**CS-304-AI -LAB(LAB TASK-5)**

**ROLL-423135**

**NAME-HARSHITH SUDA**

**CODE-1**

#include<bits/stdc++.h>

using namespace std;

struct Point {

    double x, y;

};

struct Edge {

    int u, v;

    double d;

};

int orientation(const Point &a, const Point &b, const Point &c) {

    double val = (b.y - a.y) \* (c.x - b.x) - (b.x - a.x) \* (c.y - b.y);

    if (fabs(val) < 1e-12) return 0;

    return (val > 0) ? 1 : 2;

}

bool onSegment(const Point &a, const Point &b, const Point &p) {

    return p.x <= max(a.x, b.x) + 1e-12 && p.x + 1e-12 >= min(a.x, b.x) &&

           p.y <= max(a.y, b.y) + 1e-12 && p.y + 1e-12 >= min(a.y, b.y);

}

bool segmentsIntersect(const Point &p1, const Point &q1, const Point &p2, const Point &q2) {

    int o1 = orientation(p1, q1, p2);

    int o2 = orientation(p1, q1, q2);

    int o3 = orientation(p2, q2, p1);

    int o4 = orientation(p2, q2, q1);

    if (o1 != o2 && o3 != o4) return true;

    if (o1 == 0 && onSegment(p1, q1, p2)) return true;

    if (o2 == 0 && onSegment(p1, q1, q2)) return true;

    if (o3 == 0 && onSegment(p2, q2, p1)) return true;

    if (o4 == 0 && onSegment(p2, q2, q1)) return true;

    return false;

}

struct DSU {

    vector<int> p, r;

    DSU(int n): p(n), r(n,0) { iota(p.begin(), p.end(), 0); }

    int find(int a){ return p[a]==a ? a : p[a]=find(p[a]); }

    bool unite(int a, int b){

        a = find(a); b = find(b);

        if (a==b) return false;

        if (r[a] < r[b]) swap(a,b);

        p[b]=a;

        if (r[a]==r[b]) r[a]++;

        return true;

    }

};

vector<vector<int>> generatePlanarGraph(int N = 100, int EXTRA\_LIMIT = 200, unsigned seed = std::random\_device{}()) {

*// Random points*

    mt19937\_64 rng(seed);

    uniform\_real\_distribution<double> unif(0.0, 1.0);

    vector<Point> pts(N);

    for (int i = 0; i < N; ++i) {

        pts[i].x = unif(rng);

        pts[i].y = unif(rng);

    }

*// All edges*

    vector<Edge> allEdges;

    for (int i = 0; i < N; i++) {

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

            double dx = pts[i].x - pts[j].x;

            double dy = pts[i].y - pts[j].y;

            double d = sqrt(dx\*dx + dy\*dy);

            allEdges.push\_back({i,j,d});

        }

    }

    sort(allEdges.begin(), allEdges.end(), [](const Edge &a, const Edge &b){ return a.d < b.d; });

*// MST first*

    DSU dsu(N);

    vector<pair<int,int>> edges;

    for (size\_t i = 0; i < allEdges.size(); i++) {

        Edge e = allEdges[i];

        if (dsu.unite(e.u, e.v)) {

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

            if ((int)edges.size() == N-1) break;

        }

    }

*// Add extra non-crossing edges*

    int extras\_added = 0;

    for (size\_t i = 0; i < allEdges.size(); i++) {

        if (extras\_added >= EXTRA\_LIMIT) break;

        int u = allEdges[i].u, v = allEdges[i].v;

        bool already = false;

        for (size\_t k = 0; k < edges.size(); k++) {

            if ((edges[k].first == u && edges[k].second == v) || (edges[k].first == v && edges[k].second == u)) {

                already = true; break;

            }

        }

        if (already) continue;

        bool crosses = false;

        for (size\_t k = 0; k < edges.size(); k++) {

            int a = edges[k].first, b = edges[k].second;

            if (a == u || a == v || b == u || b == v) continue;

            if (segmentsIntersect(pts[u], pts[v], pts[a], pts[b])) {

                crosses = true; break;

            }

        }

        if (!crosses) {

            edges.push\_back({u,v});

            extras\_added++;

        }

    }

*// Build adjacency matrix*

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

    for (size\_t i = 0; i < edges.size(); i++) {

        adj[edges[i].first][edges[i].second] = 1;

        adj[edges[i].second][edges[i].first] = 1;

    }

    return adj;

}

bool isSafeColor(vector<vector<int>>&graph,int c,vector<int>&color,int x){

    for(int i=0;i<graph.size();i++){

        if(graph[x][i]!=0 && color[i]==c){

            return false;

        }

    }

    return true;

}

bool graphColorUtil(vector<vector<int>>&graph,vector<int>&color,int x,int m){

    if(x==graph.size()){

        return true;

    }

    for(int i=1;i<=m;i++){

        if(isSafeColor(graph,i,color,x)){

            color[x]=i;

            if(graphColorUtil(graph,color,x+1,m)){

                return true;

            }

            color[x]=0;

        }

    }

    return false;

}

void mainGraphColoring(vector<vector<int>>&graph,int m){

    vector<int>color(graph.size(),0);

    if(graphColorUtil(graph,color,0,m)){

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

            cout<<"Node : "<<i+1<<"color: "<<color[i]<<endl;

        }

    }

    else{

        cout<<"NO SOLUTION"<<endl;

    }

    return;

}

int main(){

    ios\_base::sync\_with\_stdio(false);

    cin.tie(0);

    vector<vector<int>>graph=generatePlanarGraph(6);

    for(int i=0;i<graph.size();i++){

        for(int j=0;j<graph.size();j++){

            cout<<graph[i][j]<<" ";

        }

        cout<<endl;

    }

    cout<<endl;

    int m=4;

    mainGraphColoring(graph,m);

    return 0;

}