

# 2025 NOIP 第一场模拟赛

难度：绿绿蓝蓝

日期：2025 年 11 月 8 日

## P6739 【BalticOI 2014 Day1】 Three Friends

### ■ 题意

给定一个字符串  $U$ ，它是由字符串  $S$  经过以下操作得到的：先将  $S$  复制两份得到  $T$ （即  $T = S + S$ ），然后在  $T$  的任意位置（包括首尾）插入一个字符得到  $U$ 。要求根据  $U$  还原出原始的字符串  $S$ 。如果存在多个可能的  $S$ ，则输出 "NOT UNIQUE"；如果无法还原，则输出 "NOT POSSIBLE"。

数据范围：字符串  $U$  的长度  $n$  满足  $1 \leq n \leq 2 \times 10^6$ ，且  $n$  为奇数（因为插入一个字符后长度为奇数）。

## ■ 分析

核心思路是使用字符串哈希技术。字符串哈希将字符串视为一个基进制数，通过计算哈希值来快速比较字符串是否相等。具体步骤：

1. **预处理**：计算前缀哈希数组  $h[i]$  和基的幂数组  $pre[i]$ ，基取 22783，使用 `unsigned long long` 自然溢出作为模数。
2. **特判**：如果  $n$  为偶数，直接输出 "NOT POSSIBLE"，因为插入字符后长度必为奇数。

3. **枚举插入位置**: 设中点  $mid = \lfloor n/2 \rfloor + 1$ 。枚举插入位置  $k$ :
- 如果  $k \leq mid$ , 则删除  $U[k]$  后, 左半部分 (跳过  $k$ ) 的哈希值应与右半部分哈希值相等。
  - 如果  $k > mid$ , 则删除  $U[k]$  后, 右半部分 (跳过  $k$ ) 的哈希值应与左半部分哈希值相等。
4. **结果判断**: 根据匹配情况统计可能解的数量。如果只有一个解, 输出  $S$ ; 如果多个解但  $S$  相同, 输出  $S$ ; 如果多个解且  $S$  不同, 输出 "NOT UNIQUE"; 无解输出 "NOT POSSIBLE".

时间复杂度: 预处理  $O(n)$ , 枚举  $O(n)$ , 总时间复杂度  $O(n)$ 。

## ■ 参考代码

```
#define ull unsigned long long
const int N = 2e6 + 10;
ull base = 13331; // 哈希基数
int n, mid, len1, len2, ans;
string s, a, b, c, d;
ull hsh[N], pre[N]; // hsh: 前缀哈希数组, pre: 基数幂次数组

// 获取区间[l, r]的哈希值
ull getHash(int l, int r) { return hsh[r] - hsh[l - 1] * pre[r - l + 1]; }

// 获取区间[l, r]中删除位置k后的哈希值
ull getSubHash(int l, int r, int k) {
    // 将[l, k-1]和[k+1, r]两段拼接起来计算哈希
    return getHash(l, k - 1) * pre[r - k] + getHash(k + 1, r);
}
```

```
int main() {
    cin >> n >> s;

    // 检查字符串长度是否为奇数, 偶数长度不可能通过删除一个字符形成回文
    if (!(n % 2))
        puts("NOT POSSIBLE"), exit(0);

    s = " " + s; // 让字符串下标从1开始
    pre[0] = 1, mid = (n + 1) >> 1; // mid是中间位置

    // 预处理哈希数组和基数幂次数组
    for (int i = 1; i <= n; i++) {
        hsh[i] = hsh[i - 1] * base + (s[i] - 'A' + 1);
        pre[i] = pre[i - 1] * base;
    }

    // 情况1: 删除字符在左半部分
    len1 = getHash(mid + 1, n); // 右半部分的哈希值
    a = s.substr(mid + 1, n - mid); // 右半部分的字符串

    // 遍历左半部分的每个位置, 尝试删除该字符
    for (int i = 1; i <= mid; i++) {
        // 计算删除位置i后左半部分的哈希值
        len2 = getSubHash(1, mid, i);
        if (len1 == len2) { // 如果左右两半哈希值相等
            ans++;
            c = a; // 记录结果字符串
            break; // 找到一个解就退出
        }
    }
}
```

```
// 情况2: 删除字符在右半部分
len2 = getHash(1, mid - 1); // 左半部分(不含中间字符)的哈希值
b = s.substr(1, mid - 1);   // 左半部分的字符串

// 遍历右半部分的每个位置, 尝试删除该字符
for (int i = mid; i <= n; i++) {
    // 计算删除位置i后右半部分的哈希值
    len1 = getSubHash(mid, n, i);
    if (len1 == len2) { // 如果左右两半哈希值相等
        ans++;
        d = b; // 记录结果字符串
        break; // 找到一个解就退出
    }
}

// 输出结果
if (ans == 0)
    puts("NOT POSSIBLE"); // 无解
else if (ans == 1 || c == d)
    cout << (c.size() ? c : d) << endl; // 唯一解或两个解相同
else
    puts("NOT UNIQUE"); // 多个不同解

return 0;
}
```

## P11562 【MX-X7-T3】 [LSOT-3] 寄存器

### ■ 题意

给定一棵  $n$  个节点的树，每个节点初始值为 0。每次操作可以独立设置每条边的开关状态，然后选择一个节点通电。通电的节点会翻转值（0 变 1，1 变 0），并且通电会通过开启的边传播到相连的节点。问最少需要多少次通电操作可以使每个节点的值等于给定的  $a_i$ 。

数据范围： $1 \leq n \leq 10^6$ ， $0 \leq a_i \leq 1$ 。

## ■ 分析

每次操作相当于翻转一个连通子图（即一棵子树或一条路径）。问题转化为用最少的连通子图翻转操作，使得每个节点被翻转的次数奇偶性等于  $a_i$ （即  $a_i = 1$  的节点被翻转奇数次， $a_i = 0$  的节点被翻转偶数次）。

**关键观察：**在树结构中，最少操作次数与树上节点值变化的频率有关。具体地，考虑一条路径，如果沿路径节点值频繁变化（即相邻节点值不同），则需要更多操作来调整这些变化。因此，我们需要找到树上一条路径，使得沿路径节点值变化的次数最大。记这个最大变化次数为  $d$ ，则最少操作数为  $\lfloor (d + 1)/2 \rfloor$ 。

## 推导过程：

- 对于链情况（子任务2），问题简化为对数组进行区间翻转。观察发现，连续相同值的段可以合并，操作次数取决于值变化的次数。例如，数组  $[1, 0, 1, 0, 1]$  需要大约  $\lceil \text{变化次数}/2 \rceil$  次操作。
- 对于树情况，通过类似树的直径的方法，找到最长值变化路径（即路径上相邻节点值不同的边数最多）。两次 DFS 用于找到这条路径：
  - i. 第一次 DFS 从任意节点开始，计算到每个节点的路径上值变化的次数，并记录变化次数最大的节点  $maxp$ 。
  - ii. 第二次 DFS 从  $maxp$  开始，重新计算值变化次数，得到最大值  $d$ 。
- 答案即为  $\lfloor (d + 1)/2 \rfloor$ ，因为每次操作最多覆盖路径上连续两个值变化段，因此需要至少  $\lfloor (d + 1)/2 \rfloor$  次操作。

时间复杂度：两次 DFS 遍历整棵树，时间复杂度为  $O(n)$ 。

## ■ 参考代码

```
const int N = 1e6 + 5;
int a[N], n;
vector<int> g[N];
int maxp, maxd;

void dfs(int u, int fa, int deep) {
    if (fa != 0 && a[u] != a[fa]) deep++;
    if (a[u] == 1 && deep > maxd) {
        maxd = deep;
        maxp = u;
    }
    for (int v : g[u]) {
        if (v == fa) continue;
        dfs(v, u, deep);
    }
}
```

```
int main() {
    cin >> n;
    bool all_zero = true;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        if (a[i] == 1) all_zero = false;
    }
    if (all_zero) {
        cout << 0 << endl;
        return 0;
    }
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    maxd = 0;
    dfs(1, 0, 0);
    maxd = 0;
    dfs(maxp, 0, 0);
    cout << (maxd + 1) / 2 << endl;
    return 0;
}
```

## P10953 逃不掉的路

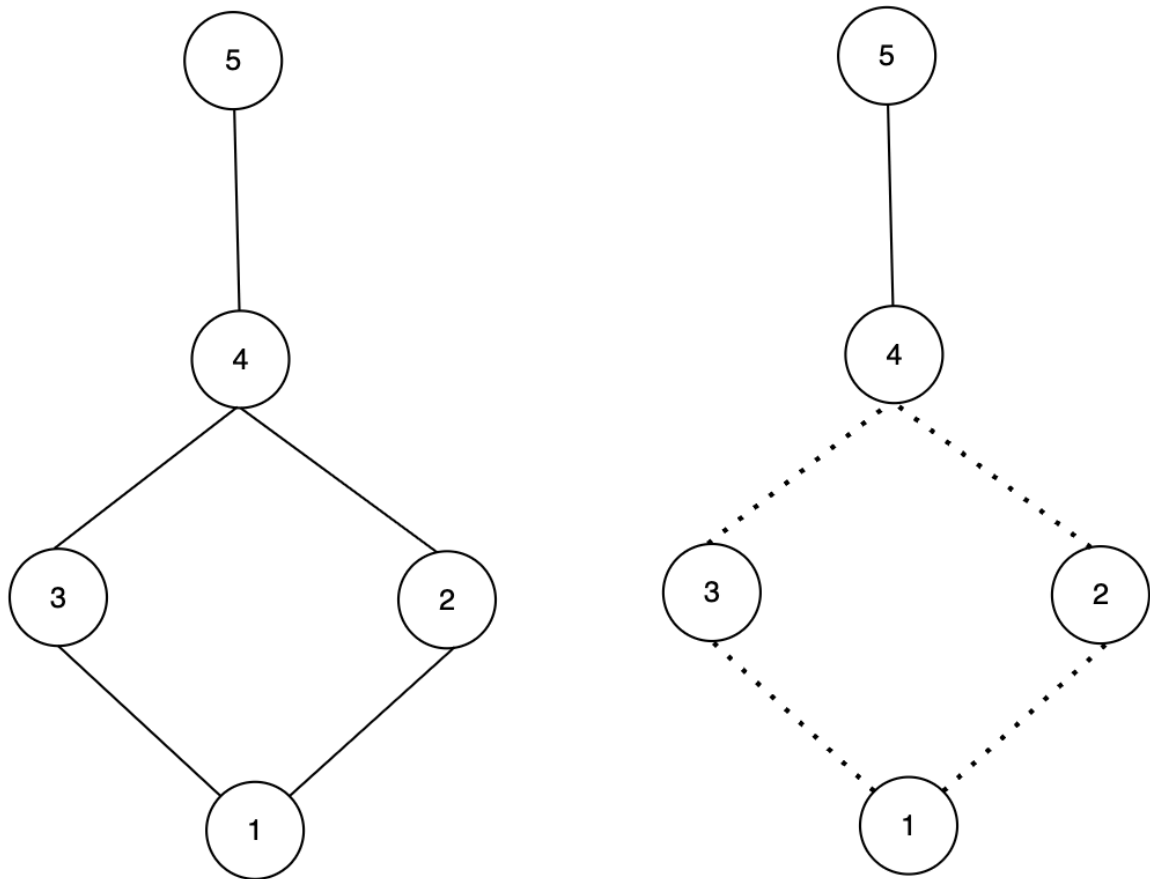
### ■ 题意

给定一个无向连通图，有  $n$  个城镇和  $m$  条双向道路。图中任意两个城镇都连通，且至少存在两条不同的路径。对于每个查询  $(a, b)$ ，需要计算从  $a$  到  $b$  的所有路径中必须经过的道路数量（即桥的数量）。

数据范围：  $1 \leq n \leq 10^5$ ,  $1 \leq m \leq 2 \times 10^5$ ,  $1 \leq q \leq 10^5$ 。

## 分析

本题需要求两点之间所有路径中的必经之路（桥）。由于图中任意两点都连通且不止一条路，说明图是边双连通分量的集合。必经之路实际上是连接不同边双连通分量的桥。



做法：

1. 使用 Tarjan 算法求出所有的边双连通分量，并进行缩点，将原图转化为一棵树（每个边双连通分量对应树的一个节点，桥对应树的边）。
2. 在树上，两点之间的唯一路径上的边数就是必经之路的数量。
3. 使用 LCA（最近公共祖先）算法快速计算树上两点间的距离。

时间复杂度：Tarjan 算法求边双连通分量的时间复杂度为  $O(n + m)$ ，LCA 预处理的时间复杂度为  $O(n \log n)$ ，每个查询的时间复杂度为  $O(\log n)$ ，总时间复杂度为  $O(n + m + q \log n)$ 。

## 参考代码

```
// 双向边
const int N = 1e5 + 10, M = 5e5 + 10;
int n, m, q;
int a, b;

// 链式前向星存储原图
struct edge {
    int v, ne;
};
int h[N], ne[M], to[M], tot = 1; // 判割边, 注意建边从 0/2 开始, 用于判反边

int low[N], dfn[N], cnt; // low数组, dfn数组, 时间戳计数器
int dotScc[N], dotCnt; // 点所属的边双连通分量编号, 边双连通分量计数器
int u[M], v[M]; // 存储原始边
int dep[N]; // 在树中的深度
int fa[N][21]; // LCA倍增数组
stack<int> stk; // Tarjan算法栈

/* 链式前向星加边 */
void add(int u, int v) { to[++tot] = v, ne[tot] = h[u], h[u] = tot; }

// 存储缩点后的树
vector<int> e[N];
```

```
/*
 * Tarjan算法求边双连通分量
 * x: 当前节点
 * lastEdge: 上一条边的编号 (用于判断反向边)
 */
void tarjan(int x, int lastEdge) {
    low[x] = dfn[x] = ++cnt;
    stk.push(x);

    for (int i = h[x]; i; i = ne[i]) {
        int v = to[i];
        // 如果v未被访问
        if (!dfn[v]) {
            tarjan(v, i);
            low[x] = min(low[x], low[v]);
        }
        // 如果v已被访问且不是反向边 (避免重复计算)
        else if (i != (lastEdge ^ 1))
            low[x] = min(low[x], dfn[v]);
    }

    // 发现边双连通分量的根节点
    if (dfn[x] == low[x]) {
        int y;
        dotCnt++; // 新的边双连通分量
        do {
            y = stk.top();
            stk.pop();
            dotScc[y] = dotCnt; // 标记节点属于哪个边双连通分量
        } while (x != y);
    }
}
```

```
/*
 * DFS预处理LCA
 * u: 当前节点
 * father: 父节点
 */
void dfs(int u, int father) {
    dep[u] = dep[father] + 1; // 更新深度
    fa[u][0] = father;       // 直接父节点

    // 预处理倍增数组
    for (int i = 1; i <= 20; i++)
        fa[u][i] = fa[fa[u][i - 1]][i - 1];

    // 递归处理子节点
    for (auto v : e[u]) {
        if (v == father)
            continue;
        dfs(v, u);
    }
}
```

```
/*
 * LCA算法求最近公共祖先
 * u, v: 需要求LCA的两个节点
 */
int lca(int u, int v) {
    // 确保u的深度不小于v
    if (dep[u] < dep[v])
        swap(u, v);

    // 将u提升到与v同一深度
    for (int j = 20; j >= 0; j--)
        if (dep[fa[u][j]] >= dep[v])
            u = fa[u][j];

    // 如果此时u==v, 说明v就是u的祖先
    if (u == v)
        return u;

    // 同时向上跳, 直到父节点相同
    for (int j = 20; j >= 0; j--) {
        if (fa[u][j] != fa[v][j]) {
            u = fa[u][j];
            v = fa[v][j];
        }
    }

    // 返回LCA
    return fa[u][0];
}
```

```
int main() {
    // 输入原图
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        cin >> u[i] >> v[i];
        add(u[i], v[i]), add(v[i], u[i]); // 添加双向边
    }

    // 对每个连通分量进行Tarjan算法求边双连通分量
    for (int i = 1; i <= n; i++)
        if (!dfn[i])
            tarjan(i, 0);

    // 构建缩点后的树 (边双连通分量树)
    for (int i = 1; i <= m; i++)
        // 如果边的两个端点不在同一个边双连通分量中
        if (dotScc[u[i]] != dotScc[v[i]]) {
            // 在缩点后的树中添加边
            e[dotScc[u[i]]].push_back(dotScc[v[i]]);
            e[dotScc[v[i]]].push_back(dotScc[u[i]]);
        }

    // 在缩点后的树上进行DFS预处理LCA
    dfs(1, 0);

    // 处理查询
    cin >> q;
    while (q--) {
        cin >> a >> b;
        // 计算在缩点树中两个节点之间的距离
        // 距离公式:  $dep[u] + dep[v] - 2 * dep[lca(u,v)]$ 
        cout << dep[dotScc[a]] + dep[dotScc[b]] -
            2 * dep[lca(dotScc[a], dotScc[b])]
            << endl;
    }

    return 0;
}
```

## P4877 [USACO14FEB] Cow Decathlon G

### ■ 题意

有  $N$  头奶牛和  $N$  项比赛，每头奶牛必须参加一项比赛，每项比赛也必须有一头奶牛参加。奶牛  $i$  参加比赛  $j$  会得到分数  $S_{i,j}$ 。此外，有  $B$  种奖励分，第  $i$  种奖励会在第  $K_i$  项比赛结束时检查，如果当时总分大于或等于  $P_i$ ，则立即获得额外  $A_i$  分。奖励分检查顺序可以自由安排。求能获得的最大总得分。

数据范围：  $1 \leq N, B \leq 20$ ,  $1 \leq K_i \leq N$ ,  $1 \leq P_i \leq 4 \times 10^4$ ,  $1 \leq A_i \leq 10^3$ ,  $1 \leq S_{i,j} \leq 10^3$ 。

## ■ 分析

由于  $N$  较小，可以使用状态压缩动态规划。用二进制位掩码表示已分配奶牛的集合，状态  $f[mask]$  表示分配奶牛集合  $mask$  后获得的最大分数。状态转移时，考虑下一个比赛（即当前已完成比赛数量  $k = \text{popcount}(mask)$ ）和未分配的奶牛，更新新状态。同时，在每个状态检查奖励分：如果  $k$  等于某个奖励的  $K_i$  且当前分数  $f[mask] \geq P_i$ ，则加上奖励分  $A_i$ 。

## 算法步骤:

1. 读入  $N$  和  $B$ , 存储奖励分。
2. 读入得分矩阵  $S$ 。
3. 初始化 DP 数组  $f$ , 大小为  $2^N$ ,  $f[0] = 0$ , 其余为负无穷。
4. 预处理每个状态  $mask$  的  $popcount$  (已完成比赛数量)。

### 5. 遍历所有状态 $mask$ :

- 若  $f[mask]$  无效则跳过。
- 计算当前已完成比赛数量  $k = \text{popcount}(mask)$ 。
- 应用奖励分: 遍历所有奖励, 若  $K_i = k$  且  $f[mask] \geq P_i$ , 则  $f[mask]_+ = A_i$ 。
- 若  $k = N$ , 则跳过转移 (已完成)。
- 否则, 对于每个未分配奶牛  $j$  (即  $mask$  中位  $j$  为 0), 计算新状态  $new\_mask = mask | (1 \ll j)$ , 新得分为  $f[mask] + S[j][k + 1]$ , 更新  $f[new\_mask]$ 。

### 6. 输出 $f[(1 \ll N) - 1]$ 。

时间复杂度:  $O(2^N \cdot N \cdot B)$ , 在  $N \leq 20$  时可行。

## ■ 参考代码

```
const int N = 21;
struct node {
    int k, p, a;
    // 按照 p 值小优先排
    bool operator<(const node &b) const { return p < b.p; }
} q[N];
/* f[i]: 集合选点的状态为 i 时, 即奶牛 i
   * 是否参加了比赛, 得到最大分值 (得分+奖励分) */
int n, b, f[1 << N], s[N][N], cnt[1 << N];
```

```
int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n >> b;
    for (int i = 1; i <= b; i++)
        cin >> q[i].k >> q[i].p >> q[i].a;
    sort(q + 1, q + 1 + b);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> s[i][j];
    for (int i = 0; i < (1 << n); i++) // 预处理集合选点状态, 1 的个数
        cnt[i] = __builtin_popcount(i);
    // 状态初始化, 选点个数为 0 的情况在, 最大分值为 0, 其他为无穷小
    memset(f, -0x3f, sizeof f);
    f[0] = 0;
```

```
for (int i = 0; i < (1 << n); i++) {
    if (f[i] < 0)
        continue;
    int conCnt = cnt[i];           // 已经完成比赛数量
    int val = f[i];               // 当前选点状态下的最大分值
    for (int j = 1; j <= b; j++) // 计算奖励分
        if (conCnt == q[j].k && val >= q[j].p)
            val += q[j].a;
    if (conCnt == n) // 全部比赛都参加完, 更新 f[i] 最大分值
    {
        f[i] = max(f[i], val);
        continue;
    }
    // 枚举每一头未被选择的牛, 去参加比赛并得分, 求不同方案下的最大分值
    for (int j = 1; j <= n; j++) {
        if (i & (1 << (j - 1)))
            continue;
        int nowstate = i | (1 << (j - 1));
        // 加入奶牛 j 后的选点状态
        int nowsum = val + s[j][conCnt + 1];
        // 现在的分值 + 奶牛 j 参加第 conCnt+1 赛项的分值就是总新得分
        if (nowsum > f[nowstate])
            f[nowstate] = nowsum;
    }
}
cout << f[(1 << n) - 1] << endl;
return 0;
}
```