云南省第五届青少年编程挑战赛 入门组决赛题解

一、 算法与考点分析

1. **核心考点**：图的遍历（连通块查找）、多源广度优先搜索（Multi-source BFS）、最短路径问题。
2. **问题转化（破题点）**：
   - 题目要求：保留**尽可能多**的地面网格（字符 '1'）。
   - 等价转换：也就是要求我们**挖掉最少**的地面网格，把两个湖泊连通。
   - 最终答案 = `初始地面总网格数` − `最少需要挖通的地面网格数`。
   - 此时问题变成了：**求湖泊 1 到湖泊 2 之间的最短曼哈顿距离（中间跨越的 '1' 的最少个数）**。
3. **复杂度分析**：
   - **时间复杂度**：$O(n \times m)$。网格大小最大为 $2000 \times 2000 = 4 \times 10^6$。无论是找湖泊还是 BFS，每个格子最多入队、出队一次，运算量千万级别，在 1.0 秒时限内绰绰有余。
   - **空间复杂度**：$O(n \times m)$。需要维护一个距离数组（兼作访问标记）和 BFS 队列，内存消耗约在 30MB 左右，远低于 256MB 的限制。

二、 逻辑链思维推导

读题后，我们可以将解题过程拆解为以下完整逻辑链（思维导图形式）：

```
【读入与预处理】
  └─ 读入 01 矩阵，同时统计整个地图中 '1'（地面）的总数量（total_land）
          ↓
【第一步：锁定湖泊 1（Flood Fill 找连通块）】
  └─ 遍历网格，找到第一个 '0'，作为起始点。
  └─ 启动一个普通的 BFS，将所有与它相连的 '0' 找出来。
  └─ 这些 '0' 组成了“湖泊 1”。
          ↓
【第二步：多源 BFS 扩张（核心状态转移）】
  └─ 把“湖泊 1”的所有网格全部放入一个主队列中，它们的初始步数(距离) dist 设为 0。
  └─ 开始一圈一圈向外扩张（向上下左右的 '1' 蔓延）：
       ├─ 遇到 '1'：说明需要挖通，距离 dist = 当前格子 dist + 1，并将该 '1' 加入队列。
       └─ 遇到未访问过的 '0'：触发终结条件！说明撞到了“湖泊 2”！
          ↓
【第三步：边界处理与输出】
  └─ 当撞到湖泊 2 的那一瞬间，当前所在格子的 dist 值，就是连通两个湖泊所需要挖掉的最少 '1' 的数量（min_dug）。
  └─ 最终答案 = total_land − min_dug。
```

三、 C++ 标准代码（标程）

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <string>

using namespace std;

const int MAXN = 2005;
string grid[MAXN];
int dist[MAXN][MAXN]; // 记录到湖泊1的最短距离，-1表示未访问过
int n, m;

// 方向数组：上下左右
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

int main() {
    // 开启快读，应对 2000x2000 的大数据
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    if (!(cin >> n >> m)) return 0;

    int total_land = 0;
    for (int i = 0; i < n; i++) {
        cin >> grid[i];
        for (int j = 0; j < m; j++) {
            if (grid[i][j] == '1') {
                total_land++; // 统计初始地面的总数
            }
            dist[i][j] = -1;  // 初始化距离矩阵为 -1
        }
    }

    queue<pair<int, int>> q; // 用于多源 BFS 的主队列
    bool found_lake1 = false;

    // 1. 寻找湖泊1，并将湖泊1的所有格子加入多源 BFS 的队列
    for (int i = 0; i < n && !found_lake1; i++) {
        for (int j = 0; j < m && !found_lake1; j++) {
            if (grid[i][j] == '0') {
                // 找到了湖泊1的第一个格子，用局部的 BFS 找出整个湖泊1
                queue<pair<int, int>> lake_q;
                lake_q.push({i, j});
                dist[i][j] = 0;
                q.push({i, j}); // 加入主队列

```

```
                while (!lake_q.empty()) {
                    auto [x, y] = lake_q.front();
                    lake_q.pop();

                    for (int k = 0; k < 4; k++) {
                        int nx = x + dx[k];
                        int ny = y + dy[k];
                        // 边界检查
                        if (nx >= 0 && nx < n && ny >= 0 && ny < m) {
                            // 必须是连通的 '0' 才属于同一个湖泊
                            if (grid[nx][ny] == '0' && dist[nx][ny] == -1) {

                                dist[nx][ny] = 0;
                                lake_q.push({nx, ny});
                                q.push({nx, ny}); // 同步加入多源BFS起点队列
                            }
                        }
                    }
                }
                found_lake1 = true; // 湖泊1找完后跳出循环
            }
        }
    }

    // 2. 多源 BFS，从湖泊1向外扩张，寻找湖泊2
    int min_dug = 0;
    bool connected = false;

    while (!q.empty() && !connected) {
        auto [x, y] = q.front();
        q.pop();

        for (int k = 0; k < 4; k++) {
            int nx = x + dx[k];
            int ny = y + dy[k];

            if (nx >= 0 && nx < n && ny >= 0 && ny < m) {
                // 如果碰到了另一个尚未被访问的 '0'，说明找到了湖泊2!
                if (grid[nx][ny] == '0' && dist[nx][ny] == -1) {
                    min_dug = dist[x][y]; // 当前所在格子的距离就是挖掉的最少格子数

                    connected = true;
                    break;
                }
                // 如果碰到了地面 '1'，并且没有走过，则将其挖通，步数+1
                if (grid[nx][ny] == '1' && dist[nx][ny] == -1) {
                    dist[nx][ny] = dist[x][y] + 1;
                    q.push({nx, ny});
```

```
                    }
                }
            }
        }

        // 3. 输出结果：保留的最大地面网格数 = 原始总地面数 - 挖通需要的最少地面数
        cout << total_land - min_dug << "\n";

        return 0;
    }
```

第二题：封印（seal）

一、 算法与考点分析

1. **核心考点**：数论、质因数分解、约数和定理、快速幂/模运算。
2. **问题转化**：
   - 给定 n 个数，求它们乘积 N 的所有正因子之和。
   - 根据**约数和定理**：如果 $N = p_1^{c_1} \times p_2^{c_2} \times \cdots \times p_k^{c_k}$（其中 p_i 为质数），那么它的所有正因子之和为：
     $$\sigma(N) = (1 + p_1 + p_1^2 + \cdots + p_1^{c_1}) \times \cdots \times (1 + p_k + p_k^2 + \cdots + p_k^{c_k})$$
   - 我们的任务是将所有的 a_i 分解质因数，把相同质数的指数加在一起，然后套用上述公式，最后对 $10^9 + 7$ 取模。
3. **复杂度分析**：因为 $a_i \le 2 \times 10^5$，我们可以用**欧拉筛法**预处理出每个数的“最小质因数（SPF）”，这样分解每个 a_i 只需要 $O(\log a_i)$ 的时间，非常迅速。

二、 逻辑链思维推导

```
【预处理】
  └ 线性筛法找出 2~200000 之间所有的质数。
  └ 同时记录下每个数的“最小质因数 (spf)”，为快速分解做准备。
        ↓
【汇总质因数】
  └ 遍历读入的每一个 a_i。
  └ 利用 spf 数组，将 a_i 不断除以它的最小质因数，直到变为 1。
  └ 将拆解出的质因数次数累加到全局的 count 数组中。
        ↓
【计算约数和】
  └ 遍历所有出现过的质数 p 和它对应的总次数 c。
  └ 计算等比数列和：(1 + p + p^2 + ... + p^c) mod (10^9+7)。
  └ 将所有质数的等比数列和相乘（边乘边取模）。
  └ 输出最终结果。
```

三、 C++ 标程

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int MAXA = 200005;
const int MOD = 1000000007;

int spf[MAXA]; // spf[i] 记录 i 的最小质因数
long long prime_count[MAXA]; // 记录每个质数在乘积中出现的总次数

void sieve() {
    for (int i = 2; i < MAXA; i++) spf[i] = i;
    for (int i = 2; i * i < MAXA; i++) {
        if (spf[i] == i) {
            for (int j = i * i; j < MAXA; j += i) {
                if (spf[j] == j) spf[j] = i;
            }
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    sieve();

    int n;
    if (!(cin >> n)) return 0;

    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;
        // 快速分解质因数
        while (a > 1) {
            prime_count[spf[a]]++;
            a /= spf[a];
        }
    }

    long long ans = 1;
    for (int p = 2; p < MAXA; p++) {
        if (prime_count[p] > 0) {
            long long current_sum = 1;
            long long term = 1;
            // 计算 1 + p + p^2 + ... + p^c
```

```
            for (int i = 1; i <= prime_count[p]; i++) {
                term = (term * p) % MOD;
                current_sum = (current_sum + term) % MOD;
            }
            ans = (ans * current_sum) % MOD;
        }
    }

    cout << ans << "\n";
    return 0;
}
```

第三题：圣诞树（tree）

一、 算法与考点分析

1. **核心考点**：树形 DP（动态规划）、状态机。
2. **问题转化**：
   - 条件：任何一条“叶子节点到根节点(1号)”的路径上，灯泡总数必须是 3 的倍数。
   - 这意味着：对于树上任意一个节点 u，**它的所有子树中的叶子节点，到 u 的路径上的灯泡总数对 3 取模的结果必须一模一样**。否则，加上从 u 到根的灯泡数后，不可能同时成为 3 的倍数。
   - 状态定义：`dp[u][i]` 表示在以 u 为根的子树中，使得所有叶子到 u 的路径灯泡数对 3 取模均为 `i` (0, 1, 2) 时，能放置的**最多灯泡数**。
3. **复杂度分析**：时间复杂度 $O(N)$，每个节点访问一次，状态转移 $O(1)$。

二、 逻辑链思维推导

```
【建树与DFS】
  └─ 把无向边存入邻接表，从节点 1 开始做深度优先搜索(DFS)。
        ↓
【状态设计与转移（核心)】
  └─ 如果当前节点是叶子节点:
      ├─ 放灯泡：路径模3等于1，dp[u][1] = 1。
      └─ 不放灯泡：路径模3等于0，dp[u][0] = 0。dp[u][2] 无效(-INF)。
  └─ 如果当前节点是普通节点，先递归计算所有儿子，汇总不同模数状态的最大灯泡数:
      ├─ 情况A（当前节点不放灯泡）：儿子们是什么模数，当前节点就是什么模数。
      └─ 情况B（当前节点放灯泡）：子节点的模数 j 加上 1 变成目标模数 i。
      └─ dp[u][i] = max(情况A, 情况B)。
        ↓
【得出答案】
  └─ 因为要求到根节点(1号)的路径总量是3的倍数，所以最终答案就是 dp[1][0]。
```

三、 C++ 标程

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int MAXN = 200005;
const long long INF = 1e18; // 使用一个极大值表示不可能的状态

vector<int> adj[MAXN];
long long dp[MAXN][3]; // dp[u][i]

void dfs(int u, int parent) {
    bool is_leaf = true;
    long long sum[3] = {0, 0, 0}; // 记录子节点统一模数为 j 时的最大灯泡和

    for (int v : adj[u]) {
        if (v != parent) {
            is_leaf = false;
            dfs(v, u);
            // 将子节点的状态累加
            for (int j = 0; j < 3; j++) {
                if (sum[j] != -INF) {
                    if (dp[v][j] == -INF) sum[j] = -INF; // 只要有一个子节点无法达到状态 j，则整个 sum[j] 废除
                    else sum[j] += dp[v][j];
                }
            }
        }
    }

    if (is_leaf) {
        // 叶子节点初始化
        dp[u][0] = 0;
        dp[u][1] = 1;
        dp[u][2] = -INF;
    } else {
        // 状态转移
        for (int i = 0; i < 3; i++) {
            // 1. 当前节点 u 不放灯泡：模数不变，直接继承子节点的 sum[i]
            long long put_0 = sum[i];

            // 2. 当前节点 u 放 1 个灯泡：需要子节点提供模数 (i+2)%3
            // 因为 ( (i+2)%3 + 1 ) % 3 == i
            long long put_1 = -INF;
            if (sum[(i + 2) % 3] != -INF) {
                put_1 = 1 + sum[(i + 2) % 3];
```

```
            }

            dp[u][i] = max(put_0, put_1);
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    if (!(cin >> n)) return 0;

    for (int i = 0; i < n - 1; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(1, 0);

    // 题目要求根节点的路径全为3的倍数，即模3为0
    cout << dp[1][0] << "\n";

    return 0;
}
```

第四题：道路规划（road）

一、 算法与考点分析

1. **核心考点**：有向无环图 (DAG)、拓扑排序、乘法原理、防整型溢出。
2. **问题转化**：
   - 给定起点 S 和终点 T，求哪个中间节点 X 经过的路径数最多。
   - 根据乘法原理：**经过节点 X 的路径总数 = (S 到 X 的路径数) $\times$ (X 到 T 的路径数)**。

- 我们可以分别对**原图**和**反向图**跑拓扑排序，利用动态规划求出上述两个路径数数组 `waysS` 和 `waysT`。

3. **避坑点（重难点）**：
   - 图中可能存在无效节点（比如 S 走到某个死胡同，无法到达 T）。在死胡同里的路径数可能会疯狂增加导致 `long long` 溢出。
   - **解决方案**：先跑两遍简单的 BFS，标记出**既能被 S 访问到，又能走到 T 的节点**。我们只在这些有效节点上做计算，题目保证了 S 到 T 路径数 $\leq 2^{63}-1$，所以有效节点上的乘积绝对不会溢出 `unsigned long long`。

二、 逻辑链思维推导

```
【提纯图（去除无用节点）】
  └ 在原图从 S 跑 BFS，得到 visS 数组（S 能到达的点）。
  └ 在反向图从 T 跑 BFS，得到 visT 数组（能到达 T 的点）。
  └ 两者都为 true 的点才是“有效节点”。（若 T 不是有效节点说明不可达，输出-1）。
        ↓
【正向拓扑排序计算 waysS】
  └ 计算有效子图的入度。
  └ 从 S 出发，waysS[S] = 1。
  └ 按照拓扑序，若 u -> v，则 waysS[v] += waysS[u]。
        ↓
【反向拓扑排序计算 waysT】
  └ 在反向图上计算有效子图的入度。
  └ 从 T 出发，waysT[T] = 1。
  └ 按照拓扑序，若 v -> u（原图），则 waysT[v] += waysT[u]。
        ↓
【统计与输出】
  └ 遍历所有非 S 非 T 的有效节点，计算 waysS[u] * waysT[u]。
  └ 维护最大值和对应的最小编号节点即可。
```

三、 C++ 标程

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int MAXN = 10005;
vector<int> adj[MAXN], rev_adj[MAXN];
bool visS[MAXN], visT[MAXN];
int in_degree_S[MAXN], in_degree_T[MAXN];
unsigned long long waysS[MAXN], waysT[MAXN];

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m, S, T;
    if (!(cin >> n >> m >> S >> T)) return 0;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        rev_adj[v].push_back(u);
    }

    // 1. 找 S 能到达的点
    queue<int> q;
    q.push(S);
    visS[S] = true;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (!visS[v]) { visS[v] = true; q.push(v); }
        }
    }

    // 2. 找能到达 T 的点
    q.push(T);
    visT[T] = true;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : rev_adj[u]) {
            if (!visT[v]) { visT[v] = true; q.push(v); }
        }
    }

```

```
    // 如果 S 无法到达 T
    if (!visS[T]) {
        cout << "-1\n";
        return 0;
    }

    // 3. 正向拓扑排序求 waysS
    for (int u = 1; u <= n; u++) {
        if (visS[u] && visT[u]) {
            for (int v : adj[u]) {
                if (visS[v] && visT[v]) in_degree_S[v]++;
            }
        }
    }
    q.push(S);
    waysS[S] = 1;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (visS[v] && visT[v]) {
                waysS[v] += waysS[u];
                if (--in_degree_S[v] == 0) q.push(v);
            }
        }
    }

    // 4. 反向拓扑排序求 waysT
    for (int u = 1; u <= n; u++) {
        if (visS[u] && visT[u]) {
            for (int v : rev_adj[u]) {
                if (visS[v] && visT[v]) in_degree_T[v]++;
            }
        }
    }
    q.push(T);
    waysT[T] = 1;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : rev_adj[u]) {
            if (visS[v] && visT[v]) {
                waysT[v] += waysT[u];
                if (--in_degree_T[v] == 0) q.push(v);
            }
        }
    }

    // 5. 寻找最重要的中间节点
    unsigned long long max_paths = 0;
```

```
    int best_node = -1;

    for (int u = 1; u <= n; u++) {
        if (u != S && u != T && visS[u] && visT[u]) {
            unsigned long long current_paths = waysS[u] * waysT[u];
            if (current_paths > max_paths) {
                max_paths = current_paths;
                best_node = u;
            } else if (current_paths == max_paths && current_paths > 0) {
                if (best_node == -1 || u < best_node) best_node = u; // 相同取编号最小
            }
        }
    }

    if (max_paths == 0 || best_node == -1) cout << "-1\n";
    else cout << best_node << " " << max_paths << "\n";

    return 0;
}
```