算法实现题：

6-1

算法实现题 6

6-1　最小长度电路板排列问题。

问题描述：最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应不同的电路板插入方案。

设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。

试设计一个队列式分支限界法找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

算法设计：对于给定的电路板连接块，设计一个队列式分支限界法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m（$1 \leq m, n \leq 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中，为 1 表示电路板 k 在连接块 j 中。

结果输出：将计算的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第 1 行是最小长度；接下来的 1 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
1 1 1 1 1	5 4 3 1 6 2 8 7
0 1 0 1 0	
0 1 1 1 0	
1 0 1 1 0	
1 0 1 0 0	
1 1 0 1 0	
0 0 0 0 1	
0 1 0 0 1	

算法设计思路：

分析问题状态：

需要找到一种电路板的排列方式，使得所有连接块的最大长度最小。每个连接块中的电路板用同一根导线连接，连接块的长度是该连接块中第一个和最后一个电路板之间的距离。

状态定义及初始化：读取输入数据，构建电路板与连接块的关系矩阵。每个状态表示当前已排列的电路板序列及对应的连接块最大长度。

分支限界：

队列操作：使用队列存储待处理的状态，每次从队列中取出一个状态进行扩展。对于当前状态，尝试将未排列的电路板插入到序列中，生成新的状态并计算其连接块最大长度。

剪枝：如果新生成的状态的最大长度大于当前最优解，则剪枝，不将其加入队列。

更新最优解：当某个状态的所有电路板都已排列完毕时，检查其最大长度是否优于当前最优解，如果是则更新最优解。

```
struct State {
    vector<int> boardOrder; // 当前电路板排列顺序
    int maxLength; // 当前连接块最大长度
    int lastBoardIndex; // 上一块电路板的位置索引
```

```
};

int n, m;
vector<vector<int>> connections; // 连接关系矩阵

// 计算给定排列下的最大连接块长度
int calculateMaxLength(const vector<int>& order) {
    vector<int> first(n + 1, INT_MAX), last(n + 1, -1);
    for (int i = 0; i < n; ++i) {
        int board = order[i];
        for (int j = 0; j < m; ++j) {
            if (connections[board][j]) {
                first[j] = min(first[j], i);
                last[j] = max(last[j], i);
            }
        }
    }

    int maxLength = 0;
    for (int j = 0; j < m; ++j) {
        if (first[j] != INT_MAX && last[j] != -1) {
            maxLength = max(maxLength, last[j] - first[j]);
        }
    }
    return maxLength;
}

void solve() {
    ifstream fin("input.txt");
   ofstream fout("output.txt");

    fin >> n >> m;
    connections.resize(n);
    for (int i = 0; i < n; ++i) {
        connections[i].resize(m);
        for (int j = 0; j < m; ++j) {
            fin >> connections[i][j];
        }
    }

    queue<State> q;
    q.push({{}, INT_MAX, -1});
    int bestLength = INT_MAX;
   vector<int> bestOrder;
```

```
    while (!q.empty()) {
        State currentState = q.front();
        q.pop();

        if (currentState.boardOrder.size() == n) {
            int length = calculateMaxLength(currentState.boardOrder);
            if (length < bestLength) {
                bestLength = length;
                bestOrder = currentState.boardOrder;
            }
            continue;
        }//已排列，跳过

        for (int nextBoard = 0; nextBoard < n; ++nextBoard) {
            if (find(currentState.boardOrder.begin(), currentState.boardOrder.end(),
nextBoard) != currentState.boardOrder.end()) {
                continue;
            }

            vector<int> newOrder = currentState.boardOrder;
            newOrder.push_back(nextBoard);

            int estimatedLength = calculateMaxLength(newOrder);
            if (estimatedLength >= bestLength) {
                continue;
            }//剪

            q.push({newOrder, estimatedLength, nextBoard});//入队
        }
    }

    fout << bestLength << endl;
    for (int board : bestOrder) {
        fout << board + 1 << " ";
    }
   fout << endl;

    fin.close();
    fout.close();
}
```

6-2

6-2 最小权顶点覆盖问题。

问题描述：给定一个赋权无向图 $G=(V, E)$，每个顶点 $v \in V$ 都有权值 $w(v)$。如果 $U \subseteq V$，且对任意 $(u, v) \in E$ 有 $u \in U$ 或 $v \in U$，就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

算法设计：对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 1, 2, …, n。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边 (u, v)。

结果输出：将计算的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt。文件的第 1 行是最小权顶点覆盖顶点权之和；第 2 行是最优解 x_i（$1 \leq i \leq n$），$x_i=0$ 表示顶点 i 不在最小权顶点覆盖中，$x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

输入文件示例	输出文件示例
input.txt	output.txt
7 7	13
1 100 1 1 1 100 10	1 0 1 1 0 0 1
1 6	
2 4	
2 5	
3 6	
4 5	
4 6	
6 7	

算法设计思路：

定义问题状态：

用一个数组 x[i]（i 从 1 到 n）表示顶点 i 是否被选入顶点覆盖集，x[i] = 1 表示选入，x[i] = 0 表示不选入。

记录当前已选顶点的权值和 weightSum，以及当前状态下未被覆盖边的数量 uncoveredEdges。

优先队列式分支限界：

初始化优先队列：初始状态所有顶点未选，权值和为 0，未覆盖边数为 m。将其加入队列。

扩展节点：从优先队列中取出当前最优节点（权值和最小的节点）。

对当前节点进行分支：分别考虑将当前未选顶点加入顶点覆盖集和不加入的情况。

加入顶点：更新权值和（加上该顶点权值），更新未覆盖边数（检查该顶点能覆盖的边）。

不加入顶点：权值和不变，未覆盖边数可能不变（若该顶点不影响未覆盖边）。

剪枝策略：如果当前节点的权值和已经大于当前已知的最小权值和，不再继续扩展该节点。如果未覆盖边数为 0，说明已经找到一个顶点覆盖，更新最小权值和及最优解。

终止条件：优先队列为空，此时已遍历完所有可能的状态，得到最小权顶点覆盖。

```
// 定义问题状态结构体
struct State {
    vector<int> x;   // 顶点选择状态，x[i] = 1 表示顶点 i 被选，0 表示未选
    int wSum;        // 已选顶点的权值和
    int ucEdges;     // 未被覆盖的边数
    State(const vector<int>& _x, int _wSum, int _ucEdges)
        : x(_x), wSum(_wSum), ucEdges(_ucEdges) {}
};
```

```cpp
// 比较函数，用于优先队列（最小堆）
struct CompareState {
    bool operator()(const State& a, const State& b) {
        return a.wSum > b.wSum;
    }
};

// 计算未覆盖边数
int countucEdges(const vector<vector<bool>>& graph, const vector<int>& x) {
    int uc = 0;
    int n = graph.size();
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (graph[i][j] && x[i] == 0 && x[j] == 0) {
                uc++;
            }
        }
    }
    return uc;
}

// 优先队列式分支限界法求解
pair<int, vector<int>> Solve(const vector<vector<bool>>& graph, const vector<int>& ws)
{
    int n = graph.size();
    priority_queue<State, vector<State>, CompareState> pq;
    vector<int> initialX(n, 0);
    int initialwSum = 0;
    int initialucEdges = countucEdges(graph, initialX);
    pq.push(State(initialX, initialwSum, initialucEdges));
    int minwSum = numeric_limits<int>::max();
    vector<int> bestX;
    while (!pq.empty()) {
        State cur = pq.top();
        pq.pop();
        if (cur.wSum >= minwSum) {
            continue;
        }
        if (cur.ucEdges == 0) {
            if (cur.wSum < minwSum) {
                minwSum = cur.wSum;
                bestX = cur.x;
            }
```

```
        } else {
            for (int i = 0; i < n; ++i) {
                if (cur.x[i] == 0) {
                    // 选入顶点 i
                    vector<int> newX = cur.x;
                    newX[i] = 1;
                    int newwSum = cur.wSum + ws[i];
                    int newucEdges = countucEdges(graph, newX);
                    pq.push(State(newX, newwSum, newucEdges));
                    // 不选入顶点 i
                    vector<int> notSelectX = cur.x;
                    int notSelectwSum = cur.wSum;
                    int notSelectucEdges = cur.ucEdges;
                    pq.push(State(notSelectX, notSelectwSum, notSelectucEdges));
                }
            }
        }
    }
    return make_pair(minwSum, bestX);
}

// 从文件读取输入数据
pair<vector<vector<bool>>, vector<int>> readInput(const string& filename) {
    ......;
}
//将数据写入文件
...
```

6-4

6-4 最小重量机器设计问题。

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。设计一个优先队列式分支限界法，给出总价格不超过 d 的最小重量机器设计。

算法设计：对于给定的机器部件重量和机器部件价格，设计一个优先队列式分支限界法，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量，以及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

算法设计思路:

定义问题状态:

用一个数组 partSupplier[n] 表示每个部件选择的供应商，其中 partSupplier[i] 表示部件 i 选择的供应商编号（从 1 到 m）。

记录当前已选部件的总重量 totalWeight 和总价格 totalCost。

优先队列式分支限界:

初始化优先队列，将初始状态（所有部件未选，总重量为 0，总价格为 0）加入队列。

扩展节点：从优先队列中取出当前最优节点，即总重量最小的节点。

对当前节点进行分支：对于当前未确定供应商的部件，分别考虑从 m 个供应商处购买的情况。

选择供应商 j：更新总重量（加上部件 i 从供应商 j 购买的重量），更新总价格（加上部件 i 从供应商 j 购买的价格）。

剪枝策略：如果当前节点的总价格已经超过给定的 d，不再继续扩展该节点;如果当前节点的总重量已经大于当前已知的最小总重量（在满足价格限制下），不再继续扩展该节点。

终止条件：当所有部件都确定了供应商且总价格不超过 d 时，更新最小总重量及对应的部件供应商选择方案；当优先队列为空时，说明已遍历完所有可能情况，得到最终结果。

```
// 定义问题状态结构体
struct State {
    vector<int> partSupplier;  // 记录每个部件选择的供应商
    int totalWeight;           // 已选部件的总重量
    int totalCost;             // 已选部件的总价格
    State(const vector<int>& _partSupplier, int _totalWeight, int _totalCost)
        : partSupplier(_partSupplier), totalWeight(_totalWeight), totalCost(_totalCost)
{}
};

// 比较函数，用于优先队列（最小堆）
```

```
struct CompareState {
    bool operator()(const State& a, const State& b) {
        return a.totalWeight > b.totalWeight;
    }
};

// 优先队列式分支限界法求解
pair<int, vector<int>> branchAndBound(const vector<vector<int>>& costMatrix, const
vector<vector<int>>& weightMatrix, int d) {
    int n = costMatrix.size();
    int m = costMatrix[0].size();
    priority_queue<State, vector<State>, CompareState> pq;//优先队列
    vector<int> initialPartSupplier(n, 0);
    int initialTotalWeight = 0;
    int initialTotalCost = 0;
    pq.push(State(initialPartSupplier, initialTotalWeight, initialTotalCost));
    int minWeight = numeric_limits<int>::max();
    vector<int> bestPartSupplier;
    while (!pq.empty()) {
        State current = pq.top();
        pq.pop();
        if (current.totalCost > d) {
            continue;
        }
        int currentPartIndex = current.partSupplier.size();
        if (currentPartIndex == n) {
            if (current.totalWeight < minWeight) {
                minWeight = current.totalWeight;
                bestPartSupplier = current.partSupplier;
            }
        } else {
            for (int j = 0; j < m; ++j) {
                vector<int> newPartSupplier = current.partSupplier;
                newPartSupplier[currentPartIndex] = j + 1;
                int newTotalWeight = current.totalWeight +
weightMatrix[currentPartIndex][j];
                int newTotalCost = current.totalCost + costMatrix[currentPartIndex][j];
                if (newTotalCost <= d && newTotalWeight < minWeight) {
                    pq.push(State(newPartSupplier, newTotalWeight, newTotalCost));
                }
            }
        }
    }
    return make_pair(minWeight, bestPartSupplier);
```

```
}

// 从文件读取输入数据
tuple<vector<vector<int>>, vector<vector<int>>, int> readInput(const string& filename)
{
    ......
}

// 将结果写入文件
void writeOutput(const string& filename, int minWeight, const vector<int>& partSupplier)
{
    ......
}
```

6-5

6-5　运动员最佳配对问题。

问题描述：羽毛球队有男女运动员各 n 人。给定 2 个 $n \times n$ 矩阵 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$。$P[i][j]$是男运动员 i 和女运动员 j 配对组成混合双打的男运动员竞赛优势；$Q[i][j]$是女运动员 i 和男运动员 j 配合的女运动员竞赛优势。由于技术配合和心理状态等因素影响，$P[i][j]$不一定等于 $Q[j][i]$。男运动员 i 和女运动员 j 配对组成混合双打的男女双方竞赛优势为 $P[i][j] \times Q[j][i]$。设计一个算法，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

算法设计：设计一个优先队列式分支限界法，对于给定的男女运动员竞赛优势，计算男女运动员最佳配对法，使各组男女双方竞赛优势的总和达到最大。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 $2n$ 行，每行 n 个数。前 n 行是 p，后 n 行是 q。

结果输出：将计算的男女双方竞赛优势的总和的最大值输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	52
10 2 3	
2 3 4	
3 4 5	
2 2 2	
3 5 3	
4 5 1	

定义问题状态:

用一个数组 match[n] 表示男运动员的配对情况，match[i] 表示男运动员 i 配对的女运动员编号（从 0 到 n - 1）。

记录当前已配对的男女运动员竞赛优势总和 advantageSum。

为了便于剪枝，还可以记录一个当前状态下，剩余未配对运动员能产生的最大竞赛优势的上界 upperBound。计算上界的方法是：对于未配对的男、女运动员，选取他们各自剩余可配对情况下的最大竞赛优势值进行累加。

分支限界:

初始化优先队列（最大堆），将初始状态（所有男运动员未配对，竞赛优势总和为 0，上界为理论最大可能值）加入队列。这里最大堆是因为要找竞赛优势总和的最大值。

扩展节点：从优先队列中取出当前最优节点（竞赛优势总和最大且上界最大的节点）。

对当前节点进行分支：对于当前未配对的男运动员，分别考虑与不同女运动员配对的情况。配对女运动员 j：更新 match 数组，更新竞赛优势总和（加上当前男运动员 i 和女运动员 j 配对的竞赛优势 P[i][j] * Q[j][i] ），同时更新上界（重新计算剩余未配对运动员能产生的最大竞赛优势）。

剪枝策略：如果当前节点的上界已经小于当前已知的最大竞赛优势总和，不再继续扩展该节点。因为即使后续都取到最大优势，也无法超过当前最优值。

终止条件：当所有男运动员都完成配对时，更新最大竞赛优势总和。当优先队列为空时，说明已遍历完所有可能情况，得到最终的最大竞赛优势总和。

```
// 定义问题状态结构体
struct State {
    vector<int> match;        // 记录男运动员的配对情况
    int advantageSum;         // 已配对的男女运动员竞赛优势总和
    int upperBound;           // 当前状态下剩余未配对运动员能产生的最大竞赛优势上界
    State(const vector<int>& _match, int _advantageSum, int _upperBound)
        : match(_match), advantageSum(_advantageSum), upperBound(_upperBound) {}
};

// 比较函数，用于优先队列（最大堆）
struct CompareState {
    bool operator()(const State& a, const State& b) {
        if (a.advantageSum != b.advantageSum) {
            return a.advantageSum < b.advantageSum;
        }
        return a.upperBound < b.upperBound;
    }
};

// 计算上界
int calculateUpperBound(const vector<vector<int>>& P, const vector<vector<int>>& Q,
const vector<int>& match) {
    int n = P.size();
    vector<bool> used(n, false);
    for (int i = 0; i < n; ++i) {
        if (match[i] != -1) {
            used[match[i]] = true;
        }
    }
    int upperBound = 0;
    for (int i = 0; i < n; ++i) {
        if (match[i] == -1) {
            int maxAdvantage = 0;
            for (int j = 0; j < n; ++j) {
```

```
                if (!used[j]) {
                    maxAdvantage = max(maxAdvantage, P[i][j] * Q[j][i]);
                }
            }
            upperBound += maxAdvantage;
        }
    }
    return upperBound;
}

// 优先队列式分支限界法求解
int branchAndBound(const vector<vector<int>>& P, const vector<vector<int>>& Q) {
    int n = P.size();
    priority_queue<State, vector<State>, CompareState> pq;
    vector<int> initialMatch(n, -1);
    int initialAdvantageSum = 0;
    int initialUpperBound = calculateUpperBound(P, Q, initialMatch);
    pq.push(State(initialMatch, initialAdvantageSum, initialUpperBound));
    int maxAdvantageSum = 0;
    while (!pq.empty()) {
        State current = pq.top();
        pq.pop();
        if (current.upperBound <= maxAdvantageSum) {
            continue;
        }
        int currentIndex = -1;
        for (int i = 0; i < n; ++i) {
            if (current.match[i] == -1) {
                currentIndex = i;
                break;
            }
        }
        if (currentIndex == -1) {
            maxAdvantageSum = max(maxAdvantageSum, current.advantageSum);
        } else {
            for (int j = 0; j < n; ++j) {
                if (find(current.match.begin(), current.match.end(), j) ==
current.match.end()) {
                    vector<int> newMatch = current.match;
                    newMatch[currentIndex] = j;
                    int newAdvantageSum = current.advantageSum + P[currentIndex][j] *
Q[j][currentIndex];
                    int newUpperBound = calculateUpperBound(P, Q, newMatch);
                    if (newUpperBound > maxAdvantageSum) {
```

```
                    pq.push(State(newMatch, newAdvantageSum, newUpperBound));
                }
            }
        }
    }
    }
    return maxAdvantageSum;
}

// 从文件读取输入数据
pair<vector<vector<int>>, vector<vector<int>>> readInput(const string& filename) {
    ......;
}
//写入数据
void writeOutput(const string& filename, int maxAdvantageSum) {
    ......;
}
```

6-10

6-10　世界名画陈列馆问题。

问题描述：世界名画陈列馆由 $m \times n$ 个排列成矩形阵列的陈列室组成。为了防止名画被盗，需要在陈列室中设置警卫机器人哨位。除了监视所在的陈列室，每个警卫机器人还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试设计一个安排警卫机器人哨位的算法，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

算法设计：设计一个优先队列式分支限界法，计算警卫机器人的最佳哨位安排，使名画陈列馆中每个陈列室都在警卫机器人的监视下，且所用的警卫机器人数最少。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 m 和 n（$1 \le m, n \le 20$）。

结果输出：将计算的警卫机器人数及其最佳哨位安排输出到文件 output.txt。文件的第 1 行是警卫机器人数；接下来的 m 行中每行 n 个数，0 表示无哨位，1 表示有哨位。

输入文件示例	输出文件示例
input.txt	output.txt
4 4	4
	0 0 1 0
	1 0 0 0
	0 0 0 1
	0 1 0 0

算法设计思路:

定义问题状态:

用一个二维数组 layout[m][n] 表示陈列室的布局，其中 layout[i][j] 为 0 表示该陈列室无警卫机器人哨位，为 1 表示有哨位。

记录当前已放置的警卫机器人数量 robotCount。

为了便于剪枝，计算当前状态下还未被监视的陈列室数量 unmonitoredCount。

分支限界:

初始化优先队列（最小堆），将初始状态（所有陈列室无哨位，机器人数量为 0，未监视陈列室数量为 m * n）加入队列。

扩展节点：从优先队列中取出当前最优节点（机器人数量最少且未监视陈列室数量最少的节点）。

对当前节点进行分支：对于当前未放置机器人的陈列室，分别考虑放置和不放置机器人的情况。

放置机器人：更新 layout 数组，将该陈列室设为有哨位（值为 1），更新机器人数量（加 1），更新未监视陈列室数量（检查该机器人能监视到的陈列室，减少相应数量）。

不放置机器人：layout 数组不变，机器人数量不变，未监视陈列室数量可能不变（若该陈列室不影响监视情况）。

剪枝策略：如果当前节点的机器人数量已经大于当前已知的最少机器人数量，不再继续扩展该节点。如果当前节点的未监视陈列室数量为 0，说明所有陈列室都已被监视，更新最少机器人数量及对应的布局。

终止条件：优先队列为空，此时已遍历完所有可能的状态，得到最少机器人数量及最佳哨位安排。

```
// 定义问题状态结构体
struct State {
    vector<vector<int>> layout;  // 陈列室布局
    int robotCount;             // 已放置的警卫机器人数量
    int unmonitoredCount;        // 未被监视的陈列室数量
    State(const vector<vector<int>>& _layout, int _robotCount, int _unmonitoredCount)
        : layout(_layout), robotCount(_robotCount), unmonitoredCount(_unmonitoredCount)
{}
};

// 比较函数，用于优先队列（最小堆）
struct CompareState {
    bool operator()(const State& a, const State& b) {
        if (a.robotCount != b.robotCount) {
            return a.robotCount > b.robotCount;
        }
        return a.unmonitoredCount > b.unmonitoredCount;
    }
};

// 检查坐标是否在陈列室范围内
bool isValid(int i, int j, int m, int n) {
    return i >= 0 && i < m && j >= 0 && j < n;
}

// 计算未被监视的陈列室数量
int countUnmonitored(const vector<vector<int>>& layout, int m, int n) {
    int count = 0;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (layout[i][j] == 0) {
                bool isMonitored = false;
                int dx[] = {-1, 1, 0, 0};//x 方向的移动
```

```
            int dy[] = {0, 0, -1, 1};//y 方向移动
            for (int k = 0; k < 4; ++k) {
                int newI = i + dx[k];
                int newJ = j + dy[k];
                if (isValid(newI, newJ, m, n) && layout[newI][newJ] == 1) {
                    isMonitored = true;
                    break;
                }//剪枝
            }
            if (!isMonitored) {
                count++;
            }
        }
    }//遍历每个陈列室
    }
    return count;
}

// 优先队列式分支限界法求解
pair<int, vector<vector<int>>> branchAndBound(int m, int n) {
    priority_queue<State, vector<State>, CompareState> pq;
    vector<vector<int>> initialLayout(m, vector<int>(n, 0));
    int initialRobotCount = 0;
    int initialUnmonitoredCount = m * n;
    pq.push(State(initialLayout, initialRobotCount, initialUnmonitoredCount));
    int minRobotCount = numeric_limits<int>::max();
    vector<vector<int>> bestLayout;
    while (!pq.empty()) {//循环找最优解
        State current = pq.top();
        pq.pop();
        if (current.robotCount >= minRobotCount) {
            continue;
        }
        if (current.unmonitoredCount == 0) {
            if (current.robotCount < minRobotCount) {
                minRobotCount = current.robotCount;
                bestLayout = current.layout;
            }
        } else {
            for (int i = 0; i < m; ++i) {
                for (int j = 0; j < n; ++j) {
                    if (current.layout[i][j] == 0) {
                        // 放置机器人
                        vector<vector<int>> newLayout = current.layout;
```

```
                        newLayout[i][j] = 1;
                        int newRobotCount = current.robotCount + 1;
                        int newUnmonitoredCount = countUnmonitored(newLayout, m, n);
                        if (newRobotCount < minRobotCount) {
                            pq.push(State(newLayout, newRobotCount,
newUnmonitoredCount));
                        }
                        // 不放置机器人
                        vector<vector<int>> notPlaceLayout = current.layout;
                        int notPlaceRobotCount = current.robotCount;
                        int notPlaceUnmonitoredCount = current.unmonitoredCount;
                        pq.push(State(notPlaceLayout, notPlaceRobotCount,
notPlaceUnmonitoredCount));
                    }
                }
            }
        }
    }
    return make_pair(minRobotCount, bestLayout);
}

// 从文件读取输入数据
pair<int, int> readInput(const string& filename) {
    ......
}

// 将结果写入文件
void writeOutput(const string& filename, int minRobotCount, const vector<vector<int>>&
bestLayout) {
    .......
}
```