**// Simple DFS with vis for tracking visited :** **Call in main as dfs(1)**

**void dfs(int node){**

**vis[node] = true;**

**for(auto &i : edges[node])**

**{**

**//Do Something;**

**if(!vis[i])**

**dfs(i);**

**//Do Something Else;**

**}**

**}**

**// DFS with parent : Call in main as dfs(1,-1):**

**void dfs(int node, int par)**

**{**

**for(auto &i :edges[node])**

**{**

**//Do Something;**

**if(i != par)**

**dfs(i, node);**

**//Do Something Else;**

**}**

**}**

// Dijkstra - priority\_queue implementation

**Void Dijkstra(int s){**

**vector<int>dist(n + 5, 1e18);  
vector<bool>vis(n + 5, false);  
priority\_queue<pair<int, int>, vector<pair<int,int>>,greater<pair<int, int>>> pq;**

**pq.push({dist[s] = 0, s});  
while (!pq.empty()) {  
 int u = pq.top().second;  
 pq.pop();  
  
 if(vis[u])  
 continue;  
  
 vis[u] = true;  
 for (auto &child : edges[u]) {  
 if (dist[child.first] > dist[u] + child.second){  
 dist[child.first] = dist[u] + child.second;  
 pq.push({dist[child.first], child.first});**  
 }  
 }  
 }

**}**

**// shortest path Dijkstra cost and parent**

**#define ll long long**

**#define boost ios::sync\_with\_stdio(false);cin.tie(0)**

**ll const N = 1e5 + 5, mod=1e9 + 7;**

**vector<pair<int,int>>edges[N];**

**int n, m, a, b;**

**void shortest(int src){**

**set< pair<int, int> > st;**

**vector<int> dist(n + 1, 1e9), parent(n + 1, -1);**

**vector<bool>visted(n + 1, false);**

**st.insert({dist[src] = 0, src});**

**parent[src] = -1;**

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

**int par = st.begin() -> second;**

**st.erase(st.begin());**

**if(visted[par])**

**continue;**

**visted[par] = true;**

**for (auto &child : edges[par]) {**

**if (dist[child.first] > dist[par] + child.second) {**

**dist[child.first] = dist[par] + child.second;**

**st.insert({dist[child.first], child.first});**

**parent[child.first] = par;**

**}**

**}**

**}**

**//print path from src to b.**

**int k = b, min = INT\_MAX;**

**if(dist[b] == INT\_MAX)**

**cout << -1 << endl;**

**else{**

**while(parent[k] != a) {**

**cout << k << '\n';**

**k = parent[k];**

**}**

**cout << k << '\n';**

**}**

**}**

**int main()**

**{**

**boost;**

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

**for(int i = 0, u, v, w; i < m; i++) {**

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

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

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

**}**

**cin >> a >> b;**

**shortest(a);**

**}**

**//recursive, using depth**

**int depth[N];**

**void dfs(int u, int d) {**

**depth[u] = d;**

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

**if(depth[v] == -1) { // not visited yet**

**dfs(v, d+1);**

**}**

**}**

**}**

**// Finding connected components**

**int count\_cc() {**

**int count = 0;**

**memset(visited, 0, sizeof(bool)\*n);**

**rep(i,0,n) {**

**if (!visited[i]) {**

**count++, dfs(i);**

**}**

**}**

**return count;**

**}**

**// Bellmen Ford // complexity: O(|V||E|)**

**int const N = 1e5 + 5;**

**int n, m; // number of nodes, number of edges**

**vector<vector<int>>edges; //edge list**

**vector<int> dist; // store min distance from s to all nodes**

**­­­­­­­­­­­­**

**void bellman\_ford(int s, bool& neg\_cycle) {**

**dist[s] = 0;**

**// relax all edges**

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

**for (auto &edge : edges) {**

**int u = edge[0];**

**int v = edge[1];**

**int w = edge[2];**

**if(dist[u] != INT\_MAX && dist[u] + w < dist[v])**

**dist[v] = dist[u] + w;**

**}**

**// optional: detect negative cycle**

**for (auto &edge : edges) {**

**int u = edge[0];**

**int v = edge[1];**

**int w = edge[2];**

**if(dist[u] != INT\_MAX && dist[u] + w < dist[v])**

**neg\_cycle = true;**

**}**

**return;**

**}**

**int32\_t main(){**

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

**dist = vector<int>(n + 1, INT\_MAX);**

**for(int i = 0, u, v, w; i < m; ++ i ){**

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

**edges.push\_back({u, v, w});**

**}**

**bool f = false;**

**bellman\_ford(1, f);**

**}**

**// detected\_cycle with DFS**

**bool DFS(int node, int parent) {  
 visisted[node] = true;  
 for (int &child: edges[node])  
 if (!visisted[child]) {  
 if (DFS(child, node))  
 return true;  
 }  
 else if (parent != child)  
 return true;  
 return false;  
}**

**class Max\_Bipartite\_Match {**

**vector<vector<int> > &G;**

**public:**

**vector<int> match, mark;**

**int max\_match, stamp;**

**Max\_Bipartite\_Match(vector<vector<int> > &bipartite\_graph, int v): G(bipartite\_graph){**

**match.resize(v);**

**mark.resize(v);**

**for (int i = 0; i < v; i++) {**

**match[i] = -1;**

**mark[i] = -1;**

**}**

**stamp = 0;**

**}**

**bool augment\_path(int vertex){**

**for (int i = 0; i < G[vertex].size(); i++) {**

**int v = G[vertex][i];**

**if (mark[v] == stamp)**

**continue;**

**mark[v] = stamp;**

**if (match[v] == -1 || augment\_path(match[v])) {**

**match[v] = vertex;**

**return true;**

**}**

**}**

**return false;**

**}**

**int max\_matching(){**

**max\_match = 0;**

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

**stamp++;**

**if (augment\_path(i)) max\_match++;**

**}**

**return max\_match;**

**}**

**};**

**// non\_bipartite\_check with DFS**

**vector<int>edges[N];  
bool isVisited[N];  
bool color[N];  
long long countA, countB;  
  
bool isNonBipartite(int node, bool c) {  
 color[node] = c;  
 (color[node]) ? countA++ : countB++;  
 isVisited[node] = true;  
 bool nonBipartite = false;  
 for (auto &child : edges[node]) {  
 if (!isVisited[child]) {  
 nonBipartite |= isNonBipartite(child, !c);  
 }else if (color[node] == color[child])  
 return true;  
 }  
 return nonBipartite;  
}  
  
int main() {  
 int n, m, u, v;  
 cin >> n >> m;  
 for (int i = 0; i < m; i++) {  
 cin >> u >> v;  
 edges[u].push\_back(v);  
 edges[v].push\_back(u);  
 }  
 bool flag = 0;  
 for (int i = 1; i <= n; i++)  
 if (isVisited[i] == false)  
 flag |= isNonBipartite(i, true);  
 if (flag)  
 cout << "not Bipartite" << '\n';  
 else  
 cout << "Bipartite" << '\n';  
}**