UCS415 Design and Analysis of Algorithms Lab Solutions

Lab Assignment 4 (Backtracking)

Question 1: N-Queens Problem

Algorithm:

```
1. Start with an empty N×N chessboard.
2. Start placing queens one by one, beginning with the leftmost column.
3. For each column:
   a. Try placing a queen in each row.
   b. Check if the current position is safe for the queen (no attacks).
   c. If safe, mark this position and recursively try placing queens in remaining columns.
   d. If placing queens in remaining columns doesn't lead to a solution, backtrack:
      - Remove the queen from current position.
      - Try the next row in the same column.
4. If all columns are filled, a solution is found.
```

C++ Implementation:

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

bool isSafe(vector<vector<int>>& board, int row, int col, int n) {
    // Check row on left side
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (int i = row, j = col; j >= 0 && i < n; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(vector<vector<int>>& board, int col, int n, vector<int>& result) {
    // Base case: If all queens are placed
    if (col >= n)
```

```
        return true;

    // Try placing queen in all rows of this column
    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            // Place the queen
            board[i][col] = 1;
            result[col] = i + 1; // +1 for 1-based indexing

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1, n, result))
                return true;

            // If placing queen doesn't lead to a solution, backtrack
            board[i][col] = 0; // Backtrack
            result[col] = 0;
        }
    }

    // If queen can't be placed in any row in this column
    return false;
}

vector<int> solveNQ(int n) {
    vector<vector<int>> board(n, vector<int>(n, 0));
    vector<int> result(n, 0);

    if (!solveNQUtil(board, 0, n, result)) {
        cout << "Solution does not exist" << endl;
        return {};
    }

    return result;
}

int main() {
    int n = 4; // Change this to the desired board size
    vector<int> solution = solveNQ(n);

    if (!solution.empty()) {
        cout << "Solution exists: [";
        for (int i = 0; i < solution.size(); i++) {
            cout << solution[i];
            if (i < solution.size() - 1)
                cout << ", ";
        }
        cout << "]" << endl;
```

```
    }

    return 0;
}
```

Output:

```
Output

Solution exists: [2, 4, 1, 3]


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

Question 2: Sudoku Solver

Algorithm:

```
1. Find an empty cell in the Sudoku grid.
2. Try placing digits from 1-9 in this empty cell.
3. For each digit:
   a. Check if placing this digit is valid (follows Sudoku rules).
   b. If valid, place the digit and recursively try to fill the rest of the grid.
   c. If this leads to a solution, return true.
   d. If not, backtrack: Remove the digit and try the next one.
4. If no digit works, return false to trigger backtracking.
5. If no empty cell is found, the Sudoku is solved.
```

C++ Implementation:

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

bool isSafe(vector<vector<int>>& grid, int row, int col, int num) {
    // Check if 'num' is not present in current row
    for (int x = 0; x < 9; x++)
        if (grid[row][x] == num)
            return false;

    // Check if 'num' is not present in current column
    for (int x = 0; x < 9; x++)
        if (grid[x][col] == num)
            return false;

    // Check if 'num' is not present in current 3x3 box
    int startRow = row - row % 3;
    int startCol = col - col % 3;
```

```
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grid[i + startRow][j + startCol] == num)
                return false;

    return true;
}

bool findEmptyLocation(vector<vector<int>>& grid, int& row, int& col) {
    for (row = 0; row < 9; row++)
        for (col = 0; col < 9; col++)
            if (grid[row][col] == 0)
                return true;
    return false;
}

bool solveSudoku(vector<vector<int>>& grid) {
    int row, col;

    // If there is no empty location, we are done
    if (!findEmptyLocation(grid, row, col))
        return true;

    // Try digits 1 to 9
    for (int num = 1; num <= 9; num++) {
        // Check if safe to place
        if (isSafe(grid, row, col, num)) {
            // Place the digit
            grid[row][col] = num;

            // Recur to fill rest of the grid
            if (solveSudoku(grid))
                return true;

            // If placing digit doesn't lead to a solution, backtrack
            grid[row][col] = 0;
        }
    }

    // Trigger backtracking
    return false;
}

void printGrid(vector<vector<int>>& grid) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++)
            cout << grid[row][col] << " ";
```

```
        cout << endl;
    }
}

int main() {
    // 0 represents empty cells
    vector<vector<int>> grid = {
        {3, 0, 6, 5, 0, 8, 4, 0, 0},
        {5, 2, 0, 0, 0, 0, 0, 0, 0},
        {0, 8, 7, 0, 0, 0, 0, 3, 1},
        {0, 0, 3, 0, 1, 0, 0, 8, 0},
        {9, 0, 0, 8, 6, 3, 0, 0, 5},
        {0, 5, 0, 0, 9, 0, 6, 0, 0},
        {1, 3, 0, 0, 0, 0, 2, 5, 0},
        {0, 0, 0, 0, 0, 0, 0, 7, 4},
        {0, 0, 5, 2, 0, 6, 3, 0, 0}
    };

    if (solveSudoku(grid)) {
        cout << "Sudoku Solution:" << endl;
        printGrid(grid);
    } else {
        cout << "No solution exists" << endl;
    }

    return 0;
}
```

Output:

Output

```
Sudoku Solution:
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9


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

Question 3: Graph Coloring Problem

Algorithm:

```
1. Assign colors to vertices one by one, starting from vertex 0.
2. For each vertex:
```

```
    a. Try all possible colors (1 to m).
    b. Check if the color can be assigned:
       - Verify that no adjacent vertex has the same color.
    c. If a color can be assigned, mark it and recursively assign colors to the rest of the vertices.
    d. If color assignment to remaining vertices is not possible, backtrack:
       - Remove color assignment for the current vertex.
       - Try the next color.
3. If all vertices are assigned colors, return true.
```

C++ Implementation:

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

bool isSafe(vector<vector<int>>& graph, vector<int>& color, int v, int c, int V) {
    for (int i = 0; i < V; i++)
        if (graph[v][i] && color[i] == c)
            return false;
    return true;
}

bool graphColoringUtil(vector<vector<int>>& graph, int m, vector<int>& color, int v, int V) {
    // Base case: If all vertices are assigned a color
    if (v == V)
        return true;

    // Try different colors for vertex v
    for (int c = 1; c <= m; c++) {
        // Check if assignment of color c to v is valid
        if (isSafe(graph, color, v, c, V)) {
            color[v] = c;

            // Recur to assign colors to rest of the vertices
            if (graphColoringUtil(graph, m, color, v + 1, V))
                return true;

            // If assigning color c doesn't lead to a solution, remove it
            color[v] = 0;
        }
    }

    // If no color can be assigned to this vertex
    return false;
}
```

```
bool graphColoring(vector<vector<int>>& graph, int m, int V) {
    vector<int> color(V, 0); // Initialize all colors as 0 (unassigned)

    if (!graphColoringUtil(graph, m, color, 0, V)) {
        cout << "Solution does not exist" << endl;
        return false;
    }

    cout << "Solution Exists: Following are the assigned colors:" << endl;
    for (int i = 0; i < V; i++)
        cout << color[i] << " ";
    cout << endl;

    return true;
}

int main() {
    vector<vector<int>> graph = {
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
    };
    int m = 3; // Number of colors
    int V = 4; // Number of vertices

    graphColoring(graph, m, V);

    return 0;
}
```

Output:

Output

```
Solution Exists: Following are the assigned colors:
1 2 3 2


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

Lab Assignment 5 (Graph Algorithms)

Question 1: Hamiltonian Circuit in a Graph

Algorithm:

```
1. Create a path array to store the Hamiltonian Circuit.
2. Initialize path[0] as vertex 0 (starting vertex).
3. For subsequent positions in the path:
   a. Try adding each vertex that's adjacent to the previously added vertex.
   b. Check if this vertex can be added to the path:
      - It should be adjacent to the last vertex in the path.
      - It should not already be in the path.
   c. If a vertex can be added, add it and recursively check for remaining vertices.
   d. If adding this vertex doesn't lead to a solution, backtrack by removing it.
4. Once all vertices are added (path length = V), check if the last vertex is adjacent to
the first vertex.
5. If yes, a Hamiltonian Circuit exists.
```

C++ Implementation:

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

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& path, int pos) {
    // Check if this vertex is adjacent to the previously added vertex
    if (graph[path[pos-1]][v] == 0)
        return false;

    // Check if the vertex has already been included in the path
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

bool hamiltonianCircuitUtil(vector<vector<int>>& graph, vector<int>& path, int pos, int V) {
    // Base case: If all vertices are included
    if (pos == V) {
        // Check if there's an edge from the last vertex to the first vertex
        if (graph[path[pos-1]][path[0]] == 1)
            return true;
```

```cpp
        else
            return false;
    }

    // Try different vertices for the current position
    for (int v = 0; v < V; v++) {
        // Check if this vertex can be added
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;

            // Recur to construct the rest of the path
            if (hamiltonianCircuitUtil(graph, path, pos + 1, V))
                return true;

            // If adding vertex v doesn't lead to a solution, remove it
            path[pos] = -1;
        }
    }

    // If no vertex can be added to the path
    return false;
}
bool hamiltonianCircuit(vector<vector<int>>& graph, int V) {
    vector<int> path(V, -1);
    path[0] = 0; // Start from vertex 0

    if (!hamiltonianCircuitUtil(graph, path, 1, V)) {
        cout << "Hamiltonian Circuit does not exist" << endl;
        return false;
    }

    cout << "Hamiltonian Circuit exists: ";
    for (int i = 0; i < V; i++)
        cout << path[i] << " ";
    cout << path[0] << endl; // Print the first vertex again to complete the circuit

    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}
    };
```

```
    int V = 5;

    hamiltonianCircuit(graph, V);

    return 0;
}
```

Output:

Output

```
Hamiltonian Circuit exists: 0 1 2 4 3 0


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

Question 2: Topological Sort (Kahn's Algorithm and DFS)

Algorithm (Kahn's Algorithm):

```
1. Compute in-degree for each vertex (number of incoming edges).
2. Create a queue and enqueue all vertices with in-degree 0.
3. While the queue is not empty:
   a. Dequeue a vertex and add it to the result.
   b. Reduce in-degree of all adjacent vertices by 1.
   c. If in-degree of an adjacent vertex becomes 0, enqueue it.
4. If count of visited vertices is not equal to the total vertices, the graph has a cycle.
```

Algorithm (DFS Approach):

```
1. Create a temporary stack to store the result.
2. Create a visited array to keep track of visited vertices.
3. For each unvisited vertex:
   a. Call DFS recursive function.
   b. In the recursive function:
      - Mark the current vertex as visited.
      - For each adjacent vertex, if not visited, recursively call the function.
      - After all adjacent vertices are processed, push current vertex to stack.
4. Print the contents of the stack in reverse order.
```

C++ Implementation:

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

```
// Kahn's Algorithm
vector<int> topologicalSortKahn(vector<vector<int>>& adj, int V) {
    vector<int> result;
    vector<int> inDegree(V, 0);

    // Calculate in-degree for each vertex
    for (int u = 0; u < V; u++) {
        for (int v : adj[u]) {
            inDegree[v]++;
        }
    }

    // Enqueue vertices with in-degree 0
    queue<int> q;
    for (int i = 0; i < V; i++) {
        if (inDegree[i] == 0)
            q.push(i);
    }

    int count = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        // Reduce in-degree of adjacent vertices
        for (int v : adj[u]) {
            if (--inDegree[v] == 0)
                q.push(v);
        }

        count++;
    }

    // Check if there was a cycle
    if (count != V) {
        cout << "Graph contains a cycle, topological sort not possible" << endl;
        return {};
    }
    return result;
}

// DFS based approach
void topologicalSortDFSUtil(vector<vector<int>>& adj, int v, vector<bool>& visited, stack<int>&
    visited[v] = true;

    for (int u : adj[v]) {
```

```
        if (!visited[u])
            topologicalSortDFSUtil(adj, u, visited, Stack);
    }

    Stack.push(v);
}

vector<int> topologicalSortDFS(vector<vector<int>>& adj, int V) {
    stack<int> Stack;
    vector<bool> visited(V, false);
    vector<int> result;

    for (int i = 0; i < V; i++) {
        if (!visited[i])
            topologicalSortDFSUtil(adj, i, visited, Stack);
    }

    while (!Stack.empty()) {
        result.push_back(Stack.top());
        Stack.pop();
    }

    return result;
}

int main() {
    int V = 6;
    vector<vector<int>> adj(V);

    // Add edges for the given example
    adj[5].push_back(0);
    adj[5].push_back(2);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(1);

    cout << "Topological Sort using Kahn's Algorithm:" << endl;
    vector<int> result1 = topologicalSortKahn(adj, V);
    for (int i : result1)
        cout << i << " ";
    cout << endl;

    cout << "Topological Sort using DFS:" << endl;
    vector<int> result2 = topologicalSortDFS(adj, V);
    for (int i : result2)
```

```cpp
        cout << i << " ";
    cout << endl;

    return 0;
}
```

Output:

```
Output
Topological Sort using Kahn's Algorithm:
4 5 0 2 3 1
Topological Sort using DFS:
5 4 2 3 1 0


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

Question 3: Strongly Connected Components (Kosaraju's Algorithm)

Algorithm:

```
1. Create an empty stack and do DFS traversal of the graph. Push vertices to stack when all
adjacent vertices are processed.
2. Transpose the graph (reverse all edges).
3. Pop vertices from the stack one by one:
   a. For each popped vertex, if it hasn't been visited:
      - Perform DFS starting from this vertex in the transposed graph.
      - All vertices visited in this DFS form one strongly connected component.
4. Count the total number of such DFS calls to get the number of SCCs.
```

C++ Implementation:

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

void fillOrder(vector<vector<int>>& adj, int v, vector<bool>& visited, stack<int>& Stack) {
    visited[v] = true;

    for (int u : adj[v]) {
        if (!visited[u])
            fillOrder(adj, u, visited, Stack);
    }

    Stack.push(v);
}

void DFSUtil(vector<vector<int>>& adj, int v, vector<bool>& visited) {
    visited[v] = true;
```

```
    cout << v << " ";

    for (int u : adj[v]) {
        if (!visited[u])
            DFSUtil(adj, u, visited);
    }
}

int kosarajuSCC(vector<vector<int>>& adj, int V) {
    stack<int> Stack;
    vector<bool> visited(V, false);

    // Step 1: Fill vertices in stack according to their finishing times
    for (int i = 0; i < V; i++) {
        if (!visited[i])
            fillOrder(adj, i, visited, Stack);
    }

    // Step 2: Create transpose of graph
    vector<vector<int>> transpose(V);
    for (int v = 0; v < V; v++) {
        for (int u : adj[v]) {
            transpose[u].push_back(v);
        }
    }

    // Step 3: Process vertices in order defined by the stack
    fill(visited.begin(), visited.end(), false);
    int count = 0;

    while (!Stack.empty()) {
        int v = Stack.top();
        Stack.pop();

        if (!visited[v]) {
            cout << "SCC " << count + 1 << ": ";
            DFSUtil(transpose, v, visited);
            cout << endl;
            count++;
        }
    }

    return count;
}

int main() {
    int V = 5;
```

```
    vector<vector<int>> adj(V);

    // Add edges for a graph with 3 SCCs
    adj[0].push_back(2);
    adj[1].push_back(0);
    adj[2].push_back(1);
    adj[0].push_back(3);
    adj[3].push_back(4);

    int scc_count = kosarajuSCC(adj, V);
    cout << "Total number of Strongly Connected Components: " << scc_count << endl;

    return 0;
}
```

Output:

```
Output
SCC 1: 0 1 2
SCC 2: 3
SCC 3: 4
Total number of Strongly Connected Components: 3


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

Lab Assignment 6 (Graph Algorithms)

Question 1: Ford-Fulkerson Algorithm for Maximum Flow

Algorithm:

```
1. Initialize residual graph with the same capacities as the original graph.
2. Initialize flow to 0.
3. While there exists an augmenting path from source to sink in the residual graph:
   a. Find augmenting path using BFS or DFS.
   b. Find the minimum capacity of the augmenting path.
   c. Increase flow by this minimum capacity.
   d. Update residual capacities:
      - Reduce capacity on edges in the path by the minimum capacity.
      - Increase capacity on reverse edges by the minimum capacity.
4. Return the total flow.
```

C++ Implementation:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits.h>
```

```cpp
using namespace std;

bool bfs(vector<vector<int>>& residualGraph, int source, int sink, vector<int>& parent, int V)
    vector<bool> visited(V, false);
    queue<int> q;
    q.push(source);
    visited[source] = true;
    parent[source] = -1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < V; v++) {
            if (!visited[v] && residualGraph[u][v] > 0) {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    return visited[sink];
}

int fordFulkerson(vector<vector<int>>& graph, int source, int sink, int V) {
    vector<vector<int>> residualGraph = graph;
    vector<int> parent(V);
    int max_flow = 0;

    // Augment the flow while there is a path from source to sink
    while (bfs(residualGraph, source, sink, parent, V)) {
        // Find the minimum residual capacity of the edges along the path
        int path_flow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            path_flow = min(path_flow, residualGraph[u][v]);
        }

        // Update residual capacities and reverse edges
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= path_flow;
            residualGraph[v][u] += path_flow;
        }

        max_flow += path_flow;
```

```
    }

    return max_flow;
}

int main() {
    int V = 6; // Number of vertices
    vector<vector<int>> graph = {
        {0, 16, 13, 0, 0, 0},
        {0, 0, 10, 12, 0, 0},
        {0, 4, 0, 0, 14, 0},
        {0, 0, 9, 0, 0, 20},
        {0, 0, 0, 7, 0, 4},
        {0, 0, 0, 0, 0, 0}
    };

    int source = 0;
    int sink = 5;

    int max_flow = fordFulkerson(graph, source, sink, V);
    cout << "Maximum flow from source to sink: " << max_flow << endl;

    return 0;
}
```

Output:

Output

```
Maximum flow from source to sink: 23


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

Lab Assignment 7 (String Matching Algorithms)

Question 1: Brute Force String Matching

Algorithm:

```
1. For each possible starting position i in the text (0 to N-M):
   a. Check if the pattern matches with the substring of text starting at position i:
      - Compare each character of pattern with corresponding character in text.
      - If a mismatch is found, break and try the next position.
   b. If all characters match, a pattern occurrence is found at position i.
2. Return all positions where pattern is found.
```

C++ Implementation:

```
#include <iostream>
```

```cpp
#include <string>
#include <vector>
using namespace std;

vector<int> bruteForceStringMatch(string txt, string pat) {
    vector<int> positions;
    int N = txt.size();
    int M = pat.size();

    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j])
                break;
        }

        if (j == M) // Pattern found
            positions.push_back(i + 1); // +1 for one-based indexing
    }

    return positions;
}

int main() {
    string txt = "THIS IS A TEST TEXT";
    string pat = "TEST";

    vector<int> positions = bruteForceStringMatch(txt, pat);

    if (positions.empty()) {
        cout << "Pattern not found" << endl;
    } else {
        for (int pos : positions)
            cout << "Pattern found at index " << pos << endl;
    }

    txt = "AABAACAADAABAABA";
    pat = "AABA";

    positions = bruteForceStringMatch(txt, pat);

    if (positions.empty()) {
        cout << "Pattern not found" << endl;
    } else {
        for (int pos : positions)
```

```
            cout << "Pattern found at index " << pos << endl;
        }

        return 0;
    }
```

Output:

Output

```
Pattern found at index 11
Pattern found at index 1
Pattern found at index 10
Pattern found at index 13


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

Question 2: Rabin-Karp String Matching Algorithm

Algorithm:

```
1. Calculate hash value for the pattern and for the first window of text.
2. Iterate through the text:
   a. Compare hash values of current window and pattern.
   b. If hash values match, check character by character to confirm a match.
   c. If match is found, record position.
   d. Slide the window by one character:
      - Remove the leftmost character of the previous window.
      - Add the new character to the right of the window.
      - Recalculate hash value efficiently using rolling hash.
3. Return all positions where pattern is found.
```

C++ Implementation:

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

// A prime number for hash calculation
const int PRIME = 101;

vector<int> rabinKarp(string txt, string pat) {
    vector<int> positions;
    int N = txt.size();
    int M = pat.size();
    int i, j;
    int patHash = 0; // Hash value for pattern
    int txtHash = 0; // Hash value for current window of text
```

```
    int h = 1;

    // Calculate h = 256^(M-1) % PRIME
    for (i = 0; i < M - 1; i++)
        h = (h * 256) % PRIME;

    // Calculate hash value for pattern and first window of text
    for (i = 0; i < M; i++) {
        patHash = (256 * patHash + pat[i]) % PRIME;
        txtHash = (256 * txtHash + txt[i]) % PRIME;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {
        // Check if hash values match
        if (patHash == txtHash) {
            // Check for characters one by one
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j])
                    break;
            }

            if (j == M) // Pattern found
                positions.push_back(i + 1); // +1 for one-based indexing
        }

        // Calculate hash value for next window: Remove leading digit, add trailing
digit
        if (i < N - M) {
            txtHash = (256 * (txtHash - txt[i] * h) + txt[i + M]) % PRIME;

            // We might get negative value of txtHash, convert it to positive
            if (txtHash < 0)
                txtHash += PRIME;
        }
    }

    return positions;
}

int main() {
    string txt = "THIS IS A TEST TEXT";
    string pat = "TEST";

    vector<int> positions = rabinKarp(txt, pat);

    if (positions.empty()) {
```

```
        cout << "Pattern not found" << endl;
    } else {
        for (int pos : positions)
            cout << "Pattern found at index " << pos << endl;
    }

    txt = "AABAACAADAABAABA";
    pat = "AABA";

    positions = rabinKarp(txt, pat);

    if (positions.empty()) {
        cout << "Pattern not found" << endl;
    } else {
        for (int pos : positions)
            cout << "Pattern found at index " << pos << endl;
    }

    return 0;
}
```

Output:

Output

```
Pattern found at index 11
Pattern found at index 1
Pattern found at index 10
Pattern found at index 13


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