Summer CSP-J 263 / CSP-S 268 题解

J 组

A. 环线巡游

- 35 分做法。
  - 根据 d_i 的正负逐步移动，越过展台 n 时回到展台 1，越过展台 1 时回到展台 n。
  - 时间复杂度为

$$O\left(m+\sum_{i=1}^{m}|d_i|\right),$$

可以通过子任务 $1, 4$。

- 20 分做法。
  - 当所有 $d_i \geq 0$ 时，直接计算

$$p \leftarrow (p+d_i) \bmod n$$

即可，其中 p 是从 0 开始的展台编号。

- 当 $|d_i| < n$ 时，也可以在每次移动后判断是否越界，至多加或减一次 n。
- 满分做法。
  - 将展台编号整体减一，把 $1 \sim n$ 转换成 $0 \sim n-1$。设当前位置为

$$p = s - 1.$$

- 在环上移动完整的 n 步会回到原位，所以一条指令只需要保留 $d_i \bmod n$:

$$p \leftarrow (p + d_i \bmod n) \bmod n.$$

- C++ 中负数取模的结果可能为负。例如 $-1 \bmod 7 = -1$，因此取模后如果 $p < 0$，还要加一次 n。
- 输出时再把零基编号转换回题目中的编号 $p+1$。
- 先对 d_i 取模，也可以避免把接近 10^{18} 的移动量不断累加到当前位置上。
- 时间复杂度为 $O(m)$，空间复杂度为 $O(1)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int64 n, s;
    int m;
    cin >> n >> m >> s;
```

```
    int64 p = s - 1;
    for (int i = 0; i < m; ++i) {
        int64 d;
        cin >> d;
        p = (p + d % n) % n;
        if (p < 0) p += n;

        if (i) cout << ' ';
        cout << p + 1;
    }
    cout << '\n';
    return 0;
}
```

B. 三人组队

- 25 分做法。
  - 当 n 较小时，可以使用动态规划。
  - 设 $f_{s,c}$ 表示能否用恰好 c 个 3 的幂凑出总和 s，从 $f_{0,0}=1$ 开始枚举下一支队伍的大小。
  - 时间复杂度约为 $O(nk\log n)$，空间复杂度为 $O(nk)$，可以通过子任务 $1,2$。
- 15 分做法。
  - 当 $k=1$ 时，判断 n 是否为 3 的幂。
  - 当 $k=2$ 时，枚举第一项 3^x，判断 $n-3^x$ 是否也是 3 的幂。
  - 可以通过子任务 3。
- 特殊性质 n 是 3 的幂。
  - 最开始只有一支大小为 n 的队伍。
  - 把一支大小为 3^x 的队伍拆成三支大小为 3^{x-1} 的队伍，队伍数量会增加 2。
  - 最终可以得到所有不超过 n 的奇数队伍数，可以通过子任务 4。
- 满分做法。
  - 将 n 写成标准三进制形式：

$$n=\sum_{x\ge 0} c_x 3^x,\qquad c_x\in\{0,1,2\}.$$

- 设三进制数位和为

$$s=\sum_{x\ge 0} c_x.$$

- 标准三进制展开本身就是一个使用 s 个三次幂的方案。
- 任意一种表示中，如果同一个 3^x 出现至少三次，就可以把三个 3^x 合成一个 3^{x+1}。每次合并会让项数减少 2，不断合并后一定得到标准三进制展开。
- 所以任何方案的项数都不能小于 s，并且与 s 的奇偶性相同。
- 反过来，从标准三进制展开开始，可以把一个 3^x 拆成三个 3^{x-1}，每次让项数增加 2。一直拆到全部为 1，项数会依次经过

$$s,s+2,s+4,\ldots,n.$$

- 因此答案为 `Yes` 当且仅当

$$s \le k \le n \quad \text{且} \quad (k-s) \bmod 2 = 0.$$

- 每组询问只需枚举 n 的三进制数位，时间复杂度为 $O(\log_3 n)$。总时间复杂度为 $O(T \log n)$，空间复杂度为 $O(1)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--) {
        int64 n, k;
        cin >> n >> k;

        int64 digitSum = 0;
        for (int64 x = n; x > 0; x /= 3) {
            digitSum += x % 3;
        }

        bool ok = digitSum <= k && k <= n && (k - digitSum) % 2 == 0;
        cout << (ok ? "Yes" : "No") << '\n';
    }
    return 0;
}
```

C. 共鸣实验

- 30 分做法。
  - 枚举所有 $i < j$，直接累加 $(a_i - a_j)^2$。
  - 时间复杂度为 $O(n^2)$，可以通过子任务 $1, 2$。
- 特殊性质 $a_i \in \{0, 1\}$。
  - 只有数值不同的数对会产生贡献。
  - 设 0 和 1 的数量分别为 c_0, c_1，答案就是 $c_0 c_1$。
- 特殊性质 $0 \le a_i \le 1000$。
  - 统计每个值的出现次数 c_x，再枚举两个不同的值：

$$\sum_{0 \le x < y \le 1000} c_x c_y (x-y)^2.$$

- 时间复杂度为 $O(n + V^2)$，其中 $V = 1001$，可以通过子任务 $3, 4$。
- 满分做法。
  - 将每对数的贡献展开：

$$(a_i - a_j)^2 = a_i^2 + a_j^2 - 2a_i a_j.$$

- 在所有无序数对中，每个 a_i^2 都会出现 $n-1$ 次，因此

$$\sum_{i<j}(a_i^2+a_j^2)=(n-1)\sum_{i=1}^{n}a_i^2.$$

- 又因为

$$\left(\sum_{i=1}^{n}a_i\right)^2=\sum_{i=1}^{n}a_i^2+2\sum_{i<j}a_ia_j,$$

所以有

$$\sum_{i<j}(a_i-a_j)^2=n\sum_{i=1}^{n}a_i^2-\left(\sum_{i=1}^{n}a_i\right)^2.$$

- 扫描序列时维护

$$S=\sum a_i,\qquad Q=\sum a_i^2,$$

最后计算 $nQ-S^2$ 即可。

- S 最大可以达到 10^{15}，不能先保存真实的 S 再平方。读入后应立即取模，并始终在模意义下维护 S,Q。
- 最终减法可能得到负数，需要再加一次模数。
- 时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;
const int64 MOD = 1000000007LL;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int64 n;
    cin >> n;

    int64 sum = 0;
    int64 squareSum = 0;
    for (int i = 0; i < n; ++i) {
        int64 x;
        cin >> x;
        x %= MOD;
        sum = (sum + x) % MOD;
        squareSum = (squareSum + x * x) % MOD;
    }

    int64 answer = ((n % MOD) * squareSum - sum * sum) % MOD;
    if (answer < 0) answer += MOD;
    cout << answer << '\n';
    return 0;
}
```

D. 分批运输

- 10 分做法。
  - 枚举相邻零件箱之间是否切开，共有 2^{n-1} 种划分。
  - 对每种划分依次检查每段的总重量能否被段号整除。
  - 时间复杂度为 $O(n2^n)$，可以通过子任务 1。
- 45 分做法。
  - 设前缀和为

$$S_i = \sum_{t=1}^{i} a_t, \qquad S_0 = 0.$$

- 令 $dp_{i,j}$ 表示将前 i 个零件箱合法划分成恰好 j 段的方案数。
- 如果最后一段为 $(p, i]$，则它合法当且仅当

$$S_i - S_p \equiv 0 \pmod{j}.$$

- 因此转移为

$$dp_{i,j} = \sum_{\substack{0 \le p < i \\ S_i - S_p \equiv 0 \pmod{j}}} dp_{p,j-1}.$$

- 直接枚举 i, j, p，时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$，可以通过子任务 $1, 2, 3$。
- 特殊性质 $a_i = 0$。
  - 任意一段的和都是 0，每个切分位置都可以独立选择。
  - 答案为 2^{n-1}，可以通过子任务 4。
- 特殊性质 $a_i = 1$。
  - 第 j 段的长度必须是 j 的正整数倍。
  - 设 $f_{j,s}$ 表示已经确定前 j 段，且总长度为 s 的方案数，则

$$f_{j,s} = f_{j,s-j} + f_{j-1,s-j}.$$

- 因为前 j 段的最小总长度为 $j(j+1)/2$，只需计算 $O(\sqrt{n})$ 层。
- 时间复杂度为 $O(n\sqrt{n})$，可以通过子任务 5。
- 满分做法。
  - 最后一段合法的条件可以改写为

$$S_p \equiv S_i \pmod{j}.$$

- 固定段数 j，从左到右枚举右端点 i，用 `bucket[r]` 维护所有满足 $S_p \bmod j = r$ 的 $dp_{p,j-1}$ 之和。
- 计算 $dp_{i,j}$ 前，先把新的切分位置 $p = i - 1$ 加入桶：

```
bucket[S[i - 1] mod j] += dp[i - 1][j - 1]
```

- 此时所有加入桶的切分点都满足 $p < i$，所以最后一段一定非空。随后直接查询：

```
dp[i][j] = bucket[S[i] mod j]
```

- 第 j 层只依赖第 $j-1$ 层，可以使用滚动数组。
- 初始状态为 $dp_{0,0}=1$，最终答案为

$$\sum_{j=1}^{n} dp_{n,j}.$$

- 每层扫描 $O(n)$ 个位置，共有 n 层，时间复杂度为 $O(n^2)$。
- 滚动数组、前缀和与当前余数桶的空间复杂度均为 $O(n)$。
- 前缀和最大约为 5×10^{12}，需要使用 `long long`。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;
const int MOD = 998244353;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    vector<int64> prefix(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        int64 x;
        cin >> x;
        prefix[i] = prefix[i - 1] + x;
    }

    vector<int> previous(n + 1, 0), current(n + 1, 0);
    previous[0] = 1;

    int answer = 0;
    for (int segments = 1; segments <= n; ++segments) {
        vector<int> bucket(segments, 0);
        fill(current.begin(), current.end(), 0);

        for (int i = segments; i <= n; ++i) {
            int previousRemainder = prefix[i - 1] % segments;
            bucket[previousRemainder] += previous[i - 1];
            if (bucket[previousRemainder] >= MOD) {
                bucket[previousRemainder] -= MOD;
            }

            int currentRemainder = prefix[i] % segments;
            current[i] = bucket[currentRemainder];
        }

        answer += current[n];
        if (answer >= MOD) answer -= MOD;
        previous.swap(current);
    }
```

```
    cout << answer << '\n';
    return 0;
}
```

S 组

A. 接力协作

- 30 分做法。
  - 对每名队员 i 枚举所有 $j \neq i$，直接计算

$$\min(x_i + y_j, x_j + y_i).$$

- 时间复杂度为 $O(n^2)$，可以通过子任务 1。
- 特殊性质 $y_i = 0$。
  - 此时合作耗时为 $\min(x_i, x_j)$。
  - 将所有 x_i 排序。对于排序后位于位置 p 的队员，左侧队员贡献自己的 x_j，右侧队员各贡献 x_i。
  - 使用前缀和即可在 $O(n \log n)$ 时间内求出全部答案，可以通过子任务 2。
- 特殊性质 $x_i - y_i$ 已经有序。
  - 比较两种分工：

$$x_i + y_j \leq x_j + y_i \iff x_i - y_i \leq x_j - y_j.$$

- 令 $d_i = x_i - y_i$。对于固定的 i：
  - 若 $j < i$，合作耗时为 $x_j + y_i$；
  - 若 $j > i$，合作耗时为 $x_i + y_j$。
- 分别维护 x, y 的前缀和即可在线性时间求解，可以通过子任务 3。
- 满分做法。
  - 将所有队员按照 $d_i = x_i - y_i$ 从小到大排序，并记录原编号。
  - 假设队员 i 排序后位于零基位置 p，那么答案为

$$ans_i = \sum_{j<p} x_j + p\,y_i + (n - p - 1)x_i + \sum_{j>p} y_j.$$

- 排序后建立 x 与 y 的前缀和，每名队员的答案都能在 $O(1)$ 时间内计算，最后再按原编号还原输出顺序。
- 当两个队员的 d 相等时，两种分工耗时相同，所以它们在排序中的相对顺序不影响答案。
- 排序比较器不能写成 `<=`，可以在 d 相同时按照原编号排序。
- 答案可能达到 10^{15}，前缀和和答案都需要使用 `long long`。
- 总时间复杂度为 $O(n \log n)$，空间复杂度为 $O(n)$。

```
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;

struct Athlete {
```

```cpp
    int64 x, y, difference;
    int originalIndex;
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    vector<int64> x(n), y(n);
    for (int64 &value : x) cin >> value;
    for (int64 &value : y) cin >> value;

    vector<Athlete> athletes(n);
    for (int i = 0; i < n; ++i) {
        athletes[i] = {x[i], y[i], x[i] - y[i], i};
    }

    sort(athletes.begin(), athletes.end(),
         [](const Athlete &a, const Athlete &b) {
        if (a.difference != b.difference) {
            return a.difference < b.difference;
        }
        return a.originalIndex < b.originalIndex;
    });

    vector<int64> prefixX(n + 1, 0), prefixY(n + 1, 0);
    for (int i = 0; i < n; ++i) {
        prefixX[i + 1] = prefixX[i] + athletes[i].x;
        prefixY[i + 1] = prefixY[i] + athletes[i].y;
    }

    vector<int64> answer(n);
    for (int i = 0; i < n; ++i) {
        int64 left = prefixX[i] + 1LL * i * athletes[i].y;
        int64 rightY = prefixY[n] - prefixY[i + 1];
        int64 right = 1LL * (n - i - 1) * athletes[i].x + rightY;
        answer[athletes[i].originalIndex] = left + right;
    }

    for (int i = 0; i < n; ++i) {
        if (i) cout << ' ';
        cout << answer[i];
    }
    cout << '\n';
    return 0;
}
```

B. 往返委托

- 10 分做法。
  - 当委托总数 $K = m + l \le 18$ 时，可以使用记忆化搜索。
  - 状态记录已经使用的委托集合、当前位置和当前时刻，再枚举下一份能够接受的委托。
  - 时间复杂度约为 $O(2^K K)$，可以通过子任务 1。
- 35 分做法。
  - 把每份委托看成一个点。如果委托 u 的到达地点等于委托 v 的出发地点，并且 u 的到达时刻不晚于 v 的开始时刻，就从 u 向 v 连边。
  - 因为每次移动至少需要一个单位时间，所以这张图按照时间方向是一张 DAG。
  - 在 DAG 上求最长路即可，时间复杂度为 $O(K^2)$，可以通过子任务 $1, 2$。
- 满分做法。
  - 显式建立所有委托之间的边会达到平方复杂度。接受下一份委托时，真正需要知道的只有：在它开始以前，到达其出发地点的方案最多已经接受了多少份委托。
  - 维护：
    - `center`：已经到达物流中心的方案中，接受委托数的最大值；
    - `station[i]`：已经到达服务站 i 的方案中，接受委托数的最大值。
  - 初始时调度员位于物流中心，所以 `center = 0`，所有 `station[i]` 都设为不可达。
  - 每份委托建立两个事件：
    1. 开始事件：读取出发地点的最优状态，计算这份委托的 dp；
    2. 到达事件：用这份委托的 dp 更新到达地点。
  - 对于出发委托 (a, x)：

$$dp = center + 1,$$

并在时刻 $x + t_a$ 用 dp 更新 `station[a]`。

- 对于返回委托 (b, y)，只有 `station[b]` 可达时才能接受：

$$dp = station[b] + 1,$$

并在时刻 $y + t_b$ 用 dp 更新 `center`。

- 将所有事件按时间排序。同一时刻必须先处理到达事件，再处理开始事件，这样才能表示“到达后立即接受下一份委托”。
- 同一时刻的开始事件不会互相转移，因为开始事件只计算本委托的 dp，至少到一个单位时间之后的到达事件才会更新地点状态。
- 最后一份委托不要求回到物流中心，所以答案是所有委托 dp 的最大值，不能只输出 `center`。
- 共有 $2K$ 个事件，时间复杂度为 $O(K \log K)$，空间复杂度为 $O(n + K)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;
const int NEG_INF = -1000000000;
```

```cpp
struct Job {
    int station;
    bool isDeparture;
    int best = NEG_INF;
};

struct Event {
    int64 time;
    int kind;  // 0 表示到达，1 表示开始
    int jobId;

    bool operator<(const Event &other) const {
        if (time != other.time) return time < other.time;
        return kind < other.kind;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m, l;
    cin >> n >> m >> l;

    vector<int64> travelTime(n + 1);
    for (int i = 1; i <= n; ++i) cin >> travelTime[i];

    int jobCount = m + l;
    vector<Job> jobs(jobCount);
    vector<Event> events;
    events.reserve(2 * jobCount);

    for (int id = 0; id < jobCount; ++id) {
        int station;
        int64 startTime;
        cin >> station >> startTime;

        jobs[id].station = station;
        jobs[id].isDeparture = id < m;

        events.push_back({startTime, 1, id});
        events.push_back({startTime + travelTime[station], 0, id});
    }

    sort(events.begin(), events.end());

    int center = 0;
    vector<int> station(n + 1, NEG_INF);
    int answer = 0;

    for (const Event &event : events) {
        Job &job = jobs[event.jobId];

        if (event.kind == 0) {
            if (job.best == NEG_INF) continue;
            if (job.isDeparture) {
```

```
                station[job.station] = max(station[job.station], job.best);
            } else {
                center = max(center, job.best);
            }
        } else if (job.isDeparture) {
            job.best = center + 1;
            answer = max(answer, job.best);
        } else if (station[job.station] != NEG_INF) {
            job.best = station[job.station] + 1;
            answer = max(answer, job.best);
        }
    }

    cout << answer << '\n';
    return 0;
}
```

C. 双机巡检

- 10 分做法。
  - 给两台机器人临时编号，设状态为某一阶段结束后两台机器人分别位于 x, y 的最小代价。
  - 枚举下一阶段由哪台机器人前往要求点，直接进行状态转移。
  - 朴素实现可以通过 $n, q \leq 30$ 的子任务 1。
- 35 分做法。
  - 记环上距离为

$$d(x, y) = \min(|x - y|, n - |x - y|).$$

- 在一个只有一个要求点 p 的阶段结束后，可以认为一台机器人停在 p，另一台机器人没有提前移动。
- 设

$$F_x = \text{一台机器人位于当前要求点，另一台位于 } x \text{ 的最小代价}.$$

- 设上一个单点要求为 p，新的要求为 y。有两种转移：
  1. 仍由位于 p 的机器人前往 y：

$$F'_x = F_x + d(p, y).$$

2. 由位于 x 的机器人前往 y，此时原来位于 p 的机器人变成另一台：

$$F'_p \leftarrow \min\left(F'_p, \min_x\{F_x + d(x, y)\}\right).$$

- 每次线性扫描全部 x 求最小值，时间复杂度为 $O(qn)$，可以通过子任务 $1, 2$。
- 特殊性质所有阶段都有两个要求点。
  - 每个阶段结束后，两台机器人的位置就是给出的两个点。
  - 设上一阶段位置为 $\{a, b\}$，当前阶段要求为 $\{u, v\}$，只需比较两种匹配：

$$\min\big(d(a, u) + d(b, v),\ d(a, v) + d(b, u)\big).$$

- 时间复杂度为 $O(q)$，可以通过子任务 3。

- 满分做法。
  - 把连续的单点阶段看成一个单点块，使用上面的 F_x 状态。
  - 当某个两点阶段要求 u, v 时，如果前面是一段单点块，最后一个单点要求为 p，增加的代价为

$$\min\Big(d(p,u)+\min_x\{F_x+d(x,v)\}, d(p,v)+\min_x\{F_x+d(x,u)\}\Big).$$

- 处理完两点阶段后，两台机器人的位置固定为 $\{u, v\}$，之前的 F 状态可以清空。
- 从固定位置 $\{a, b\}$ 进入第一个单点要求 p 时，初始状态为

$$F_a = d(b,p), \qquad F_b = d(a,p).$$

- 现在需要支持三种操作：
  1. 所有 F_x 加上同一个数；
  2. 查询 $\min_x\{F_x + d(x,y)\}$；
  3. 对单个位置的 F_x 做取最小值。
- 用一个全局变量 A 表示所有状态共同增加的代价，在线段树中保存

$$B_x = F_x - A.$$

- 这样全体加法只需修改 A，查询变为

$$A + \min_x\{B_x + d(x,y)\}.$$

- 在数轴上有

$$\min_x\{B_x + |x-y|\} = \min\Big(y + \min_{x\le y}(B_x - x), -y + \min_{x\ge y}(B_x + x)\Big).$$

- 因此在线段树中分别维护 $B_x - x$ 和 $B_x + x$ 的区间最小值。
- 为了处理环，把位置 x 同时复制到 x 和 $x+n$，查询时分别在 y 和 $y+n$ 上查询数轴距离，再取较小值。
- 清空单点块时，记录这一块中被修改过的叶子，只恢复这些叶子，不需要重建整棵线段树。
- 每个阶段只进行常数次查询或单点修改，时间复杂度为 $O((n+q)\log n)$，空间复杂度为 $O(n)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

using int64 = long long;

class DistanceStructure {
public:
    explicit DistanceStructure(int pointCount)
        : pointCount(pointCount), doubledCount(2 * pointCount) {
        size = 1;
        while (size < doubledCount) size <<= 1;
        minMinus.assign(2 * size, infinity());
        minPlus.assign(2 * size, infinity());
    }

    void chminPoint(int position, int64 value) {
        chminDoubled(position, value);
        chminDoubled(position + pointCount, value);
```

```
        minimumValue = min(minimumValue, value);
    }

    int64 queryCycle(int position) const {
        return min(queryLine(position), queryLine(position + pointCount));
    }

    int64 getMinimumValue() const {
        return minimumValue;
    }

    void clear() {
        for (int position : touched) {
            assignDoubled(position, infinity());
        }
        touched.clear();
        minimumValue = infinity();
    }

private:
    static constexpr int64 infinity() {
        return numeric_limits<int64>::max() / 4;
    }

    void chminDoubled(int position, int64 value) {
        int index = size + position - 1;
        int64 oldValue = minMinus[index] >= infinity() / 2
            ? infinity()
            : minMinus[index] + position;

        if (value >= oldValue) return;
        if (oldValue >= infinity() / 2) touched.push_back(position);

        minMinus[index] = value - position;
        minPlus[index] = value + position;
        pull(index >> 1);
    }

    void assignDoubled(int position, int64 value) {
        int index = size + position - 1;
        if (value >= infinity() / 2) {
            minMinus[index] = infinity();
            minPlus[index] = infinity();
        } else {
            minMinus[index] = value - position;
            minPlus[index] = value + position;
        }
        pull(index >> 1);
    }

    void pull(int index) {
        while (index > 0) {
            minMinus[index] = min(minMinus[index << 1], minMinus[index << 1 |
1]);
            minPlus[index] = min(minPlus[index << 1], minPlus[index << 1 | 1]);
            index >>= 1;
```

```
        }
    }

    int64 rangeMin(const vector<int64> &tree, int left, int right) const {
        int64 result = infinity();
        int l = size + left - 1;
        int r = size + right - 1;
        while (l <= r) {
            if (l & 1) result = min(result, tree[l++]);
            if (!(r & 1)) result = min(result, tree[r--]);
            l >>= 1;
            r >>= 1;
        }
        return result;
    }

    int64 queryLine(int position) const {
        int64 fromLeft = rangeMin(minMinus, 1, position) + position;
        int64 fromRight = rangeMin(minPlus, position, doubledCount) - position;
        return min(fromLeft, fromRight);
    }

    int pointCount;
    int doubledCount;
    int size;
    vector<int64> minMinus, minPlus;
    vector<int> touched;
    int64 minimumValue = infinity();
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, q;
    cin >> n >> q;

    auto distance = [n](int x, int y) -> int64 {
        int direct = abs(x - y);
        return min(direct, n - direct);
    };

    DistanceStructure structure(n);
    bool inSingleBlock = false;
    int fixedFirst = 1, fixedSecond = n;
    int lastRequest = 1;
    int64 fixedCost = 0;
    int64 blockAdd = 0;

    for (int stage = 0; stage < q; ++stage) {
        int count, first;
        cin >> count >> first;

        if (count == 1) {
            if (!inSingleBlock) {
                structure.chminPoint(fixedFirst, distance(fixedSecond, first));
```

```
                structure.chminPoint(fixedSecond, distance(fixedFirst, first));
                blockAdd = 0;
                lastRequest = first;
                inSingleBlock = true;
            } else {
                int64 switchRobot = blockAdd + structure.queryCycle(first);
                blockAdd += distance(lastRequest, first);
                structure.chminPoint(lastRequest, switchRobot - blockAdd);
                lastRequest = first;
            }
        } else {
            int second;
            cin >> second;

            if (!inSingleBlock) {
                fixedCost += min(
                    distance(fixedFirst, first) + distance(fixedSecond, second),
                    distance(fixedFirst, second) + distance(fixedSecond, first)
                );
            } else {
                int64 moveToFirst = distance(lastRequest, first)
                    + blockAdd + structure.queryCycle(second);
                int64 moveToSecond = distance(lastRequest, second)
                    + blockAdd + structure.queryCycle(first);
                fixedCost += min(moveToFirst, moveToSecond);

                structure.clear();
                blockAdd = 0;
                inSingleBlock = false;
            }

            fixedFirst = first;
            fixedSecond = second;
        }
    }

    if (inSingleBlock) {
        fixedCost += blockAdd + structure.getMinimumValue();
    }

    cout << fixedCost << '\n';
    return 0;
}
```

D. 瓶颈环

- 45 分做法。
  - 枚举不同的边权 W，保留所有边权不超过 W 的边。
  - 在当前子图中使用 Tarjan 算法求桥。无向图中的一条边属于某个简单环，当且仅当它不是桥。
  - 一个点只要与至少一条非桥边相连，就属于某个简单环。为第一次满足条件的点记录当前 W。
  - 设不同边权数量为 K，时间复杂度为 $O(K(n+m))$，可以通过子任务 $1, 2, 4$。

- 45 分做法。
  - 先用 Kruskal 求出一棵最小生成树。
  - 对于每条非树边 (u, v, w)，它与最小生成树上 u 到 v 的路径组成一个环。
  - 逐点遍历这条树上路径，用 w 更新路径上所有点的答案。
  - 设非树边数量为 $q = m - n + 1$，时间复杂度为 $O(m \log m + qn)$，可以通过子任务 $1, 3, 4$。
- 满分做法。
  - 用 Kruskal 求最小生成树 T。对于一条未加入生成树的边 $e = (u, v, w)$，它与树上 u 到 v 的路径组成一个基本环。
  - Kruskal 拒绝 e 时，u, v 已经被此前选中的树边连通，所以路径上的所有边权都不超过 w。基本环又包含边 e，因此这个环的瓶颈值恰好为 w。
  - 只处理这些基本环不会漏掉答案。假设点 x 位于某个瓶颈值为 W 的简单环中：
    - 如果环上与 x 相邻的边中有非树边，取这条边，它的树上端点路径经过 x；
    - 否则取一条与 x 相邻的树边 f。删除 f 后，原环的剩余部分仍然跨过这个割，其中必然存在一条非树边 e。边 e 的树上端点路径经过 f，也就经过 x。
  - 这条非树边位于原环上，所以 $w_e \le W$。因此每个简单环都能找到一个瓶颈值不更大的基本环覆盖同一个点。
  - 问题转化成：对于每条非树边 (u, v, w)，用 w 更新树上路径 $u \rightsquigarrow v$ 的所有点。
  - 非树边已经按照权值从小到大排列。一个点第一次被路径覆盖时，当前权值就是它的最小答案，之后不需要再访问。
  - 使用倍增预处理 LCA。处理一条路径时，令

$$l = \mathrm{LCA}(u, v),$$

分别从 u, v 向 l 处理，最后单独处理 l。

- 再维护一个并查集跳过已经赋值的点：
  - `find(x)` 表示从 x 开始沿父亲向上，第一个还没有获得答案的点；
  - 点 x 获得答案后，将它连到 `find(parent[x])`；
  - 以后再次访问到 x 时，就会直接跳到上面的未赋值点。
- 一侧路径的处理过程为：

```
x = find(endpoint)
while depth[x] > depth[l]:
    answer[x] = w
    erase x to find(parent[x])
    x = find(x)
```

- 两侧处理结束后，只有在 `find(l) == l` 时才能给 LCA 赋值。如果 LCA 已经被删除，`find(l)` 可能跳到本次路径以外的祖先，不能给这个祖先赋值。
- 每个点最多真正被赋值并删除一次，所以全部跳点操作的摊还复杂度为 $O((n + m)\alpha(n))$。
- Kruskal 排序需要 $O(m \log m)$，倍增预处理和 LCA 查询需要 $O((n + m) \log n)$。
- 总时间复杂度为

$$O(m \log m + (n + m) \log n),$$

空间复杂度为 $O(n \log n + m)$。

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int u, v, weight;
};

class DisjointSet {
public:
    explicit DisjointSet(int n) : parent(n + 1), size(n + 1, 1) {
        iota(parent.begin(), parent.end(), 0);
    }

    int find(int x) {
        return parent[x] == x ? x : parent[x] = find(parent[x]);
    }

    bool unite(int x, int y) {
        x = find(x);
        y = find(y);
        if (x == y) return false;
        if (size[x] < size[y]) swap(x, y);
        parent[y] = x;
        size[x] += size[y];
        return true;
    }

private:
    vector<int> parent, size;
};

class JumpSet {
public:
    explicit JumpSet(int n) : next(n + 1) {
        iota(next.begin(), next.end(), 0);
    }

    int find(int x) {
        int root = x;
        while (next[root] != root) root = next[root];
        while (next[x] != x) {
            int following = next[x];
            next[x] = root;
            x = following;
        }
        return root;
    }

    void eraseTo(int x, int parent) {
        next[x] = find(parent);
    }

private:
```

```
    vector<int> next;
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;

    vector<Edge> edges(m);
    for (Edge &edge : edges) {
        cin >> edge.u >> edge.v >> edge.weight;
    }

    sort(edges.begin(), edges.end(), [](const Edge &a, const Edge &b) {
        if (a.weight != b.weight) return a.weight < b.weight;
        if (a.u != b.u) return a.u < b.u;
        return a.v < b.v;
    });

    DisjointSet kruskal(n);
    vector<vector<int>> tree(n + 1);
    vector<Edge> nonTreeEdges;
    nonTreeEdges.reserve(m - n + 1);

    for (const Edge &edge : edges) {
        if (kruskal.unite(edge.u, edge.v)) {
            tree[edge.u].push_back(edge.v);
            tree[edge.v].push_back(edge.u);
        } else {
            nonTreeEdges.push_back(edge);
        }
    }

    int levels = 1;
    while ((1 << levels) <= n) ++levels;

    vector<vector<int>> up(levels, vector<int>(n + 1, 0));
    vector<int> depth(n + 1, 0);

    vector<int> order(1, 1);
    order.reserve(n);
    for (int index = 0; index < (int)order.size(); ++index) {
        int vertex = order[index];
        for (int neighbor : tree[vertex]) {
            if (neighbor == up[0][vertex]) continue;
            up[0][neighbor] = vertex;
            depth[neighbor] = depth[vertex] + 1;
            order.push_back(neighbor);
        }
    }

    for (int level = 1; level < levels; ++level) {
        for (int vertex = 1; vertex <= n; ++vertex) {
            up[level][vertex] = up[level - 1][up[level - 1][vertex]];
```

```
        }
    }

    auto lca = [&](int x, int y) {
        if (depth[x] < depth[y]) swap(x, y);

        int difference = depth[x] - depth[y];
        for (int level = 0; level < levels; ++level) {
            if ((difference >> level) & 1) x = up[level][x];
        }

        if (x == y) return x;

        for (int level = levels - 1; level >= 0; --level) {
            if (up[level][x] != up[level][y]) {
                x = up[level][x];
                y = up[level][y];
            }
        }
        return up[0][x];
    };

    vector<int> answer(n + 1, -1);
    JumpSet unassigned(n);

    auto paintUp = [&](int vertex, int ancestor, int weight) {
        int current = unassigned.find(vertex);
        while (depth[current] > depth[ancestor]) {
            answer[current] = weight;
            unassigned.eraseTo(current, up[0][current]);
            current = unassigned.find(current);
        }
    };

    for (const Edge &edge : nonTreeEdges) {
        int ancestor = lca(edge.u, edge.v);
        paintUp(edge.u, ancestor, edge.weight);
        paintUp(edge.v, ancestor, edge.weight);

        if (unassigned.find(ancestor) == ancestor) {
            answer[ancestor] = edge.weight;
            unassigned.eraseTo(ancestor, up[0][ancestor]);
        }
    }

    for (int vertex = 1; vertex <= n; ++vertex) {
        if (vertex > 1) cout << ' ';
        cout << answer[vertex];
    }
    cout << '\n';
    return 0;
}
```