**B**  
B-8

Cần đảm bảo đồ thị không có chu trình trước.

B-9

#include <iostream>

#include <vector>

#include <stack>

#include <algorithm>

class Graph {

public:

Graph(int vertices) : V(vertices) {

graph.resize(V);

}

void addEdge(int u, int v) {

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

}

bool isConnected() {

std::vector<bool> visited(V, false);

dfs(0, visited);

return std::all\_of(visited.begin(), visited.end(), [](bool v) { return v; });

}

bool isEulerianCycle() {

if (!isConnected()) {

return false;

}

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

if (graph[v].size() != outDegree(v)) {

return false;

}

}

return true;

}

std::string eulerianCycle() {

if (!isEulerianCycle()) {

return "No Eulerian cycle exists in the graph.";

}

std::stack<int> eulerianPath;

std::vector<int> circuit;

eulerianPath.push(0);

while (!eulerianPath.empty()) {

int u = eulerianPath.top();

if (!graph[u].empty()) {

int v = graph[u].back();

graph[u].pop\_back();

eulerianPath.push(v);

} else {

circuit.push\_back(eulerianPath.top());

eulerianPath.pop();

}

}

std::reverse(circuit.begin(), circuit.end());

std::string result = "Eulerian cycle: ";

for (int vertex : circuit) {

result += std::to\_string(vertex) + " -> ";

}

result.pop\_back(); // Remove the last space

result.pop\_back(); // Remove the last arrow

return result;

}

private:

int V;

std::vector<std::vector<int>> graph;

void dfs(int v, std::vector<bool>& visited) {

visited[v] = true;

for (int neighbor : graph[v]) {

if (!visited[neighbor]) {

dfs(neighbor, visited);

}

}

}

int outDegree(int v) const {

return graph[v].size();

}

};

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(1, 3);

g.addEdge(3, 4);

g.addEdge(4, 1);

std::cout << g.eulerianCycle() << std::endl;

return 0;

}

B-10

Hãy mô tả một thuật toán thời gian tuyến tính tính thành phần liên thông mạnh chứa một đỉnh v cho trước:

B1: Dfs từ v, thêm các đỉnh được gặp vào 1 stack theo thứ tự  
B2: đổi Hướng của các cạnh từ đồ thị ban đầu, coi đây là đồ thị G2  
B3: Pop lần lượt các đỉnh trong stack, thực hiện Dfs từ đỉnh đó trên đồ thị G2, nếu được tập hơn chứa đỉnh v thì đó chính là đáp án.

Dựa trên thuật toán đó, hãy mô tả một thuật toán thời gian bậc hai đơn giản để tính các thành phần liên thông mạnh của một đồ thị có hướng.

1. **Khởi tạo:**
   * Tạo một stack để lưu trữ các đỉnh đã thăm.
   * Tạo một mảng boolean **visited** để theo dõi các đỉnh đã được thăm.
   * Khởi tạo stack.
2. **DFS (đối với tất cả các đỉnh):**
   * Duyệt qua tất cả các đỉnh của đồ thị.
   * Nếu đỉnh chưa được thăm, thực hiện DFS từ đỉnh đó và thêm các đỉnh vào stack sau khi hoàn thành DFS.
3. **Khởi tạo Đồ Thị Đảo:**
   * Tạo một đồ thị đảo bằng cách đảo hướng tất cả các cạnh của đồ thị ban đầu.
4. **DFS (đối với các đỉnh trên stack):**
   * Duyệt qua các đỉnh trong stack.
   * Nếu đỉnh chưa được thăm, thực hiện DFS từ đỉnh đó và xác định thành phần liên thông mạnh.

**Độ phức tạp O(V + E)**

B-11

Bước 1: sắp xếp topo, đưa thứ tự các cạnh vào 1 danh sách hoặc mảng.  
Bước 2: Duyệt lần lượt các đỉnh của danh sách. Với mỗi đỉnh, kiểm tra trong danh sách kề xem có đỉnh nào đứng liền sau danh sách xếp topo hay không, nếu có tiếp tục duyệt, nếu không thì đồ thị không có đường đi hamilton.

B-12  
Bước 1: sắp xếp topo, đưa thứ tự các cạnh vào 1 danh sách hoặc mảng.  
Bước 2: Duyệt lần lượt các đỉnh của danh sách. Với mỗi đỉnh, kiểm tra trong danh sách kề xem có đỉnh nào đứng liền sau danh sách xếp topo hay không, nếu có tiếp tục duyệt, nếu không thì đồ thị không có đường đi hamilton.  
Bước 3: Nếu đồ thị có đường đi hamilton chứng tỏ Đồ thị có duy nhất một thứ tự sắp xếp topo  
**Chứng Minh**: giả sử G(V, E) Dag có đường đi Hamilton.  
tồn tại đường đi p1,p2,…pn trong đó (v1, v2, v….vn) = V

Lưu ý rằng vi phải đến sau mỗi đỉnh v1, ..., vi−1 trong mọi thứ tự tô pô, vì chúng ta có đường đi [vj,⋯,vi] cho mỗi j nhỏ hơn i. Tương tự, vi+1,…,vn phải đến sau vi trong mọi thứ tự tô pô vì vi có đường đi đến mỗi đỉnh trong số đó.

Do đó, vi xuất hiện ở vị trí i trong mọi thứ tự tô pô, tức là mọi thứ tự tô pô đều bằng p.

B-13

Trường hợp cơ bản: Với V=1, chỉ có một đỉnh và không có cạnh. Do đó, |G(1)| = 2^0 = 1.

Giả sử với mọi k < V, |G(k)| = 2^(k choose 2)

Với V = k + 1, thêm 1 đỉnh vào đồ thị k đỉnh trước đó

Với mỗi trường hợp thêm 1 đỉnh trong G(k) trường hợp, lại có 2^k cách để nối vào đồ thị trước đó  
do đó khi thêm 1 đỉnh vào, ta có 2^(k choose 2) \* 2^k = 2 ^ (k +1 choose) cách, dpcm.

B-14

B-15

**#include <bits/stdc++.h>**

**using** **namespace** std;

**const** **int** maxN **=** 110;

**int** n, m;

**int** indegree[maxN], ans[maxN];

vector **<int>** g[maxN], topo;

queue **<int>** listSource;

main() {

cin **>>** n **>>** m;

**while** (m**--**) {

**int** u, v;

cin **>>** u **>>** v;

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

indegree[v]**++**;

}

**for** (**int** u **=** 1; u **<=** n; **++**u)

**if** (**!**indegree[u]) listSource.push(u);

**while** (**!**listSource.empty()) {

**int** u **=** listSource.front();

listSource.pop();

topo.push\_back(u);

**for** (**auto** v **:** g[u]) {

indegree[v]**--**;

**if** (**!**indegree[v]) listSource.push(v);

}

}

**if** (topo.size() **<** n) {

cout **<<** "Error: graph contains a cycle";

**return** 0;

}

**for** (**auto** x **:** topo) cout << x << “ “;

}