

Department of Computer Science and Engineering

Course Code: CSE246

Course Title: Algorithms

Section: 04

Semester: Fall 2023

**Project Report**

**Submitted to:**

**Sadika Islam Sneha**

Lecturer

Department of Computer Science and Engineering

East West University

**Submitted by:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| B M Shahria Alam | 2021-3-60-016 |
| Golam Kibria | 2021-3-60-215 |
| Sidratul Moontaha | 2021-3-60-048 |
| MD Imran Khan | 2021-3-60-206 |

**Date of submission:** 24th December, 2023.

**Problem 01: Traveling Salesman Problem**

Solution:

**TAG:** Dynamic programming, Implementation

**COMPLEXITY:**  O(N!), where N is the number of cities.

* The time complexity of the TSP algorithm dominates the overall complexity. The worst-case time complexity is O(n!) due to the exhaustive search of all possible permutations.
* The space complexity is O(n^2) for the graph and O(n) for the visited and path arrays.

To solve the problem, we can use backtracking. We start by generating all possible permutations of the cities and calculate the cost of each permutation. We then select the permutation with the minimum cost. The time complexity of this approach is O(n!), where n is the number of cities. However, this approach is not practical for large values of n.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

#define MAX 100

#define INF 1e9

int n, e, graph[MAX][MAX], path[MAX];

bool visited[MAX];

// Function to find the minimum cost using backtracking

void tsp(int curr\_pos, int count, int cost, int& ans) {

// Reached all cities, update the minimum cost and path

if (count == n && graph[curr\_pos][0]) {

ans = min(ans, cost + graph[curr\_pos][0]);

return;

}

for (int i = 0; i < n; i++) {

if (!visited[i] && graph[curr\_pos][i]) {

visited[i] = true;

path[count] = i;

tsp(i, count + 1, cost + graph[curr\_pos][i], ans);

visited[i] = false;

}

}

}

int main() {

cin >> n >> e;

memset(graph, INF, sizeof(graph));

memset(visited, false, sizeof(visited));

// Input the roads and their costs

for (int i = 0; i < e; i++) {

int u, v, w;

cin >> u >> v >> w;

graph[u][v] = w;

graph[v][u] = w;

}

visited[0] = true;

path[0] = 0;

int ans = INF;

tsp(0, 1, 0, ans);

cout << "Minimum cost: " << ans << endl;

cout << "Path: ";

for (int i = 0; i < n; i++) {

cout << path[i] << " ";

}

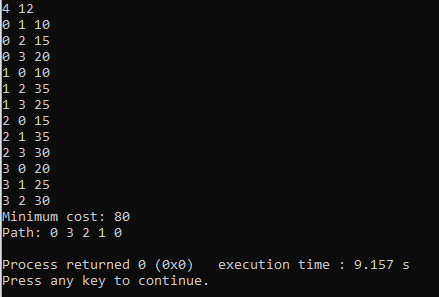
cout << path[0] << " ";

cout << endl;

return 0;

}

**Output:**

****

**Applications:**

The TSP has several applications even in its purest formulation, such as planning, logistics, the manufacture of microchips, DNA Sequencing etc. It also helps to find the most efficient route for data to travel between various nodes.

**Problem 02: Hamiltonian Cycle**

Solution:

**TAG:** Graph

**COMPLEXITY:**  *O*(*V*!), where *V* is the number of vertices.

* Time Complexity: The worst-case time complexity is *O*(*V*!), where *V* is the number of vertices. This is because, in the worst case, the algorithm may need to consider all permutations of the vertices
* Space Complexity: The space complexity is *O*(*V*) for the “visited” and “path” vectors. Where *V* is the number of vertices in the graph.

To solve the problem, we can use the backtracking approach. We start with an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

bool hamiltonianCycle(vector<vector<int>>& adj, vector<bool> visited, vector<int>& path, int v,int V)

{

if (path.size() == V)

return true;

for (auto adjnode : adj[v])

{ //check adjnodes are already visited or not

if (visited[adjnode] == 0)

{

visited[adjnode] = 1;

path.push\_back(adjnode);

//recursive call

if (hamiltonianCycle(adj, visited, path, adjnode,V))

return true;

//backtrack

visited[adjnode] = 0;

path.pop\_back();

}

}

return false;

}

int main()

{

int V,E;

cin>>V>>E;

vector<vector<int>> adj(V+1,vector<int>(V+1,0));

//adj list

for(int i=0;i<E;i++){

int u,v;cin>>u>>v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

vector<bool> visited(V+1, 0);

vector<int> path;

path.push\_back(0);

visited[0] = 1;

if (hamiltonianCycle(adj, visited, path, 0,V))

{

int last = path.back();

vector<int>& v = adj[path[0]];

// check if first and last vertex are connected

if (find(v.begin(), v.end(), last) == v.end())

cout << "NO" << endl;

else

{ cout<<"YES"<<endl;

cout << "Hamiltonian Cycle: ";

for (int i : path)

cout << i << " ";

cout << path[0] << endl;

}

}

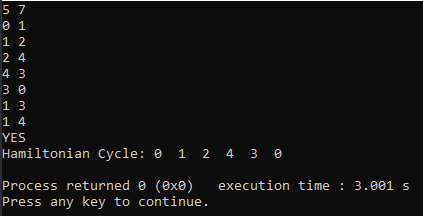
else

cout << "NO" << endl;

return 0;

}

**Output:**

****

**Applications:**

**Network Design**: In designing network connections where each node needs to be visited exactly once. A Hamiltonian Cycle can help ensure that each component is connected exactly once.

**Problem 03: Graph Coloring Problem**

Solution:

**TAG:** DFS, Greedy algorithm.

**COMPLEXITY:**   O(m^n), where n is the number of nodes in the graph.

* Time Complexity : In the worst case, the solve function iterates through m colors, leading to a time complexity of O(m).Recursive calls for each node lead to a branching factor of O(m^n), where n is the number of nodes. Considering both factors, the time complexity of the solve function is O(m^n).
* Space complexity: The adjacency matrix takes O(N^2) space as it represents connections between all pairs of nodes.

To solve the problem, we can use the backtracking approach. We start by assigning the first color to the first vertex. For each of the remaining vertices, we try to assign a color that has not been assigned to any of its neighbors. If we cannot assign a color to a vertex, we backtrack and try a different color for the previous vertex.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

bool isSafe(int node, vector<int>& color, vector<vector<int>>& adj, int n, int col) {

for (int k = 0; k < n; k++) {

if (k != node && adj[k][node] == 1 && color[k] == col) {

return false;

}

}

return true;

}

bool solve(int node, vector<int>& color, int m, int N, vector<vector<int>>& adj) {

if (node == N) {

return true;

}

for (int i = 0; i < m; i++) {

if (isSafe(node, color, adj, N, i)) {

color[node] = i;

if (solve(node + 1, color, m, N, adj)) return true;

color[node] = -1;

}

}

return false;

}

int main() {

int N, E, m;

cin >> N >> E >> m;

vector<vector<int>> adj(N + 1, vector<int>(N + 1, 0));

for (int i = 0; i < E; i++) {

int u, v;

cin >> u >> v;

adj[u][v] = 1;

adj[v][u] = 1;

}

vector<int> color(N + 1, -1);

if (solve(0, color, m, N, adj)) {

cout << "YES" << endl;

} else {

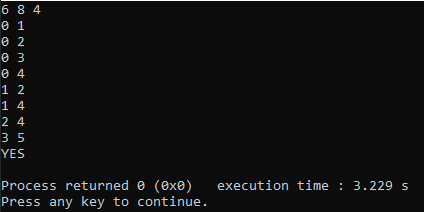
cout << "NO" << endl;

}

return 0;

}

**Output:**

****

**Applications:**

**Spectrum Management:** In scenarios where the available frequency spectrum is limited, such as in crowded urban areas, effective graph coloring algorithms contribute to better spectrum utilization. Regulatory bodies use such techniques for spectrum allocation and management.

**Problem 04: N-Queen Problem**

Solution:

**TAG:** Backtracking.

**COMPLEXITY:**   O(N!), where N is the size of the chessboard.

* **Time Complexity:** The time complexity of the N-Queens problem using backtracking is generally considered to be exponential. In the worst case, the algorithm explores all possible configurations of placing queens on the board. For each cell, the algorithm checks whether placing a queen in that cell is safe by examining the previous rows. The number of recursive calls is on the order of O(N!), where N is the size of the chessboard (number of rows or columns). Hence, the time complexity is O(N!).
* **Space Complexity:** The board vector is a 2D vector of strings representing the chessboard. Its size is N x N. Therefore, the space complexity for the chessboard representation is O(N^2).Result Vector(ans vector):The ans vector stores all the valid arrangements of queens on the chessboard. In the worst case, the number of valid arrangements is O(N!). Each arrangement is a 2D vector of strings.Therefore, the space complexity for the result vector is O(N! \* N^2).

The algorithm uses a recursive approach with backtracking to explore all possible queen placements on the chessboard. It starts by placing queens row by row, ensuring that each placement is safe by checking the current column, and both diagonals. If a safe placement is found, the algorithm proceeds to the next row. If no safe placement is found, it backtracks to the previous row and explores other possibilities.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

// Function to check if placing a queen at a given position is safe

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

for (int i = 0; i < row; i++) {

// Check the column

if (board[i][col] == '1') {

return false;

}

// Check diagonals

if (col - (row - i) >= 0 && board[i][col - (row - i)] == '1') {

return false;

}

if (col + (row - i) < board.size() && board[i][col + (row - i)] == '1') {

return false;

}

}

return true;

}

// Recursive function to solve the N-Queens problem using backtracking

void solve(int row, vector<string>& board, vector<vector<string>>& ans) {

// If all queens are placed without being threaten, add the current arrangement to the result

if (row == board.size()) {

ans.push\_back(board);

return;

}

// Try placing a queen in each column of the current row

for (int col = 0; col < board.size(); col++) {

if (isSafe(row, col, board)) {

// Place the queen and move on to the next row

board[row][col] = '1';

solve(row + 1, board, ans);

//undo the placement for backtracking

board[row][col] = '0';

}

}

}

// Function to solve the N-Queens problem and return all possible arrangements

vector<vector<string>> solveNQueens(int n) {

vector<vector<string>> ans;

// Initialize an empty chessboard

vector<string> board(n, string(n, '0'));

// Start solving from the first row

solve(0, board, ans);

return ans;

}

int main() {

int n ;

cin>>n; // Set the size of the chessboard and the number of queens

vector<vector<string>> ans = solveNQueens(n);

// Print the result

for (int i = 0; i < ans.size(); i++) {

cout << "Arrangement " << i + 1 << "\n";

for (int j = 0; j < ans[i].size(); j++) {

cout << ans[i][j] << endl;

}

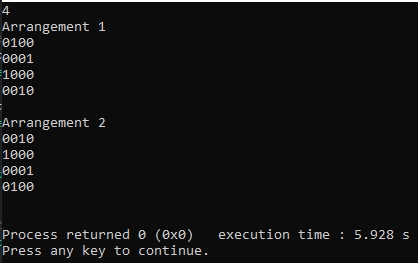
cout << endl;

}

return 0;

}

**Output:**

****

**Applications:**

**Robotics**: For robotic path planning, the N-Queens problem can be used to model obstacle avoidance, ensuring that no two components interfere with each other and optimal path selection

**Problem 05: Optimal Polygon Triangulation**

Solution:

**TAG:** Dynamic Programming.

**COMPLEXITY:** O(N^3), where N is the number of vertices in the polygon.

* The time complexity of the algorithm is O(N^3), where N is the number of vertices in the polygon.
* The space complexity of the algorithm is also O(N^2), which is the size of the memorization table.

dist(a, b):  This function calculates the Euclidean distance between two points.

trianglePerimeter(a, b, c): This function calculates the perimeter of a triangle formed by three points.

cost(i, j, coordinates, dp): This recursive function calculates the minimum cost of triangulation for the subpolygon formed by vertices from i to j. It considers all possible cuts between i and j, calculates the cost of the triangle formed by each cut, and recursively calculates the cost of the two resulting subpolygons. The memorization table (dp) is used to store already computed results to avoid redundant calculations.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

// Function to calculate the distance between two points

double dist(pair<int, int> a, pair<int, int> b) {

return sqrt(pow(a.first - b.first, 2) + pow(a.second - b.second, 2));

}

// Function to calculate the perimeter of a triangle formed by three points

double trianglePerimeter(pair<int, int> a, pair<int, int> b, pair<int, int> c) {

return dist(a, b) + dist(b, c) + dist(c, a);

}

// Function to calculate the minimum cost of triangulation for a subpolygon

double cost(int i, int j, vector<pair<int, int>>& coordinates, vector<vector<double>>& dp) {

if (j - i < 2) {

return 0;

}

if (dp[i][j] != -1) {

return dp[i][j];

}

double min\_cost = 1e9;

for (int k = i + 1; k < j; k++) {

double triangle\_cost = trianglePerimeter(coordinates[i], coordinates[k], coordinates[j]);

double subpolygon\_cost = cost(i, k, coordinates, dp) + cost(k, j, coordinates, dp);

double total\_cost = triangle\_cost + subpolygon\_cost;

min\_cost = min(min\_cost, total\_cost);

}

dp[i][j] = min\_cost;

return min\_cost;

}

int main() {

int N;

cin >> N;

// Function to find the optimal polygon triangulation

vector<pair<int, int>> coordinates(N);

for (int i = 0; i < N; i++) {

cin >> coordinates[i].first >> coordinates[i].second;

}

vector<vector<double>> dp(N, vector<double>(N, -1));

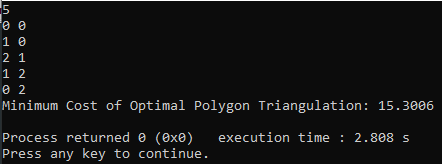
double min\_cost = cost(0, N - 1, coordinates, dp);

cout << "Minimum Cost of Optimal Polygon Triangulation: " << min\_cost << endl;

return 0;

}

**Output:**

****

**Applications:**

**Network Design**: In network design, especially in wireless communication, triangulation algorithms can be applied to optimize the placement of network nodes for better coverage and connectivity.

**Artificial Intelligence and Machine Learning**: Geometric algorithms, including polygon triangulation, can be used as part of data preprocessing in machine learning tasks related to spatial data analysis and pattern recognition.

**Problem 06: Optimal Binary search tree**

Solution:

**TAG:** Dynamic programming, Implementation

**COMPLEXITY:** O(N^3), where N is the length of the given input.

* The outer loop runs for each subarray length (len), which takes O(n) time.
* The inner loops iterate over all possible subarrays, taking O(n) time in the worst case.
* Inside the innermost loop, the sum of frequencies is computed, which also takes O(n) time.
* Therefore, the overall time complexity is O(n^3).
* Space complexity O(n^2)
* The dp table has a size of n x n, resulting in O(n^2) space complexity.

First of all, what is a binary tree? Basically every node of a tree has two children that is why it's called a binary tree. Then in the binary search tree there is a searching cost of an element. And that cost depends on the level of that node. If an element in a root node then its searching cost is 0, if it's were level 1 node then the cost would be 1.In the given problem you have some element and the search frequency (number of times you need to search that element). And by that priority we are going to build a binary search tree. And there we are going to use a dynamic programming approach where dp[i][j] will tell us the minimum cost of searching the element from i to j.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

int main() {

int n;

cin >> n;

int keys[n], freq[n];

for (int i = 0; i < n; i++) {

cin >> keys[i];

}

for (int i = 0; i < n; i++) {

cin >> freq[i];

}

int dp[n][n];

memset(dp, 0, sizeof(dp));

for (int i = 0; i < n; i++) {

dp[i][i] = freq[i];

}

for (int len = 2; len <= n; len++) {

for (int i = 0; i <= n - len; i++) {

int j = i + len - 1;

dp[i][j] = INT\_MAX;

int sum = 0;

for (int k = i; k <= j; k++) {

sum += freq[k];

}

for (int k = i; k <= j; k++) {

int val = sum + (k > i ? dp[i][k - 1] : 0) + (k < j ? dp[k + 1][j] : 0);

dp[i][j] = min(dp[i][j], val);

}

}

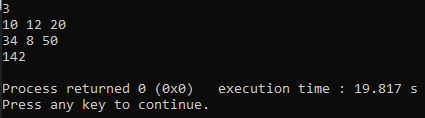
}

cout << dp[0][n - 1] << endl;

return 0;

}

**Output:**

****

**Applications:**

Utilized in priority queues, data caching, spell checking, and dictionary implementation are a few real-world uses for BTS.

**Problem 07: Johnson’s Algorithm**

Solution:

**TAG:** Greedy Algorithm, DP, Graph

**COMPLEXITY:** O(N^2 log N + N \* E), where N is the number of nodes and E is the number of edges in the graph.

Johnson’s Algorithm is used to find the shortest paths between all pairs of vertices in a given weighted directed graph where weights may be negative as well. The algorithm works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra’s algorithm to be used on the transformed graph.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

#define INF INT\_MAX

struct Edge {

int u, v, weight;

};

vector<int> BellmanFord(int V, int src, vector<Edge>& edges) {

vector<int> dist(V, INF);

dist[src] = 0;

for (int i = 0; i < V - 1; i++) {

for (Edge& edge : edges) {

int u = edge.u;

int v = edge.v;

int weight = edge.weight;

if (dist[u] != INF && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

for (Edge& edge : edges) {

int u = edge.u;

int v = edge.v;

int weight = edge.weight;

if (dist[u] != INF && dist[u] + weight < dist[v]) {

cout << "Negative cycle exists" << endl;

exit(0);

}

}

return dist;

}

void Dijkstra(int V, vector<pair<int, int>> adj[], int src, vector<int>& dist) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({0, src});

dist[src] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto& i : adj[u]) {

int v = i.first;

int weight = i.second;

if (dist[v] > dist[u] + weight) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

}

void Johnson(int V, vector<Edge>& edges) {

vector<Edge> new\_edges;

for (int i = 0; i < V; i++) {

new\_edges.push\_back({V, i, 0});

}

new\_edges.insert(new\_edges.end(), edges.begin(), edges.end());

vector<int> p = BellmanFord(V + 1, V, new\_edges);

vector<pair<int, int>> adj[V];

for (Edge& edge : edges) {

adj[edge.u].push\_back({edge.v, edge.weight + p[edge.u] - p[edge.v]});

}

for (int i = 0; i < V; i++) {

vector<int> dist(V, INF);

Dijkstra(V, adj, i, dist);

for (int j = 0; j < V; j++) {

if (dist[j] != INF) {

dist[j] += p[j] - p[i];

}

}

for (int j = 0; j < V; j++) {

if (dist[j] != INF) {

cout << i <<" " << j << " "<< dist[j] << endl;

}

}

}

}

int main() {

int V,E;

cin>>V>>E;

vector<Edge> edges(E);

for (int i = 0; i < E; i++) {

cin >> edges[i].u >> edges[i].v >> edges[i].weight;

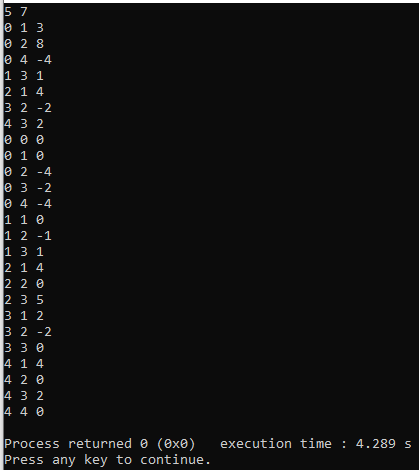
}

Johnson(V, edges);

return 0;

}

**Output:**

****

**Applications:**

Johnson's algorithm is one approach that can often be applied to minimize the total elapsed time between the start of a job on the first machine and the completion of a job on another machine. The heuristic algorithm can be implemented in job scheduling.

**Problem 08: Sequence Alignment Problem**

Solution:

**TAG:** Dynamic Programming

**COMPLEXITY:** O(mn), where m and n are the lengths of the two strings.

* The space complexity of the algorithm is also O(mn), which is the size of the matrix.

The Sequence Alignment Problem is solved using Dynamic Programming. We can create a matrix of size (m+1) x (n+1), where m and n are the lengths of the two strings. We can then fill in the matrix using the following recurrence relation:

1. If i = 0 or j = 0, then dp[i][j] = i \* p\_gap + j \* p\_gap
2. Otherwise, dp[i][j] = min(dp[i-1][j-1] + p\_xy, dp[i-1][j] + p\_gap, dp[i][j-1] + p\_gap)

The minimum penalty will be stored in dp[m][n]. We can then backtrack through the matrix to find the aligned strings.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

vector<vector<int>> dp(1000+1, vector<int>(1000+1, 0));

pair<string, string> sequence\_alignment(string X, string Y, int p\_gap, int p\_xy) {

int m = X.size();

int n = Y.size();

// Initialize the first row and column of the matrix

for(int i=0; i<=m; i++) {

for(int j=0; j<=n; j++) {

if(i == 0) dp[i][j] = j \* p\_gap;

else if(j == 0) dp[i][j] = i \* p\_gap;

else if(X[i-1] == Y[j-1]) dp[i][j] = dp[i-1][j-1];

else dp[i][j] = min({dp[i-1][j-1] + p\_xy, dp[i-1][j] + p\_gap, dp[i][j-1] + p\_gap});

}

}

// Backtrack through the matrix to find the aligned strings

string align\_X = "", align\_Y = "";

int i = m, j = n;

while(i > 0 && j > 0) {

if(X[i-1] == Y[j-1]) {

align\_X = X[i-1] + align\_X;

align\_Y = Y[j-1] + align\_Y;

i--; j--;

} else if(dp[i][j] == dp[i-1][j-1] + p\_xy) {

align\_X = X[i-1] + align\_X;

align\_Y = Y[j-1] + align\_Y;

i--; j--;

} else if(dp[i][j] == dp[i-1][j] + p\_gap) {

align\_X = X[i-1] + align\_X;

align\_Y = "\_" + align\_Y;

i--;

} else {

align\_X = "\_" + align\_X;

align\_Y = Y[j-1] + align\_Y;

j--;

}

}

while(i > 0) {

align\_X = X[i-1] + align\_X;

align\_Y = "\_" + align\_Y;

i--;

}

while(j > 0) {

align\_X = "\_" + align\_X;

align\_Y = Y[j-1] + align\_Y;

j--;

}

return {align\_X, align\_Y};

}

int main() {

string X,Y ;

cin>>X>>Y;

int p\_gap, p\_xy;

cin>>p\_gap>>p\_xy;

pair<string, string> result = sequence\_alignment(X, Y, p\_gap, p\_xy);

// Output the result

cout << "Minimum penalty aligning two problem is: " << dp[X.size()][Y.size()] << endl;

cout << "Aligned as: " << endl;

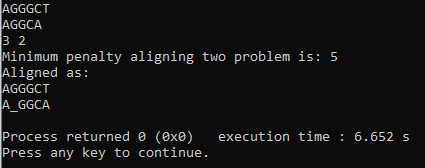
cout << result.first << endl;

cout << result.second << endl;

return 0;

}

**Output:**

****

**Applications:**

Sequence alignments are useful in bioinformatics for identifying sequence similarity, producing phylogenetic trees, and developing homology models of protein structures.

**Problem 09: Clique Problem**

Solution:

**TAG:** Greedy

**COMPLEXITY:** O(V+E\*k), Assuming clique size=K, Number of vertex =V, Number of edge=E.

* The worst-case time complexity is exponential due to the recursive exploration of all possible subsets of vertices.
* The Bron–Kerbosch algorithm has a time complexity of O(3^(n/3)), where n is the number of vertices. In practice, it may perform better on sparse graphs.
* Each recursive call involves a loop through the candidate set, and in the worst case, the recursion tree has an exponential number of nodes.
* The space complexity is O(n), where n is the number of vertices, due to the space required for storing the graph and the recursive call stack.

To solve this problem, in our algorithm, we have calculated degrees of all the nodes and kept in array. The main logic is we will check all the vertices having degree equal or greater than to clique\_size. After finding the vertex we will check if the following vertices form a subgraph or not. We took three parameters in clique function which are start, value, clique size. Here start parameter represents the starting node from which we will start iterating, value represents the number of nodes are in the keep array and third parameter is represents the size of the click we want to find. Then we will start checking from the starting node if the degree is greater or equal to clique\_size - 1. If condition satisfies, we store the node in keep array. After storing the nodes we check if those nodes form a clique or not. If those make a clique then we check the size of clique is equal clique\_size or not. If it forms a clique and the number of vertices in the store is equal to the required size of the clique then we store the clique in a vector caller store\_clique. If it doesn’t form a clique and the number of elements in the store array is less than clique\_size, then we call the function clique recursively and increase the number of elements in the store.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

void extend(vector<vector<int>>& cliques, vector<int>& R, vector<int>& P, vector<int>& X, const vector<vector<int>>& graph, int size) {

if (P.empty() && X.empty() && R.size() == size) {

cliques.push\_back(R);

return;

}

while (!P.empty()) {

int v = P.back();

P.pop\_back();

R.push\_back(v);

vector<int> P\_new, X\_new;

for (int u : P)

if (graph[v][u])

P\_new.push\_back(u);

for (int u : X)

if (graph[v][u])

X\_new.push\_back(u);

extend(cliques, R, P\_new, X\_new, graph, size);

R.pop\_back();

X.push\_back(v);

}

}

int main() {

int N, E, K;

cin >> N >> E;

vector<vector<int>> graph(N, vector<int>(N, 0));

for (int i = 0; i < E; i++) {

int u, v;

cin >> u >> v;

graph[u][v] = graph[v][u] = 1;

}

cin >> K;

vector<vector<int>> cliques;

vector<int> R, P(N), X;

for (int i = 0; i < N; i++)

P[i] = i;

extend(cliques, R, P, X, graph, K);

for (auto& clique : cliques) {

sort(clique.begin(), clique.end());

for (int v : clique)

cout << v << ' ';

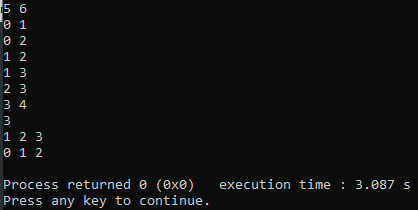
cout << '\n';

}

return 0;

}

**Output:**

****

**Applications:**

The maximum clique problem (MCP) is an important combinatorial optimization problem with a wide range of practical applications in numerous fields, including information retrieval, signal transmission analysis, classification theory, economics, scheduling, and biomedical engineering.

**Problem 10: Strongly Connected Graph (SCC) to Direct Acyclic Graph (DAG)**

Solution:

**TAG:** DFS

**COMPLEXITY:**  O(N + E), where N is the number of nodes and E is the number of edges in the graph.

* The time complexity of the algorithm is O(N + E), where N is the number of nodes and E is the number of edges in the graph.
* The space complexity of the algorithm is also O(N + E), which is the size of the graph.

The problem requires finding the strongly connected components (SCC) of a directed graph and constructing the directed acyclic graph (DAG) found from this given graph. To solve this problem, we can use Kosaraju’s algorithm. The algorithm consists of two phases:

1. In the first phase, we perform a depth-first search (DFS) on the graph and store the vertices in the order in which they finish.
2. In the second phase, we perform another DFS on the transpose of the graph (i.e., the graph with all edges reversed) in the order given by the first phase. Each tree in the DFS forest of the second phase corresponds to a strongly connected component.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

// Function to perform depth-first search

void dfs(int node, vector<int>& vis, vector<int>& store, vector<vector<int>>& adj) {

vis[node] = 1;

for(auto& i : adj[node]) {

if(!vis[i]) {

dfs(i, vis, store, adj);

}

}

store.push\_back(node);

}

int main() {

int n, e;

cin >> n >> e;

// Create adjacency lists for the graph and its transpose

vector<vector<int>> adj(n+1), tadj(n+1);

// Read the edges

for(int i = 0; i < e; i++) {

int u, v;

cin >> u >> v;

adj[u].push\_back(v);

tadj[v].push\_back(u); // Transpose of the graph

}

vector<int> vis(n+1, 0), order, SCC;

// Perform DFS on the original graph to get the vertices in topological order

for(int i = 1; i <= n; i++) {

if(!vis[i]) {

dfs(i, vis, order, adj);

}

}

// Reset the visited array for the second DFS

vis.assign(n+1, 0);

// Perform DFS on the transposed graph in the order given by the first DFS

reverse(order.begin(), order.end());

int k=1;

for(auto& node : order) {

if(!vis[node]) {

SCC.clear();

dfs(node, vis, SCC, tadj);

cout<<k<<" : ";

for(auto& vertex : SCC) {

cout << vertex << " ";

}

cout << "\n";

k++;

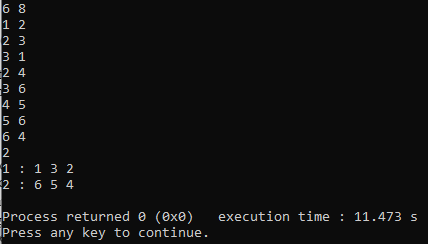
}

}

return 0;

}

**Output:**

****

**Applications:**

In project management, tasks are often represented as nodes in a graph, and dependencies between tasks are modeled as edges. The tasks have precedence relationships, and creating a Directed Acyclic Graph (DAG) from a Strongly Connected Graph (SCG) can be beneficial.

**Problem 11: Multiple Shortest Path Printing**

Solution:

**TAG:** DFS, SHORTEST PATH

**COMPLEXITY:** O(E log V), where E is the number of edges and V is the number of vertices in the graph.

* The time complexity of the algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph. The space complexity of the algorithm is O(V log V), which is the size of the priority queue.

This problem can be solved using a modified version of Dijkstra’s algorithm. In the algorithm of Dijkstra, we use D[v] to denote a node’s distance from the source. Here, we can use a two-dimensional array to solve this problem (D[v][p]), where the array will hold the shortest distance of v from the source in path number p. In a similar fashion, we can also save multiple paths’ parent nodes information.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

#define MAX 100005

#define INF 1e9

vector<pair<int,int>> adj[MAX];

int n,m,s,t;

int dist[MAX],cnt[MAX];

bool vis[MAX];

vector<int> path[MAX];

void dijkstra(){

priority\_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;

pq.push({0,s});

dist[s]=0;

cnt[s]=1;

while(!pq.empty()){

int u=pq.top().second;

pq.pop();

if(vis[u]) continue;

vis[u]=true;

for(auto v:adj[u]){

if(dist[v.first]>dist[u]+v.second){

dist[v.first]=dist[u]+v.second;

cnt[v.first]=cnt[u];

pq.push({dist[v.first],v.first});

path[v.first].clear();

path[v.first].push\_back(u);

}

else if(dist[v.first]==dist[u]+v.second){

cnt[v.first]+=cnt[u];

path[v.first].push\_back(u);

}

}

}

}

int main(){

cin>>n>>m;

for(int i=1;i<=m;i++){

int u,v,w;

cin>>u>>v>>w;

adj[u].push\_back({v,w});

}

cin>>s>>t;

memset(dist,0x3f,sizeof(dist));

dijkstra();

// Output the minimum distance

cout<<"The minimum distance between "<<s<<" and "<<t<<" is "<<dist[t]<<endl;

// Output the number of shortest paths

cout<<"The number of shortest paths between "<<s<<" and "<<t<<" is "<<cnt[t]<<endl;

// Output the intermediate nodes

cout<<"The intermediate nodes to reach "<<t<<" from "<<s<<" are: ";

queue<vector<int>> q;

q.push({t});

while(!q.empty()){

vector<int> cur=q.front();

q.pop();

int last=cur[0];

if(last==s){

for(int i=cur.size()-1;i>=0;i--){

cout<<cur[i]<<" ";

}

cout<<endl;

}

else{

for(auto i:path[last]){

vector<int> tmp=cur;

tmp.insert(tmp.begin(),i);

q.push(tmp);

}

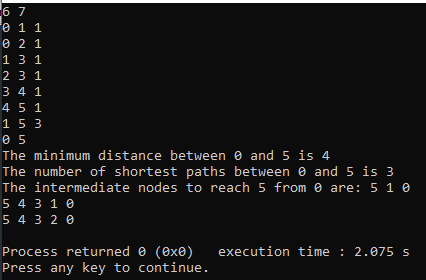
}

}

return 0;

}

**Output:**

****

**Applications:**

In GPS navigation, multiple shortest path printing is useful when providing alternative routes to a destination. Users can be presented with different routes based on factors like traffic conditions, road closures, or user preferences.

**Problem 12: Maximum Sum Interval**

Solution:

**TAG:** GREEDY

**COMPLEXITY:** O(n) or linear.

* Time Complexity O(n).
* The algorithm iterates through each element of the array exactly once, performing constant-time operations at each step.
* The time complexity is linear with respect to the size of the input array.
* The algorithm uses a constant amount of additional space, regardless of the size of the input array.
* The space complexity is constant.

This is a pretty straightforward solution. First of all we will have a sum variable and we will calculate the sum of adjacent elements whenever we get a negative sum we will update our sum variable with 0 because if we don't get a positive sum we will not take any elements from the given array. And this algorithm is called KADANE'S algorithm.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin>>n;

vector<int> a(n);

for(int i=0; i<n; i++)

{

cin>>a[i];

}

long long cur = 0, ans = 0, min=0;

for(int i=0; i<n; i++)

{

cur += a[i];

cur = max(cur, min);

ans = max(ans, cur);

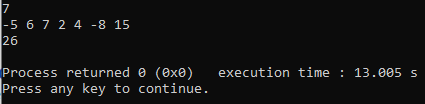
}

cout<<ans<<"\n";

return 0;

}

**Output:**



**Applications:**

The real-life application of Maximum Sum Interval, which involves finding the interval with the maximum sum in a sequence, is frequently encountered in Finance and Stock Market Analysis.

In finance, the maximum sum interval is used to identify the most profitable time periods for buying and selling stocks.

Traders and investors analyze historical stock prices to find intervals with the maximum cumulative returns.

This information helps in making informed decisions about when to enter or exit the market to maximize profits or minimize losses.

**Problem 13: Matrix Chain Multiplication**

Solution:

**TAG:** DP

**COMPLEXITY:** O(n^3), where n is the number of matrices in the sequence.

* The Time Complexity: O(n^3)
* The dominant factor is the triple nested loop in the matrixChainOrder function, where the outer loop runs for each length of the subchain, and the inner loops iterate over the subchain and the split point.
* The overall time complexity is cubic in terms of the number of matrices (n).
* Space Complexity: O(n^2)
* The space complexity is determined by the matrices m and s, both of size n x n
* It is quadratic in terms of the number of matrices (n).

The algorithm used to solve the Matrix Chain Multiplication problem is Dynamic Programming (DP). The following steps are mentioned below:

1. Create a table m[][] to store the minimum number of multiplications needed to compute the matrix chain product. The table should be filled in a manner similar to the memorized recursive approach.
2. Create a table s[][] to store the index of the matrix after which the split is made.
3. Print the association order of how the matrix is multiplied, e.g, ((AB)C), (A(BC)), etc.

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

void printParenthesis(int i, int j, int n, int\* s, char& name)

{

if (i == j) {

cout << name++;

return;

}

cout << "(";

printParenthesis(i, \*((s + i \* n) + j), n, s, name);

printParenthesis(\*((s + i \* n) + j) + 1, j, n, s, name);

cout << ")";

}

void matrixChainOrder(int p[], int n)

{

int m[n][n];

int s[n][n];

for (int i = 1; i < n; i++)

m[i][i] = 0;

for (int L = 2; L < n; L++) {

for (int i = 1; i < n - L + 1; i++) {

int j = i + L - 1;

m[i][j] = INT\_MAX;

for (int k = i; k <= j - 1; k++) {

int q = m[i][k] + m[k + 1][j]

+ p[i - 1] \* p[k] \* p[j];

if (q < m[i][j]) {

m[i][j] = q;

s[i][j] = k;

}

}

}

}

char name = 'A';

cout << "The optimal parenthesization is : ";

printParenthesis(1, n - 1, n, (int\*)s, name);

cout << "\nMinimum number of multiplications is : " << m[1][n - 1];

}

int main()

{

int n;

cin>>n;

int arr[n];

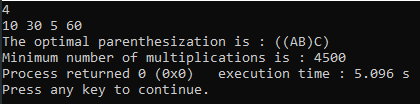
for(int i=0;i<n;i++)cin>>arr[i];

matrixChainOrder(arr, n);

return 0;

}

**Output:**



**Applications:**

Matrix multiplication has numerous real-time applications across various fields. Some of the key applications include: Computer Graphics: In computer graphics, matrices are used to represent transformations like rotation, translation, scaling, and shearing

**Problem 14: Articulation Point and Bridges**

Solution:

**TAG:** DFS, Implementation

**COMPLEXITY:** O(E+N) here E is the number of edges and N is the number of nodes.

* The time complexity is O(N + E), where V is the number of vertices, and E is the number of edges. This complexity arises from the DFS traversal and the linear-time checks for articulation points and bridges.
* The space complexity is O(N + E), primarily due to the adjacency list representation of the graph (O(N+ E)) and additional arrays (O(N)) used during DFS.

In this problem we are not given a source node. But we are gonna assume node 1 is a source node here. By taking 1 as a source node we are gonna run a dfs from node 1 now if we are going to visit some node y from some node x. Then if the node y is not previously and its depth is greater than the node x. Then we can say node x is an articulation point. Because any child of x is not connected with any child of y.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

void dfs(int u, int p, vector<int>& disc, vector<int>& low, vector<int>& ap, vector<array<int, 2>>& br, vector<vector<int>>& adj, int& Time) {

    int children = 0;

    low[u] = disc[u] = ++Time;

    for (auto v : adj[u]) {

        if (v == p) continue;

        if (!disc[v]) {

            children++;

            dfs(v, u, disc, low, ap, br, adj, Time);

            if(disc[u] < low[v]) br.push\_back({u, v});

            if (disc[u] <= low[v]) ap[u] = 1;

            low[u] = min(low[u], low[v]);

        }

        else low[u] = min(low[u], disc[v]);

    }

    if(p == u && children > 1) ap[u] = 1;

}

int main() {

    int n, e;

    cin >> n >> e;

    vector<vector<int>> adj(n+1);

    for(int i = 1; i <= e; i++) {

        int u, v;

        cin >> u >> v;

        adj[u].push\_back(v);

        adj[v].push\_back(u);

    }

    vector<int> disc(n+1, 0), ap(n+1, 0), low(n+1, INT\_MAX);

    vector<array<int, 2>> br;

    int Time = 0;

    for (int u = 1; u <= n; u++)

        if (!disc[u])

            dfs(u, u, disc, low, ap, br, adj, Time);

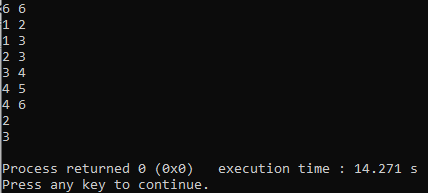
    cout << accumulate(ap.begin(), ap.end(), 0ll) << "\n";

    cout << br.size() << "\n";

    return 0;

}

**Output:**



**Applications:**

Real life application of Articulation Points:

In a communication network, such as the internet, an articulation point represents a critical node whose removal can disrupt the entire network. Identifying articulation points is crucial for designing resilient and fault-tolerant networks. For example, in a telecommunications network, an articulation point could be a central hub or a key server. Recognizing and strengthening these points ensures the network's robustness.

Real life application of Bridges:

Bridges in a network are edges whose removal can potentially disconnect the network. In the context of computer networks, bridges might represent crucial communication links. Recognizing and securing these bridges helps prevent network breakdowns and ensures reliable communication pathways.

**Problem 15: Closest Pair of Points**

Solution:

**TAG:** BRUTEFORCE

**COMPLEXITY:**  O(n log n)

* Time Complexity: Sorting the points initially: O(n \* log(n)), where n is the number of points. Sorting the strip: O(n \* log(n)).Calculating distances in the strip: O(n).
* The recursive calls use O(log(n)) stack space, the strip vector requires O(n) space, and the points vector requires O(n) space. Therefore, the overall space complexity is O(n).

To solve the Closest Pair of Points problem using a divide and conquer algorithm, you can implement the following C++ code based on the concept of the "Closest Pair of Points" algorithm.

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

#define MAX 100005

#define INF 1e9

vector<pair<int,int>> points;

int n;

struct Compare {

bool operator()(pair<int,int> a, pair<int,int> b) {

return a.second < b.second;

}

};

float dist(pair<int,int> p1, pair<int,int> p2) {

return sqrt((p1.first - p2.first)\*(p1.first - p2.first) + (p1.second - p2.second)\*(p1.second - p2.second));

}

float min\_dp (int l, int r) {

float min\_dist = FLT\_MAX;

for (int i = l; i <= r; ++i)

for (int j = i+1; j <= r; ++j)

min\_dist = min(min\_dist, dist(points[i], points[j]));

return min\_dist;

}

float closestUtil (int l, int r) {

if (r-l+1 <= 3)

return min\_dp(l, r);

int mid = (l+r)/2;

float dl = closestUtil(l, mid);

float dr = closestUtil(mid+1, r);

float d = min(dl, dr);

vector<pair<int,int>> strip;

for (int i = l; i <= r; ++i){

if (abs(points[i].first - points[mid].first) < d)

strip.push\_back(points[i]);

}

//sort(strip.begin(), strip.end(), { return a.second < b.second;});

sort(strip.begin(), strip.end(), Compare());

float min\_strip = d;

for (int i = 0; i < strip.size(); ++i)

for (int j = i+1; j < strip.size() && (strip[j].second - strip[i].second) < min\_strip; ++j)

min\_strip = min(min\_strip, dist(strip[i], strip[j]));

return min(d, min\_strip);

}

float closestPair () {

sort(points.begin(), points.end());

return closestUtil(0, n-1);

}

int main(){

cin>>n;

for(int i=0;i<n;i++){

int x,y;

cin>>x>>y;

points.push\_back({x,y});

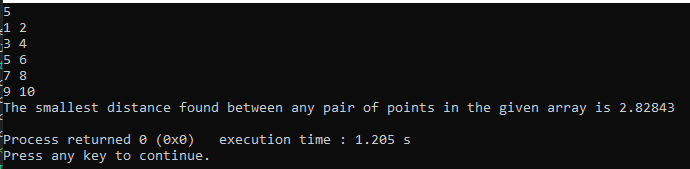
}

cout<<"The smallest distance found between any pair of points in the given array is "<<closestPair()<<endl;

return 0;

}

**Output:**



**Applications:**

In robotics and autonomous vehicle navigation, the Closest Pair of Points algorithm is applied to identify obstacles or objects in close proximity. This information helps robots and vehicles make real-time decisions to navigate safely and avoid collisions. For instance, in self-driving cars, identifying the closest pair of points can aid in adjusting the vehicle's path to avoid potential obstacles and ensure a smooth and secure journey.

**Problem 16: LCS(Longest Common Subsequences)**

Solution:

**TAG:** Dynamic Programming, Implementation

**COMPLEXITY:** O(N\*M) Where N is the length of the first string and M is the second string.

* The time complexity  is O(n \* m), where n is the length of string ‘a’  and m is the length of string ‘b’. This is because the code uses a nested loop that iterates through each pair of indices (i, j) in the strings, and the maximum number of iterations is proportional to the product of the lengths of the two strings.
* The space complexity is  O(n \* m) due to the creation of a 2D vector dp of size (n+1) x (m+1).

We are gonna use a tabulation form of dynamic programming approach.It's again a dynamic programming solution, so what is the state of our approach? The state is dp[i][j]. Where dp[i][j] will tell us from 0 to i of the first string and 0 to j of second string maximum length of common subsequences. For that such operation complexity will be O(N\*M).

**Source Code:**

#include<bits/stdc++.h>

using namespace std;

string s1,s2;

vector<vector<int>>DP(41, vector<int>(41, -1));

string lcs(int n, int m)

{

for (int j = 0; j <= m; j++)

{

DP[0][j] = 0;

}

for (int i = 0; i <= n; i++)

{

DP[i][0] = 0;

}

for(int i=1;i<=n;i++)

{

for(int j=1;j<=m;j++)

{

if(s1[i-1] == s2[j-1])

{

DP[i][j] = 1+DP[i-1][j-1];

}

else

{

DP[i][j] = max(DP[i-1][j], DP[i][j-1]);

}

}

}

int len = DP[n][m];

string ans = "";

for(int i=0; i<len; i++)

{

ans+='@';

}

int i=n, j=m;

int index = len-1;

while(i>0 && j>0)

{

if(s1[i-1] == s2[j-1])

{

ans[index] = s1[i-1];

index--;

i--;

j--;

}

else if(DP[i-1][j] > DP[i][j-1])

{

i--;

}

else

{

j--;

}

}

return ans;

}

int main()

{

cin>>s1>>s2;

int n = s1.size();

int m = s2.size();

string s = lcs(n,m);

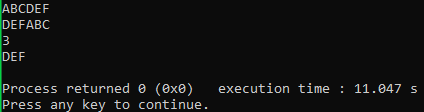
cout<<s.size()<<endl;

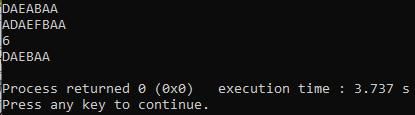
cout<<s<<endl;

return 0;

}

**Output:**





**Applications:**

In genomics, identifying the lexicographically smallest LCS is applied to analyze DNA sequences. The LCS helps researchers understand shared genetic information among different species or individuals. Finding the lexicographically smallest LCS is crucial for identifying the most conserved genetic elements, aiding in evolutionary studies and medical research related to genetic similarities and differences.