《离散数学II》实验报告

课程简介

课程名称：离散数学II

授课对象：计算机科学与技术 本科

教材版本：《离散数学教程》北京大学出版社

学时：52　其中：**理论：**44　**实验：**8

授课教师：　　**电子邮箱：**

实验二(必做，基本实验，3-4学时)

实验题目：通信网络上的线路设计问题

实验目的：

1、掌握无向连通图生成树的求解方法；

2、掌握基本回路系统和环路空间的求解方法；

3、掌握基本割集系统和断集空间的求解方法；

4、了解生成树、环路空间和断集空间的实际应用。

实验要求：

在通信网络中，节点之间有网络线路传输数据包，假设两节点之间最多有一条网络线路，给定该通信网络节点之间的连接关系，求解如下问题：

1、通过图结构描述上述通信网络？（如给出相邻矩阵：

$$A=\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

。）

2、在通信网络中确保消息可以有效地从源节点传播到所有目标节点，以减少了冗余的数据转发，我们通过构建一棵生成树，沿着生成树的拓扑结构进行传输，请给出一棵生成树的相邻矩阵。

3、有多少种上述实现方案？（提示：求方阵的行列式和秩见参考代码。）

4、如何在该通信网络中，解决数据包转发和路由选择的问题？我们可以通过探测和避免环路空间，可以保证数据在网络中的正常传输，并降低数据包丢失和延迟。

请输出基本回路系统（输出形式如：{$e_1e_4e_3$,$e_2e_5e_3$}）和环路空间（输出形式如：{Φ,$e_1e_4e_3$,$e_2e_5e_3$,$e_1e_4e_5e_2$}）。

5、在网络安全和隐私保护中，如何识别可能容易遭到攻击或数据泄露的路径或节点？通过分析断集空间，可以采取相应的保护措施，提高网络安全性和数据隐私性。

请输出基本割集系统（输出形式如：{{e_1,e_4},{e_2,e_5},{e_3,e_4,e_5}}）和断集空间（输出形式如：

{Φ,{e_1,e_4},{e_2,e_5},{e_3,e_4,e_5},{e_1,e_2,e_4,e_5},{e_1,e_3,e_5},{e_2,e_3,e_4},{e_1,e_2,e_3}}）。

）

***说明**：要求学生设计的程序要对教师给出的任意通信网络得出正确结果。

实验内容和实验步骤：（由学生填写）

需求分析

陈述程序设计的任务：

设计并实现一个程序，该程序能够对给定的通信网络进行以下分析：

通过图结构描述通信网络：将通信网络表示为图结构，并输出其相邻矩阵。

构建生成树：找到一棵生成树，并输出其相邻矩阵。

计算生成树的实现方案数量：计算所有可能的生成树数量。

解决数据包转发和路由选择问题：找到基本回路系统和环路空间，以避免环路并确保数据包的正常传输。

网络安全和隐私保护分析：识别可能容易遭到攻击或数据泄露的路径或节点，通过分析基本割集系统和断集空间，采取相应的保护措施。

输入的形式和输入值的范围：

输入为通信网络节点之间的连接关系，以邻接表或边列表的形式给出。

每个连接关系由一对节点表示，例如 (node1, node2)。

输出的形式：

相邻矩阵（表示通信网络）。

生成树的相邻矩阵。

生成树的数量。

基本回路系统和环路空间。

基本割集系统和断集空间。

程序所能实现的功能：

构建并输出通信网络的相邻矩阵。

找到并输出一棵生成树的相邻矩阵。

计算并输出生成树的数量。

找到并输出基本回路系统和环路空间。

找到并输出基本割集系统和断集空间。

概要设计

数据结构定义：

Graph：使用邻接矩阵表示通信网络。

Tree：使用邻接矩阵表示生成树。

Edge：表示通信网络中的一条边。

主程序的流程：

读取输入，构建通信网络图。

计算并输出通信网络的相邻矩阵。

使用算法（如 Prim 或 Kruskal 算法）找到一棵生成树，并输出其相邻矩阵。

计算并输出生成树的数量（通过计算行列式或矩阵树定理）。

使用深度优先搜索（DFS）或广度优先搜索（BFS）找到基本回路系统和环路空间。

使用图论算法（如最小割集算法）找到基本割集系统和断集空间。

程序模块之间的调用关系：

主程序：调用构建图、生成树、计算生成树数量、计算基本回路系统和环路空间、计算基本割集系统和断集空间的函数。

构建图：负责读取输入并构建通信网络图。

生成树：负责找到一棵生成树。

生成树数量：负责计算生成树的数量。

基本回路系统和环路空间：负责找到基本回路系统和环路空间。

基本割集系统和断集空间：负责找到基本割集系统和断集空间。

实验测试数据、代码及相关结果分析：（由学生填写）

```
#include<iostream>
```

```
#include<vector>
#include<stack>
#include<unordered_set>
using namespace std;

vector<vector<int>>  Correlation_matrix(vector<vector<int>>  adjacency_matrices)//
求取关联矩阵
{
	vector<vector<int>> t;
	for (int i = 0; i < adjacency_matrices.size(); i++) {
		for (int j = 0; j < adjacency_matrices.size(); j++) {
			if (adjacency_matrices[i][j] > 0) {
				for (int k = 0; k < adjacency_matrices[i][j]; k++) {
					vector<int> line;
					line.assign(adjacency_matrices.size(), 0);
					line[i] = 1;
					line[j] = 1;
					t.push_back(line);
					adjacency_matrices[i][j]--;
					adjacency_matrices[j][i]--;
				}
			}
		}
	}
	return t;
}

int Calculate_determinant(int n, vector<vector<int> >a)
{
	// 使用递归算法计算行列式的值
	if (n == 1)
	{
		return a[0][0];
	}
	else if (n == 2)
		return a[0][0] * a[1][1] - a[0][1] * a[1][0];
	else
```

```
    {
        int sum = 0;
        for (int k = 0; k < n; k++)
        {
            vector<vector<int> > b;
            for (int i = 1; i < n; i++)
            {
                vector<int>c;
                for (int j = 0; j < n; j++)
                {
                    if (j == k)
                        continue;
                    c.push_back(a[i][j]);
                }
                b.push_back(c);
            }
            sum = k % 2 == 0 ? sum + a[0][k] * Calculate_determinant(n - 1, b) :
sum - a[0][k] * Calculate_determinant(n - 1, b);
        }
        return sum;
    }
}

vector<vector<int>> Make_degree_matrix(vector<vector<int>>adjacency_matrices)//
构造度数矩阵
{
    vector<vector<int>> degree_matrix;
    for (int i = 0; i < adjacency_matrices[0].size(); i++) {
        int sum_degree_temp = 0;
        for (auto& j : adjacency_matrices) {
            sum_degree_temp += j[i];
        }
        vector<int> temp;
        temp.assign(adjacency_matrices.size(), 0);
        temp[i] = sum_degree_temp;
        degree_matrix.push_back(temp);
    }
```

```
    return degree_matrix;
}

int tree_num(vector<vector<int>>adjacency_matrices)//计算生成树的数量
{
    vector<vector<int>> degree_matrix = Make_degree_matrix(adjacency_matrices);
    for (int i = 0; i < adjacency_matrices.size(); i++) {
        for (int j = 0; j < adjacency_matrices.size(); j++) {
            degree_matrix[i][j] -= adjacency_matrices[i][j];//构造拉普拉斯矩阵
        }
    }
    degree_matrix.pop_back();//删除最后拉普拉斯矩阵的最后一行与最后一列
    for (auto& row : degree_matrix) {
        row.pop_back();
    }
    int row = degree_matrix.size();
    int value_determinant = Calculate_determinant(row, degree_matrix);//计算行列式的值
    return value_determinant;
}

void Show_correlation_matrix(vector<vector<int>> correlation_matrix)//显示关联矩阵
{
    cout << "此邻接矩阵的关联矩阵如下" << endl;
    cout << "\t";
    for (int i = 0; i < correlation_matrix.size(); i++) {
        cout << "e" << i << "\t";
    }
    cout << endl;

    for (int i = 0; i < correlation_matrix[0].size(); i++) {
        cout << "v" << i << "\t";
        for (auto& j : correlation_matrix) {
            cout << j[i] << "\t";
        }
        cout << endl;
```

```
    }
}

void Show_adjacency_matrices(vector<vector<int>> adjacency_matrices)//显示邻接矩阵
{
    cout << "邻接矩阵显示如下" << endl;
    cout << "\t";
    for (int i = 0; i < adjacency_matrices.size(); i++) {
        cout << "v" << i << "\t";
    }
    cout << endl;
    for (int i = 0; i < adjacency_matrices.size(); i++) {
        cout << "v" << i << "\t";
        for (int j = 0; j < adjacency_matrices.size(); j++) {
            cout << adjacency_matrices[i][j] << "\t";
        }
        cout << endl;
    }
}

void Show_tree_num(vector<vector<int>> adjacency_matrices) {
    int value_determinant = tree_num(adjacency_matrices);
    cout << "生成树的数量为: " << value_determinant << endl;
}

// 深度优先搜索（DFS）构建生成树
void dfs(int node, vector<vector<int>>adjacency_matrices, vector<bool>& visited, vector<vector<int>>& tree) {
    visited[node] = true;
    for (int i = 0; i < adjacency_matrices.size(); ++i) {
        if (adjacency_matrices[node][i] != 0 && !visited[i]) {
            // 记录边 i->node 或 node->i
            tree[node][i] = 1;
            tree[i][node] = 1;
            dfs(i, adjacency_matrices, visited, tree);
        }
```

```
    }
}

vector<vector<int>> Generate_tree(vector<vector<int>>adjacency_matrices) {
    int n = adjacency_matrices.size();
    vector<bool> visited(n, false);  // 标记节点是否访问过
    vector<vector<int>> tree(adjacency_matrices.size(), vector<int>(adjacency_matrices.size(), 0));  // 存储生成树的边

    // 从第一个节点开始 DFS 构建生成树
    dfs(0, adjacency_matrices, visited, tree);
    return tree;
}

vector<vector<int>> Find_trimming(vector<vector<int>>adjacency_matrices, vector<vector<int>> tree) { // 找出割边
    vector<vector<int>> trimming;
    for (int i = 0; i < adjacency_matrices.size(); i++) {
        for (int j = 0; j < adjacency_matrices.size(); j++) {
            adjacency_matrices[i][j] -= tree[i][j];
            if (adjacency_matrices[i][j] == 1 && i < j) {
                vector<int>temp;
                temp.push_back(i);
                temp.push_back(j);
                trimming.push_back(temp);
            }
        }
    }
    return trimming;
}

vector<vector<int>> Find_tree(vector<vector<int>> tree) {  // 找出树的边
    vector<vector<int>> tree_side;
    for (int i = 0; i < tree.size(); i++) {
        for (int j = 0; j < tree.size(); j++) {
            if (tree[i][j] == 1 && i < j) {
                vector<int>temp;
```

```
                temp.push_back(i);
                temp.push_back(j);
                tree_side.push_back(temp);
            }
        }
    }
    return tree_side;
}

void dfs_now(int begin, int end, vector<vector<int>>tree, vector<int>& loop_now, int symbol) { //  环路 dfs
    loop_now.push_back(end);
    for (int i = 0; i < tree[end].size(); i++) {
        if (tree[end][i] == 1 && i != begin)
        {
            if (i == symbol) {

                loop_now.push_back(symbol);
                return;
            }
            else {
                dfs_now(end, i, tree, loop_now, symbol);
            }
        }
    }
}

vector<vector<int>>Construct_loops(vector<vector<int>>trimming, vector<vector<int>> tree) { //构造基本回路系统
    vector<vector<int>>loops;
    for (int i = 0; i < trimming.size(); i++) {
        vector<int>loop_now;
        int begin = trimming[i][0];
        int end = trimming[i][1];
        dfs_now(begin, end, tree, loop_now, begin);
        auto it = find(loop_now.begin(), loop_now.end(), begin);
        if (it != loop_now.end()) {
```

```
                loop_now.erase(it + 1, loop_now.end());
            }
            loops.push_back(loop_now);
        }
        return loops;
}

void dfs_cut(int node, vector<vector<int>>& adj, vector<bool>& visited, unordered_set<int>& component) {
        stack<int> s;
        s.push(node);
        visited[node] = true;
        while (!s.empty()) {
            int current = s.top();
            s.pop();
            component.insert(current);

            for (int neighbor = 0; neighbor < adj.size(); ++neighbor) {
                if (adj[current][neighbor] == 1 && !visited[neighbor]) {
                    visited[neighbor] = true;
                    s.push(neighbor);
                }
            }
        }
}

// 求图中去除生成树一条边后的断集
vector<vector<int>> Find_cut_set(vector<vector<int>>& adj, vector<int>edge_to_remove, vector<vector<int>>& tree) {
        int n = adj.size();
        vector<bool> visited(n, false);
        adj[edge_to_remove[0]][edge_to_remove[1]] = 0;
        adj[edge_to_remove[1]][edge_to_remove[0]] = 0;
        tree[edge_to_remove[0]][edge_to_remove[1]] = 0;
        tree[edge_to_remove[1]][edge_to_remove[0]] = 0;

        // 获取去除该边后的连通分量
```

```
    unordered_set<int> component;
    dfs_cut(edge_to_remove[0], tree, visited, component);

    // 找到与断开的两个连通块相连接的边
    vector<vector<int>> cut_set;
    for (int u = 0; u < n; ++u) {
        for (int v = u + 1; v < n; ++v) {
            if (adj[u][v] == 1) {
                // 如果 u 和 v 属于不同的连通块，则这条边是割边
                if (component.find(u) == component.end() && component.find(v) != component.end() || component.find(u) != component.end() && component.find(v) == component.end()) {
                    vector<int> temp;
                    temp.push_back(u);
                    temp.push_back(v);
                    cut_set.push_back(temp);
                }
            }
        }
    }
    adj[edge_to_remove[0]][edge_to_remove[1]] = 1;
    adj[edge_to_remove[1]][edge_to_remove[0]] = 1;
    tree[edge_to_remove[0]][edge_to_remove[1]] = 1;
    tree[edge_to_remove[1]][edge_to_remove[0]] = 1;
    return cut_set;
}
vector<int> trans_V_to_E(vector<vector<int>>correlation_matrix, vector<vector<int>>side_v) {
    int n = correlation_matrix.size();
    int n_v = side_v.size();
    vector<int> side_e;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n_v; j++) {
            if (correlation_matrix[i][side_v[j][0]] == 1 && correlation_matrix[i][side_v[j][1]] == 1)
                side_e.push_back(i);
        }
    }
```

```
    return side_e;
}

void show_system(vector<vector<int>>side_e) {
    for (int i = 0; i < side_e.size(); i++) {
        for (int j = 0; j < side_e[i].size(); j++) {
            cout << "e" << side_e[i][j];
        }
        cout << ",";
    }
}


vector<int> Union_sets(const vector<int>& vec1, const vector<int>& vec2) {
    unordered_set<int> set1(vec1.begin(), vec1.end());
    unordered_set<int> set2(vec2.begin(), vec2.end());
    unordered_set<int> common;  // 用来存储两个集合的交集
    // 查找两个集合的交集
    for (int num : set1) {
        if (set2.find(num) != set2.end()) {
            common.insert(num);  // 如果在 set2中也存在，加入交集
        }
    }
    vector<int> result;
    for (int num : vec1) {
        if (common.find(num) == common.end()) {
            result.push_back(num);
        }
    }
    for (int num : vec2) {
        if (common.find(num) == common.end()) {
            result.push_back(num);
        }
    }
    return result;
}
```

```
bool Is_exist(const vector<vector<int>>& loopSpace, const vector<int>& loop) {
    unordered_set<int> loopSet(loop.begin(), loop.end());
    // 遍历回路空间中的每一个回路
    for (const auto& existingLoop : loopSpace) {
        unordered_set<int> existingLoopSet(existingLoop.begin(), existingLoop.end());
        if (loopSet == existingLoopSet) {
            return true; // 如果回路已存在，则返回 true
        }
    }
    return false;
}

void Generate_space_recursive(const vector<vector<int>>& basicCycles, vector<vector<int>>& loopSpace, vector<vector<int>>& newCombinations, size_t idx) {
    if (idx >= basicCycles.size()) {
        return;
    }

    // 尝试将当前回路直接加入到回路空间中
    const auto& cycle = basicCycles[idx];
    if (!Is_exist(loopSpace, cycle)) {
        loopSpace.push_back(cycle);
    }

    for (size_t i = 0; i < loopSpace.size(); ++i) {
        vector<int> newCycle = Union_sets(loopSpace[i], cycle);
        if (!Is_exist(loopSpace, newCycle) && !Is_exist(newCombinations, newCycle)) {
            newCombinations.push_back(newCycle);
        }
    }

    if (!newCombinations.empty()) {
        loopSpace.insert(loopSpace.end(), newCombinations.begin(), newCombinations.end());
```

```
            newCombinations.clear();
            Generate_space_recursive(basicCycles, loopSpace, newCombinations, idx
+ 1);
        }
}

vector<vector<int>> Generate_space(const vector<vector<int>>& basicCycles) {
        vector<vector<int>> loopSpace;
        vector<vector<int>> newCombinations;
        Generate_space_recursive(basicCycles, loopSpace, newCombinations, 0);
        return loopSpace;
}

int main()
{
        int num_vertices, t;
        cout << "请输入点的个数:" << endl;
        cin >> num_vertices;

        cout << "请输入矩阵" << endl;
        vector<vector<int>> adjacency_matrices;
        for (int i = 0; i < num_vertices; i++) {
            vector<int> temp;
            for (int j = 0; j < num_vertices; j++) {
                cin >> t;
                temp.push_back(t);
            }
            adjacency_matrices.push_back(temp);
        }

        vector<vector<int>>                    correlation_matrix                    =
Correlation_matrix(adjacency_matrices);
        Show_correlation_matrix(correlation_matrix);//显示关联矩阵
        Show_tree_num(adjacency_matrices);
        vector<vector<int>> tree = Generate_tree(adjacency_matrices);

        cout << "该图生成树的邻接矩阵" << endl;
```

```
Show_adjacency_matrices(tree);

vector<vector<int>>trimming = Find_trimming(adjacency_matrices, tree);
vector<vector<int>>loops = Construct_loops(trimming, tree);

vector<vector<int>>loops_side_e;// 用边代表的回路系统
for (int i = 0; i < loops.size(); i++) {
    vector<vector<int>>loops_side_v;//用点代表的回路系统
    for (int j = 0; j < loops[i].size(); j++) {
        int temp = j + 1;
        if (temp == loops[i].size())
            temp = 0;
        vector<int>temp_v;
        temp_v.push_back(loops[i][j]);
        temp_v.push_back(loops[i][temp]);
        loops_side_v.push_back(temp_v);
    }
    vector<int>temp_e = trans_V_to_E(correlation_matrix, loops_side_v);
    loops_side_e.push_back(temp_e);

}

cout << "基本回路系统:{ ";
show_system(loops_side_e);
cout << "}" << endl;
cout << "环路空间: {  Φ,";
show_system(Generate_space(loops_side_e));
cout << "}" << endl;
vector<vector<int>> tree_side = Find_tree(tree);
vector<vector<int>>cut_side_e;
for (int i = 0; i < tree_side.size(); i++) {
    vector<vector<int>>cut_side_v;
    vector<int> temp_v;
    temp_v.push_back(tree_side[i][0]);
    temp_v.push_back(tree_side[i][1]);
    cut_side_v.push_back(temp_v);
    vector<vector<int>>  cut_set  =  Find_cut_set(adjacency_matrices,
```

```
tree_side[i], tree);
            for (int i = 0; i < cut_set.size(); i++) {
                vector<int> temp_v;
                temp_v.push_back(cut_set[i][0]);
                temp_v.push_back(cut_set[i][1]);
                cut_side_v.push_back(temp_v);
            }
            vector<int>temp_e = trans_V_to_E(correlation_matrix, cut_side_v);
            cut_side_e.push_back(temp_e);
        }

        cout << "基本割集系统:{ ";
        show_system(cut_side_e);
        cout << "}" << endl;
        cout << "断集空间:{  Φ,";
        show_system(Generate_space(cut_side_e));
        cout << "}" << endl;
        return 0;
}
```

```
请输入点的个数:
4
请输入矩阵
0 1 1 1
1 0 0 1
1 0 0 1
1 1 1 0
此邻接矩阵的关联矩阵如下
        e0      e1      e2      e3      e4
v0      1       1       1       0       0
v1      1       0       0       1       0
v2      0       1       0       0       1
v3      0       0       1       1       1
生成树的数量为: 8
该图生成树的邻接矩阵
邻接矩阵显示如下
        v0      v1      v2      v3
v0      0       1       0       0
v1      1       0       0       1
v2      0       0       0       1
v3      0       1       1       0
基本回路系统:{ e0e1e3e4,e0e2e3,}
环路空间: { Φ,e0e1e3e4,,e0e2e3,e1e4e2,}
基本割集系统:{ e0e1e2,e1e2e3,e1e4,}
断集空间:{ Φ,e0e1e2,,e1e2e3,e0e3,e1e4,e0e2e4,e2e3e4,e0e3e1e4,}

C:\Users\HP\Desktop\24 VS C++\离散实验1\x64\Debug\离散实验1.exe (进程 44252)已退出, 代码为 0。
按任意键关闭此窗口. . .
```

```
请输入矩阵
0 1 0 1 0
1 0 1 1 0
0 1 0 1 0
1 1 1 0 1
0 0 0 1 0
此邻接矩阵的关联矩阵如下
        e0      e1      e2      e3      e4      e5
v0      1       1       0       0       0       0
v1      1       0       1       1       0       0
v2      0       0       1       0       1       0
v3      0       1       0       1       1       1
v4      0       0       0       0       0       1
生成树的数量为: 8
该图生成树的邻接矩阵
邻接矩阵显示如下
        v0      v1      v2      v3      v4
v0      0       1       0       0       0
v1      1       0       1       0       0
v2      0       1       0       1       0
v3      0       0       1       0       1
v4      0       0       0       1       0
基本回路系统:{ e0e1e2e4,e2e3e4,}
环路空间: { Φ,e0e1e2e4,,e2e3e4,e0e1e3,}
基本割集系统:{ e0e1,e1e2e3,e1e3e4,e5,}
断集空间:{ Φ,e0e1,,e1e2e3,e0e2e3,e1e3e4,e0e3e4,e2e4,e0e2e1e4,e5,e0e1e5,e1e2e3e5,e0e2e3e5,e1e3e4e5,e0e3e4e5,e2e4e5,e0e2e1
e4e5,}
```

实验总结：（由学生填写）

本次通信网络线路设计实验旨在深入理解通信网络中的关键图论概念及其实际应用。通过实践，我们掌握了无向连通图生成树的求解方法，这对于构建高效、无冗余的通信网络至关重要。同时，我们也学习了基本回路系统和环路空间的求解方法，以及基本割集系统和断集空间的求解技巧，这些都有助于分析网络的可靠性和稳定性。实验不仅增强了我们的理论知识，还让我们深刻体会到生成树、环路空间和断集空间在实际通信网络设计中的重要性。通过这次实验，我们对通信网络线路设计有了更全面、深入的理解。

附录：实验报告的要求

实验报告是反映学生实验效果的最主要的依据，也是学生正确地表达问题、综合问题和发现问题的能力的基本培养手段，因而是非常重要的内容，本课程的实验报告中要包括以下几项内容：

（一） 实验题目；

（二） 实验目的；

（三） 实验要求；

（四） 实验内容和实验步骤；

1．需求分析：陈述程序设计的任务，强调程序要做什么，明确规定：

（1）输入的形式和输入值的范围；

（2）输出的形式；

（3）程序所能实现的功能；

2．概要设计：说明用到的数据结构定义、主程序的流程及各程序模块之间的调用关系。

3．详细设计：提交带注释的源程序或者用伪代码写出每个操作所涉及的算法。

4．调试分析：

（1）调试过程中所遇到的问题及解决方法；

（2）算法的时空分析；

（五） 实验结果：列出对于给定的输入所产生的输出结果。若可能，测试随输入规模的增长所用算法的实际运行时间的变化。

（六） 实验总结：有关实验过程中的感悟和体会、经验和教训等。

C++参考代码：

```
#include <vector>

using namespace std;

// 传入的参数依次为行列式的阶数和行列式数组，返回值为行列式的值
int valueOfMatrix(int n, vector<vector<int> >a)
{
    // 使用递归算法计算行列式的值
    if (n == 1)
    {
        return a[0][0];
    }
    else if(n == 2)
        return a[0][0]*a[1][1]-a[0][1]*a[1][0];
    else
    {
        int sum = 0;
        for(int k=0; k<n; k++)
        {
            vector<vector<int> > b;
            for(int i=1; i<n; i++)
            {
                vector<int>c;
                for(int j=0; j<n; j++)
                {
                    if(j == k)
                        continue;
                    c.push_back(a[i][j]);
                }
                b.push_back(c);
            }
            sum = k%2==0 ? sum+a[0][k]*valueOfMatrix(n-1, b) : sum-a[0][k]*valueOfMatrix(n-1, b);
        }
        return sum;
    }
```

```
}

// 传入的参数为行列式的阶数和行列式数组，返回值为矩阵的秩
int rankOfDeterminant(int n, vector<vector<int> > matrix)
{
    // 求二进制矩阵的秩，即消元，最后看斜对角线上有几个1
    int row = 0;
    for(int col=0; col < n && row < n; col++, row++)  // 从每一列开始，将每一列消到只有 1 个 1 或者全 0
    {
        int i = 0;
        for(i = row; i < n; ++i)  // 寻找这一列第一个非 0 的行
        {
            if(matrix[i][col] != 0)
                break;
        }
        if(n == i)
            --row;
        else
        {
            swap(matrix[row], matrix[i]);
            for(int k = i+1; k < n; k++)
            {
                if(0 != matrix[k][col])
                {
                    for(int j = col; j < n; j++)
                    {
                        matrix[k][j] ^= matrix[row][j];// 用第 r 行的 1 消除这一列上其余全部行的 1
                    }
                }
            }
        }
    }
    return row;
}
```