, string str, unordered_map<char, string>& codes) {
    if (!root) return;
    if (!root->left && !root->right) codes[root->ch] = str;
    buildCodes(root->left, str + "0", codes);
    buildCodes(root->right, str + "1", codes);
}

void huffmanCoding(const string& text) {
    unordered_map<char, int> freq;
    for (char c : text) freq[c]++;
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (auto& p : freq) pq.push(new Node(p.first, p.second));
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        pq.push(new Node('\0', left->freq + right->freq, left, right));
    }
    Node* root = pq.top();
    unordered_map<char, string> codes;
    buildCodes(root, "", codes);
    cout << "Huffman Codes:\n";
    for (auto& p : codes) cout << p.first << ": " << p.second << endl;
    string encoded;
```

```
    for (char c : text) encoded += codes[c];
    cout << "Original string: " << text << endl;
    cout << "Encoded string: " << encoded << endl;
    // Decoding
    cout << "Decoded string: ";
    Node* curr = root;
    for (char bit : encoded) {
        curr = (bit == '0') ? curr->left : curr->right;
        if (!curr->left && !curr->right) {
            cout << curr->ch;
            curr = root;
        }
    }
    cout << endl;
}

int main() {
    string text = "HUFFMAN";
    huffmanCoding(text);
    return 0;
}
```

SAMPLE OUTPUT:

```
Huffman Codes:
M: 111
A: 110
U: 00
F: 01
N: 100
H: 101
Original string: HUFFMAN
Encoded string: 101000101111110100
Decoded string: HUFFMAN
```

FLOWCHART:


Huffman Coding Algorithm
Start
Calculate Frequencies of Characters
Create Min-Heap of (symbol, frequency) Nodes
Heap Size > 1?
Yes
No
Extract Two Nodes with Smallest Frequencies
Create New Node (sum of frequencies)
Set Extracted Nodes as Left and Right Children
Insert New Node Back into Heap
Traverse Tree to Assign Huffman Tree
Left edge = 0
Right edge = 1
Encode Input String Using Huffman Codes
Decode Bitstring Using Huffman Tree
End
Sample Output
Huffman Codes:
M: 111
A: 110
U: 00
F: 01
N: 100
H: 101
Original string: HUFFMAN
Encoded string: 10100010111111000
Decoded string: HUFFMAN

EXPERIMENT NO.7

AIM:

To implement Kruskal's algorithm for finding the minimum spanning tree (MST) of a weighted, undirected graph.

INTRODUCTION:

Kruskal's algorithm is a greedy method for constructing the MST of a connected, undirected graph. It sorts all edges by weight and adds the smallest edge that does not form a cycle, using a disjoint-set data structure for cycle detection. The process continues until the MST contains (V-1) edges, where V is the number of vertices.

ALGORITHM:

```
KRUSKAL(G)
1. Sort all edges of graph G in non-decreasing order of
weight
2. Initialize MST as empty
3. For each edge (u,v) in sorted list:
   a. If u and v belong to different sets:
        - Add edge (u,v) to MST
        - Union the sets of u and v
4. Continue until MST has (V-1) edges
5. Return MST
```

SOURCE CODE:

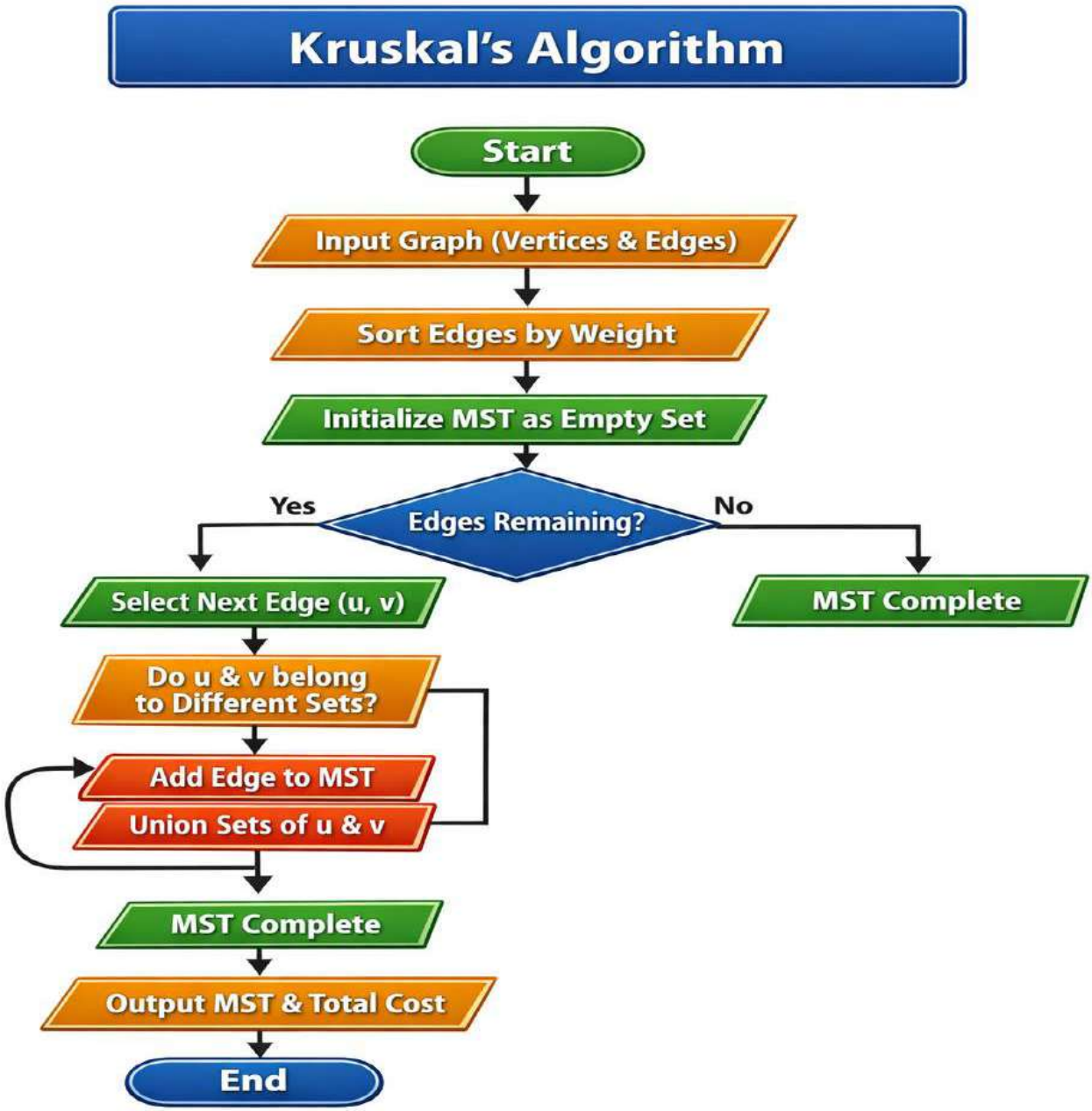
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const { return weight < other.weight; }
};

struct DSU {
    vector<int> parent, rank;
    DSU(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    void unite(int x, int y) {
        x = find(x); y = find(y);
        if (x != y) {
            if (rank[x] < rank[y]) swap(x, y);
            parent[y] = x;
            if (rank[x] == rank[y]) rank[x]++;
        }
    }
};

void kruskalMST(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    DSU dsu(V);
    int cost = 0;
    vector<Edge> mst;
    for (const auto& e : edges) {
        if (dsu.find(e.u) != dsu.find(e.v)) {
            dsu.unite(e.u, e.v);
            cost += e.weight;
            mst.push_back(e);
        }
    }
```

```
    cout << "Edges in MST:\n";
    for (const auto& e : mst)
        cout << e.u << " - " << e.v << " : " << e.weight << endl;
    cout << "Total cost: " << cost << endl;
}

int main() {
    int V = 4;
    vector<Edge> edges = {{0, 1, 10}, {1, 3, 15}, {2, 3, 4}, {2, 0, 6}, {0, 3,
5}};
    kruskalMST(V, edges);
    return 0;
}
```

SAMPLE OUTPUT:

```
Edges in MST:
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10
Total cost: 19
```

FLOWCHART:


Kruskal's Algorithm
Start
Input Graph (Vertices & Edges)
Sort Edges by Weight
Initialize MST as Empty Set
Edges Remaining?
Yes
No
Select Next Edge (u, v)
MST Complete
Do u & v belong to Different Sets?
Add Edge to MST
Union Sets of u & v
MST Complete
Output MST & Total Cost
End

EXPERIMENT NO. 8

AIM:

To implement Prim's algorithm for finding the minimum spanning tree of a weighted, undirected graph.

INTRODUCTION:

Prim's algorithm is a greedy approach that builds the MST by starting from an arbitrary vertex and repeatedly adding the smallest edge connecting the tree to a new vertex. It is efficient for dense graphs and can be implemented using a priority queue for optimal performance

ALGORITHM:

```
PRIM(G, start)
1. Initialize MST set with start vertex
2. Initialize key[v] = ∞ for all vertices except start
(key[start]=0)
3. While MST does not include all vertices:
   a. Select vertex u with minimum key not in MST
   b. Add u to MST
   c. For each neighbor v of u:
        If v not in MST and weight(u,v) < key[v]:
            key[v] = weight(u,v)
            parent[v] = u
4. Return MST edges
```

SOURCE CODE:

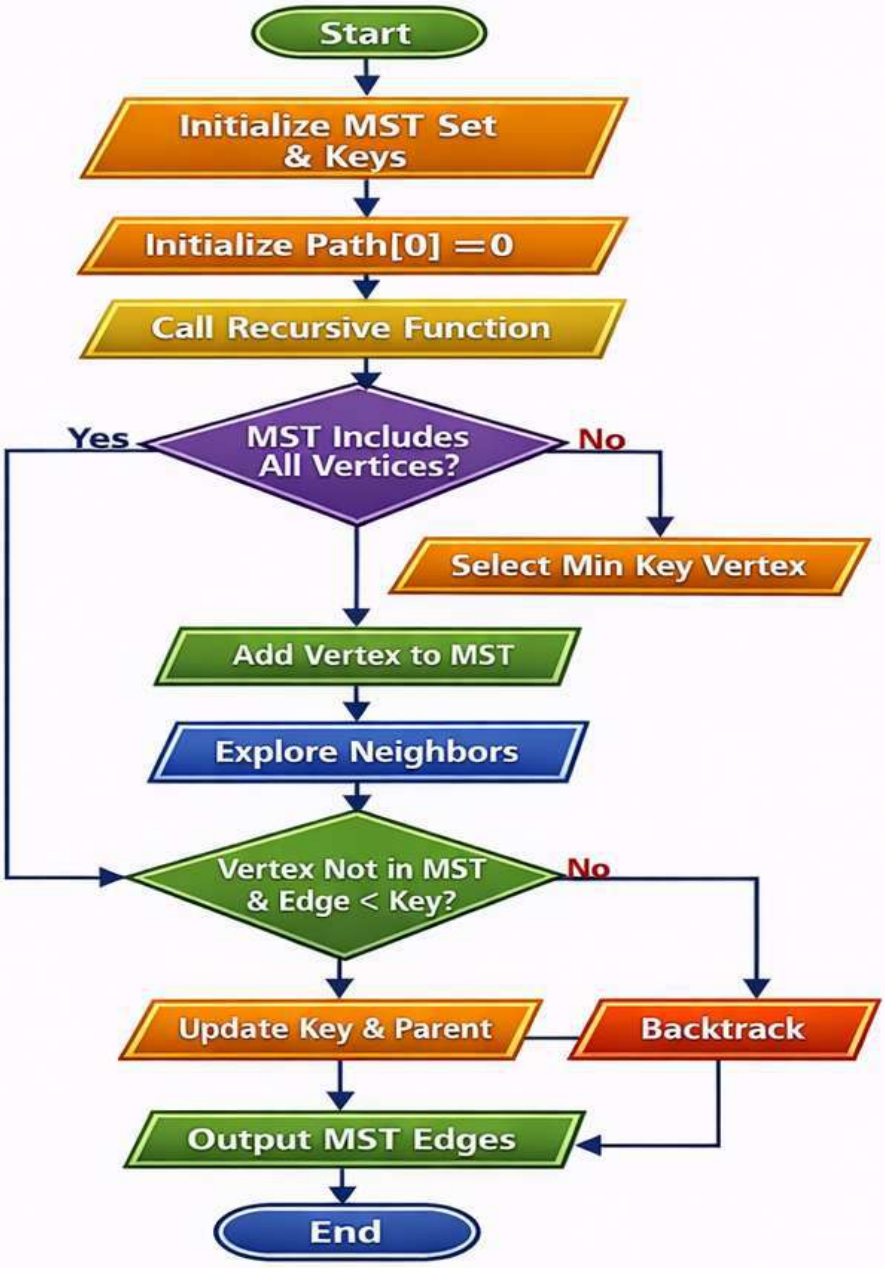
```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

typedef pair<int, int> pii;

void primMST(const vector<vector<pii>>& adj, int V) {
    vector<int> key(V, INT_MAX), parent(V, -1);
    vector<bool> inMST(V, false);
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    key[0] = 0;
    pq.push({0, 0});
    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        inMST[u] = true;
        for (auto& [v, w] : adj[u]) {
            if (!inMST[v] && w < key[v]) {
                key[v] = w;
                pq.push({key[v], v});
                parent[v] = u;
            }
        }
    }
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; ++i)
        cout << parent[i] << " - " << i << " \t" << key[i] << endl;
}

int main() {
    int V = 5;
    vector<vector<pii>> adj(V);
```

```
    adj[0] = {{1, 2}, {2, 3}};
    adj[1] = {{0, 2}, {3, 15}, {4, 2}};
    adj[2] = {{0, 3}, {3, 7}, {4, 1}};
    adj[3] = {{1, 15}, {2, 7}};
    adj[4] = {{1, 2}, {2, 1}};
    primMST(adj, V);
    return 0;
}
```

SAMPLE OUTPUT:

```
Edge   Weight
0 - 1  2
4 - 2  1
2 - 3  7
1 - 4  2
```

FLOWCHART:


Start
Initialize MST Set & Keys
Initialize Path[0] =0
Call Recursive Function
MST Includes All Vertices?
Yes
No
Select Min Key Vertex
Add Vertex to MST
Explore Neighbors
Vertex Not in MST & Edge < Key?
No
Update Key & Parent
Backtrack
Output MST Edges
End

EXPERIMENT NO. 9

AIM:

To implement **Dijkstra's** algorithm for finding the shortest paths from a single source to all other vertices in a weighted graph.

INTRODUCTION:

Dijkstra's algorithm is a greedy method for computing the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights. It maintains a set of vertices with known shortest distances and iteratively selects the vertex with the minimum tentative distance, updating neighbors accordingly. The algorithm is efficiently implemented using a priority queue.

ALGORITHM:

```
DIJKSTRA(G, source)
1. Initialize dist[v] = ∞ for all v, dist[source] = 0
2. Create priority queue Q with (source,0)
3. While Q not empty:
   a. Extract vertex u with minimum dist
   b. For each neighbor v of u:
        If dist[u] + weight(u,v) < dist[v]:
            dist[v] = dist[u] + weight(u,v)
            Update Q with (v, dist[v])
4. Return dist[]
```

SOURCE CODE:

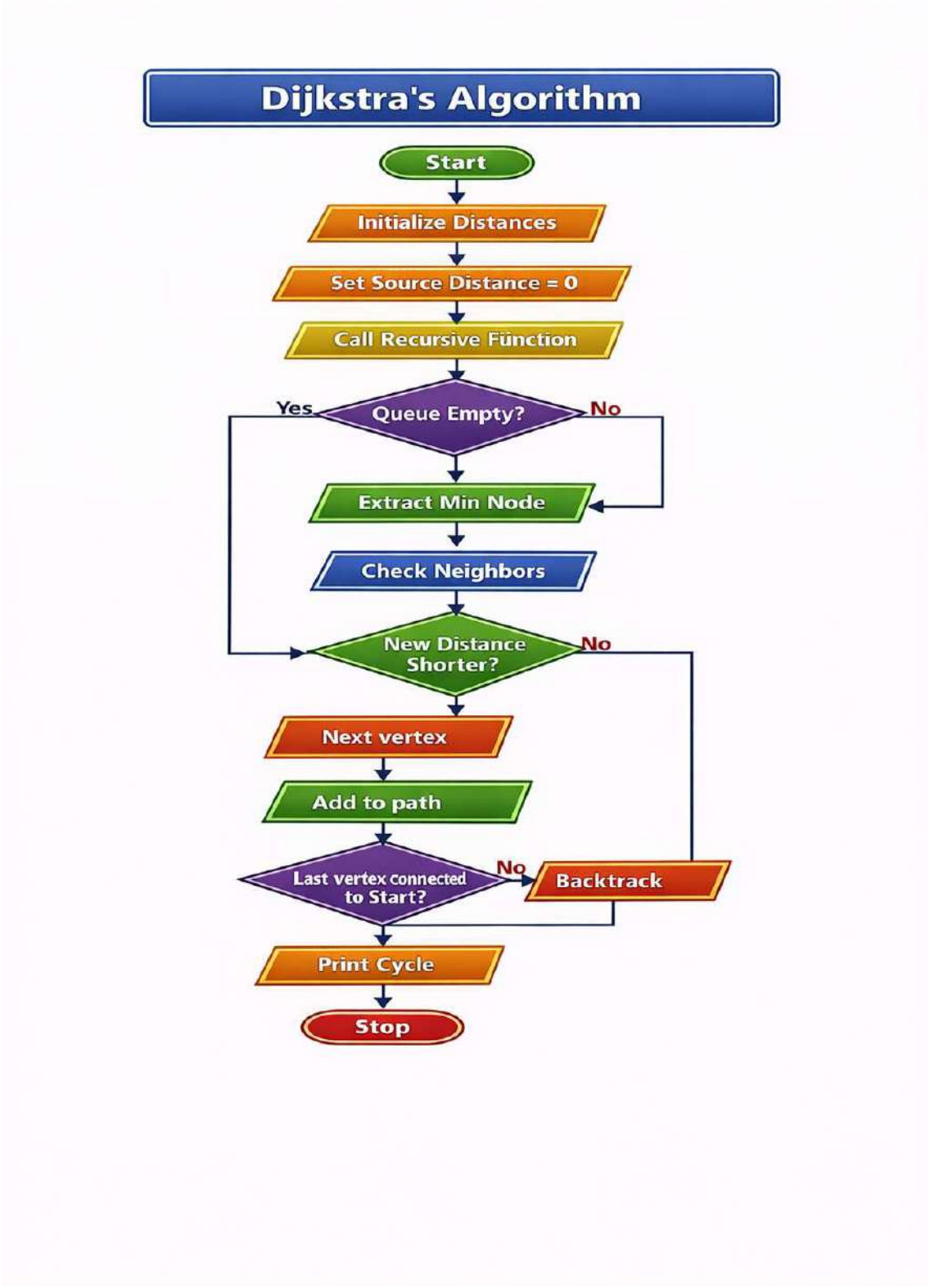
```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

typedef pair<int, int> pii;

void dijkstra(const vector<vector<pii>>& adj, int V, int src) {
    vector<int> dist(V, INT_MAX);
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    dist[src] = 0;
    pq.push({0, src});
    while (!pq.empty()) {
        int u = pq.top().second; pq.pop();
        for (auto& [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    cout << "Vertex\tDistance from Source\n";
    for (int i = 0; i < V; ++i)
        cout << i << "\t" << dist[i] << endl;
}

int main() {
    int V = 9;
    vector<vector<pii>> adj(V);
    adj[0] = {{1, 4}, {7, 8}};
    adj[1] = {{0, 4}, {2, 8}, {7, 11}};

    adj[2] = {{1, 8}, {3, 7}, {5, 4}, {8, 2}};
    adj[3] = {{2, 7}, {4, 9}, {5, 14}};
```

```
    adj[4] = {{3, 9}, {5, 10}};
    adj[5] = {{2, 4}, {3, 14}, {4, 10}, {6, 2}};
    adj[6] = {{5, 2}, {7, 1}, {8, 6}};
    adj[7] = {{0, 8}, {1, 11}, {6, 1}, {8, 7}};
    adj[8] = {{2, 2}, {6, 6}, {7, 7}};
    dijkstra(adj, V, 0);
    return 0;
```

SAMPLE OUTPUT:

```
Vertex  Distance from Source
0       0
1       4
2       12
3       19
4       21
5       11
6       9
7       8
8        14
```

FLOWCHART :


Dijkstra's Algorithm
Start
Initialize Distances
Set Source Distance = 0
Call Recursive Function
Queue Empty?
Yes
No
Extract Min Node
Check Neighbors
New Distance Shorter?
No
Next vertex
Add to path
Last vertex connected to Start?
No
Backtrack
Print Cycle
Stop

EXPERIMENT NO.10

AIM:

To implement the **Floyd-Warshall** algorithm for finding shortest paths between all pairs of vertices in a weighted graph**.**

INTRODUCTION:

The **Floyd-Warshall** algorithm is a dynamic programming method for computing shortest paths between all pairs of vertices in a graph. It iteratively updates the distance matrix by considering each vertex as an intermediate point, handling both positive and negative edge weights (but not negative cycles). The algorithm is particularly useful for dense graphs and transitive closure computations**.**

ALGORITHM :

```
FLOYD_WARSHALL(G)
1. Initialize dist[i][j] = weight(i,j) if edge exists, else ∞
2. For k = 1 to V:
   For i = 1 to V:
      For j = 1 to V:
         If dist[i][k] + dist[k][j] < dist[i][j]:
             dist[i][j] = dist[i][k] + dist[k][j]
3. Return dist matrix
```

SOURCE CODE:

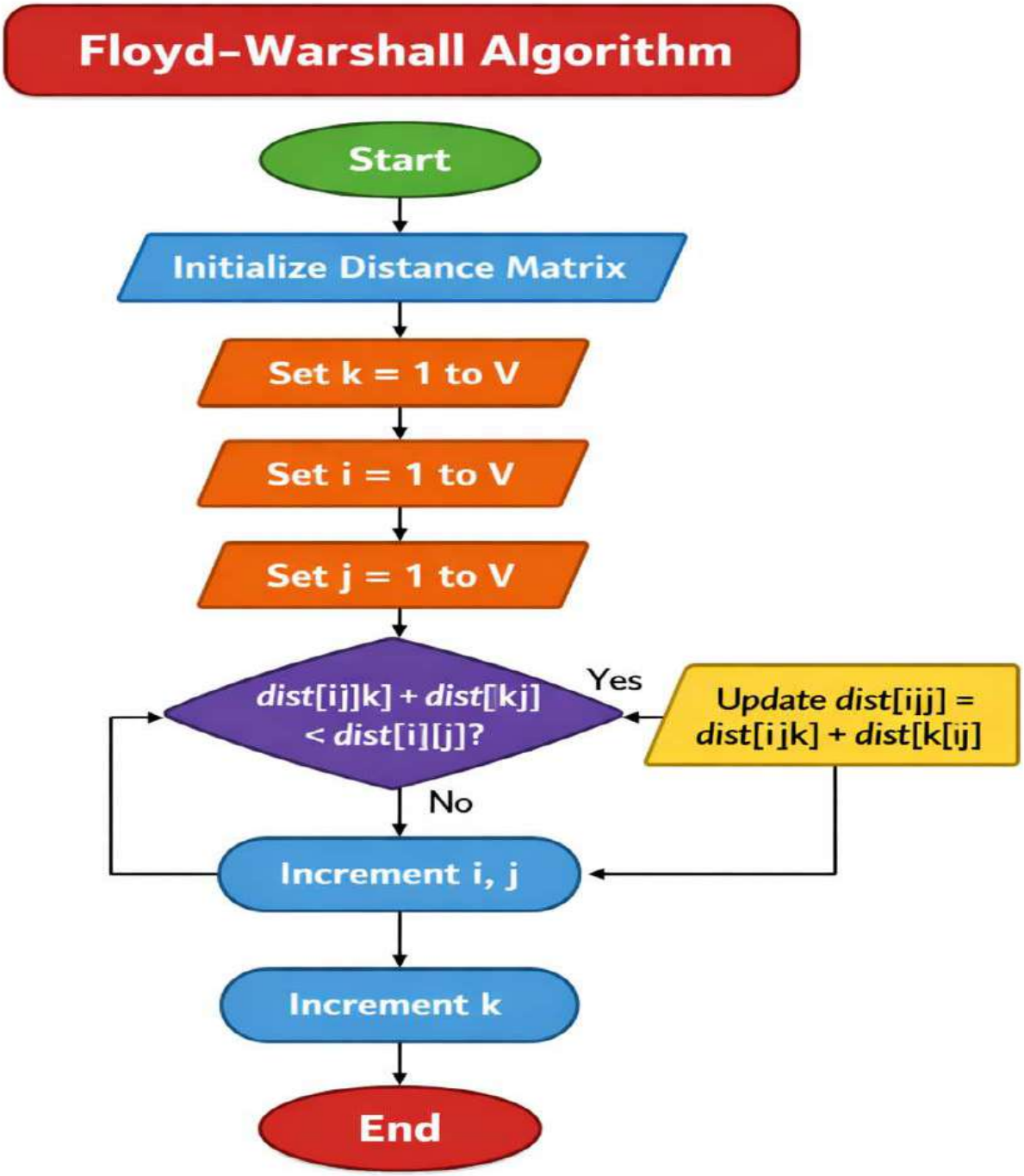
```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

#define INF INT_MAX

void floydWarshall(vector<vector<int>>& graph) {
    int V = graph.size();
```

```
    vector<vector<int>> dist = graph;
    for (int k = 0; k < V; ++k)
        for (int i = 0; i < V; ++i)
            for (int j = 0; j < V; ++j)
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
    cout << "Shortest distances between every pair of vertices:\n";
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (dist[i][j] == INF)
                cout << "INF\t";
            else
                cout << dist[i][j] << "\t";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> graph = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };
    floydWarshall(graph);
    return 0;
}
```

SAMPLE OUTPUT:

```
Shortest distances between every pair of vertices:
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0
```

FLOWCHART:


Floyd–Warshall Algorithm
Start
Initialize Distance Matrix
Set k = 1 to V
Set i = 1 to V
Set j = 1 to V
dist[ij]k] + dist[kj] < dist[i][j]?
Yes
Update dist[ijj] = dist[ijk] + dist[k[ij]
No
Increment i, j
Increment k
End

EXPERIMENT NO.11

AIM:

To implement the **Traveling Salesman** Problem using dynamic programming (Held-Karp algorithm).

INTRODUCTION:

The **Traveling Salesman Problem (TSP)** seeks the shortest possible route that visits each city exactly once and returns to the origin. TSP is NP-hard, and exact solutions are computationally expensive for large instances. The Held-Karp algorithm uses dynamic programming and bitmasking to solve TSP in $O(n^2 \cdot 2^n)$ time, making it feasible for moderate-sized graphs.

ALGORITHM :

```
TSP_BRANCH_BOUND(G)
1. Initialize best_cost = ∞
2. Start from source city
3. Use recursive function:
   a. Track current path and cost
   b. If all cities visited and return to source:
        Update best_cost if current_cost < best_cost
   c. Else:
        For each unvisited city:
            If current_cost + lower_bound < best_cost:
                Recurse with city added
4. Return best_cost and path
```

SOURCE CODE:

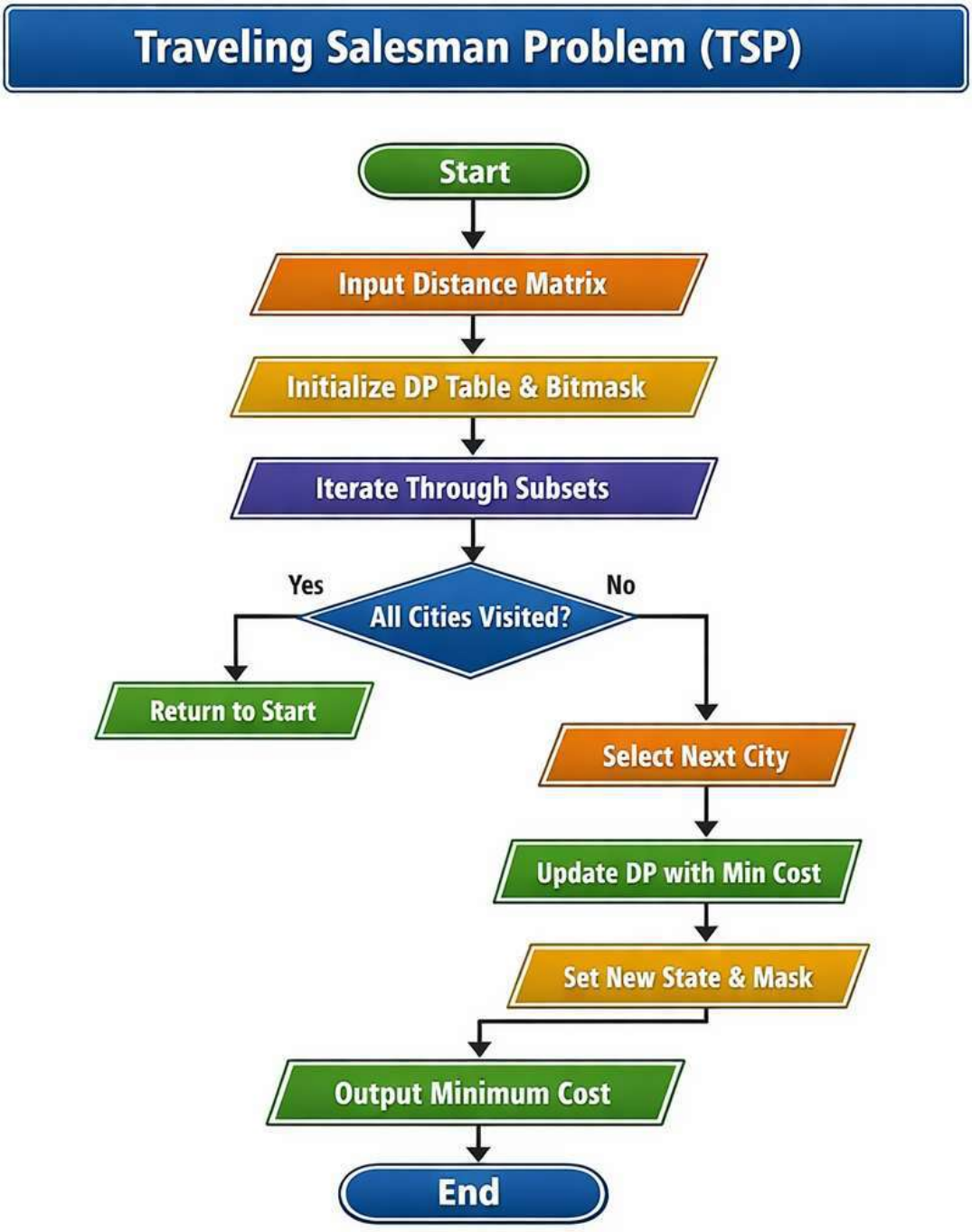
```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX / 2;

int tsp(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<vector<int>> dp(1 << n, vector<int>(n, INF));
    dp[1][0] = 0;
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int u = 0; u < n; ++u) {
            if (!(mask & (1 << u))) continue;
            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v) || graph[u][v] == 0) continue;
                dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v],
dp[mask][u] + graph[u][v]);
            }
        }
    }
    int res = INF;
    for (int u = 1; u < n; ++u)
        if (graph[u][0] != 0)
            res = min(res, dp[(1 << n) - 1][u] + graph[u][0]);
    return res;
}
```

```
int main() {
    vector<vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };
    int minCost = tsp(graph);
    cout << "Minimum cost: " << minCost << endl;
    return 0;
}
```

SAMPLE OUTPUT:

```
{0, 10, 15, 20}
{10, 0, 35, 25}
{15, 35, 0, 30}
{20, 25, 30, 0}
Optimal tour: 0 -> 1 -> 3 -> 2 -> 0
Minimum Cost: 80
```

FLOWCHART:


Traveling Salesman Problem (TSP)
Start
Input Distance Matrix
Initialize DP Table & Bitmask
Iterate Through Subsets
All Cities Visited?
Yes
No
Return to Start
Select Next City
Update DP with Min Cost
Set New State & Mask
Output Minimum Cost
End

EXPERIMENT NO.12

AIM:

To implement a backtracking algorithm to find a Hamiltonian cycle in a given graph.

INTRODUCTION:

A **Hamiltonian cycle** is a cycle that visits each vertex of a graph exactly once and returns to the starting vertex. Determining the existence of such a cycle is an NP-complete problem. The standard approach uses backtracking, recursively constructing paths and pruning invalid branches. This experiment demonstrates the use of backtracking and pruning in solving combinatorial problems.

ALGORITHM :

```
HAMILTONIAN_CYCLE(G)
1. Initialize path[] with first vertex
2. Define recursive function HAMILTONIAN_UTIL(pos):
   a. If pos == V:
        If edge exists from path[pos-1] to path[0]:
            Print path (Hamiltonian cycle found)
        Else return
   b. For v = 1 to V-1:
        If v is safe (adjacent to path[pos-1] and not already in
path):
            path[pos] = v
            Call HAMILTONIAN_UTIL(pos+1)
            Remove v (backtrack)
3. Call HAMILTONIAN_UTIL(1)
```

SOURCE CODE:

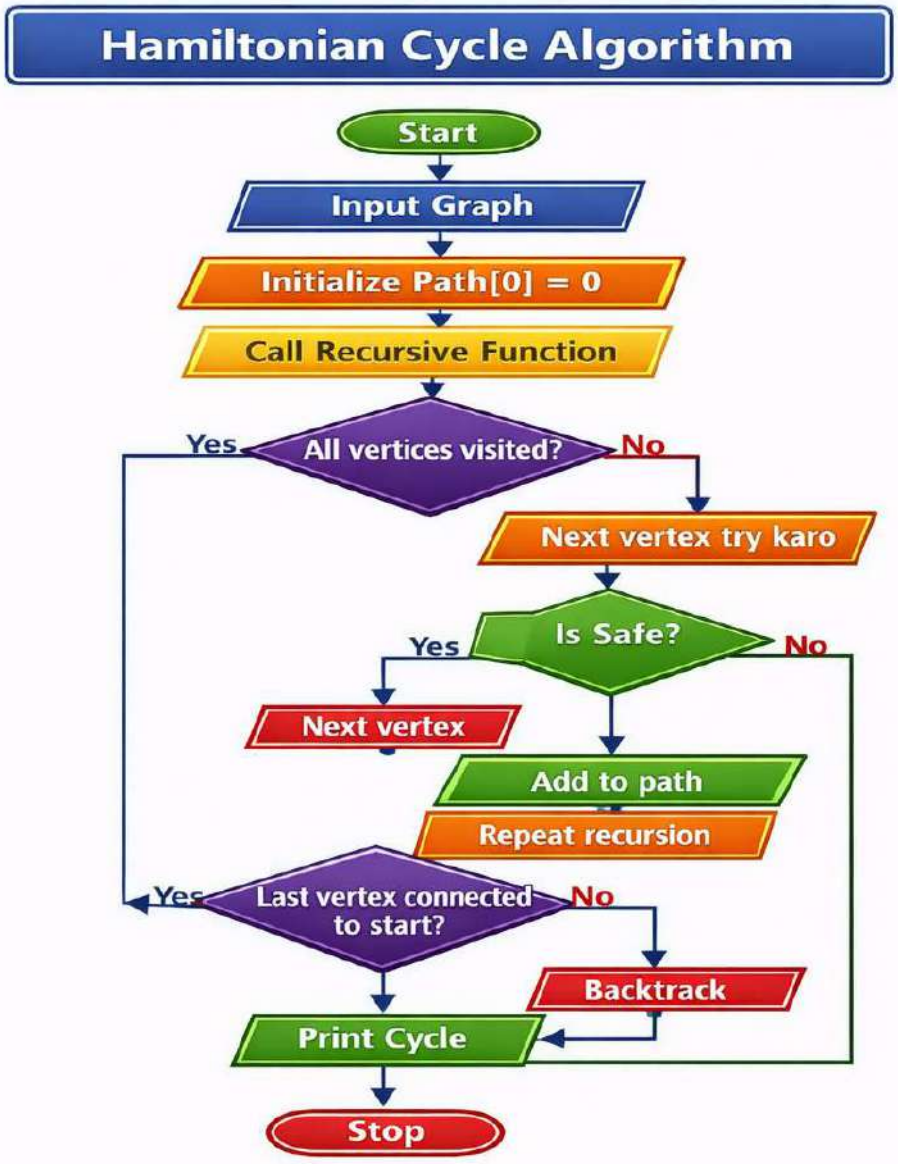
```
#include <iostream>
#include <vector>
using namespace std;

#define N 5

bool isSafe(int v, const vector<vector<int>>& graph, vector<int>& path, int pos) {
    if (graph[path[pos - 1]][v] == 0)
        return false;
    for (int i = 0; i < pos; ++i)
        if (path[i] == v)
            return false;
    return true;
}

bool hamiltonianCycleUtil(const vector<vector<int>>& graph, vector<int>& path, int pos) {
    if (pos == N) {
        return graph[path[pos - 1]][path[0]] == 1;
    }
    for (int v = 1; v < N; ++v) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1))
                return true;
            path[pos] = -1;
        }
    }
    return false;
}

bool hamiltonianCycle(const vector<vector<int>>& graph) {
    vector<int> path(N, -1);
    path[0] = 0;

    if (!hamiltonianCycleUtil(graph, path, 1)) {
        cout << "Solution does not exist" << endl;
        return false;
    }

    cout << "Hamiltonian Cycle: ";
    for (int i = 0; i < N; ++i) {
```

```
        cout << path[i];
        if (i != N - 1) cout << " -> ";
    }
    cout << " -> " << path[0] << endl;

    return true;
}

int main() {
    vector<vector<int>> graph = {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };

    hamiltonianCycle(graph);
    return 0;
}
```

SAMPLE OUTPUT:

```
  {
          {0, 1, 0, 1, 0},
          {1, 0, 1, 1, 1},
          {0, 1, 0, 0, 1},
          {1, 1, 0, 0, 1},
          {0, 1, 1, 1, 0}
      };


Hamiltonian Cycle: 0 -> 1 -> 2 -> 4 -> 3 -> 0
```

FLOWCHART:


Hamiltonian Cycle Algorithm
Start
Input Graph
Initialize Path[0] = 0
Call Recursive Function
All vertices visited?
Yes
No
Next vertex try karo
Is Safe?
Yes
No
Next vertex
Add to path
Repeat recursion
Last vertex connected to start?
Yes
No
Backtrack
Print Cycle
Stop