A - 算法竞赛

题解

从 y_1 枚举到 y_2，若枚举的年份不在停办年份里则答案加一。复杂度 O(n + y_2 - y_1)。

参考代码

```
#include <bits/stdc++.h>
#define MAXS ((int) 1e4)
using namespace std;

int n, Y1, Y2, ans;

// vis[y] == true 表示年份 y 是停办年份
bool vis[MAXS + 10];

void solve() {
    memset(vis, 0, sizeof(vis));
    scanf("%d%d", &Y1, &n);
    for (int i = 1; i <= n; i++) {
        int x; scanf("%d", &x);
        vis[x] = true;
    }
    scanf("%d", &Y2);

    ans = 0;
    // 从 Y1 枚举到 Y2
    for (int i = Y1; i <= Y2; i++) {
        if (vis[i]) continue;
        // 当前年份不是停办年份则答案加一
        ans++;
    }
    printf("%d\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

B - 基站建设

题解

维护 f(i) 表示只考虑前 i 个位置，且第 i 个位置必须建设基站的最小总成本。考虑上一个建设基站的位置 j，得到 dp 方程

f(i) = min f(j) + a_i
j
由于题目要求每个区间里都至少要有一个基站，因此 [j + 1, i - 1] 之间不能存在一个完整的区间。
因此，对于每个 1 <= i <= n，我们计算 p_i 满足 [p_i, i] 之间不存在一个完整的区间，且 p_i 尽可能小，则 j >= p_{i - 1} - 1。所有 p_i 的值可以用双指针法求出（因为如果 [l, r] 里存在一个完整的区间，那么 [l' <= l, r' >= r] 里肯定也存在一个完整的区间，满足双指针法的特性）。

上述 dp 可以用 单调队列 优化到 O(n)。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 5e5)
using namespace std;

int n, A[MAXN + 10];

vector<int> B[MAXN + 10];
int LIM[MAXN + 10];
long long f[MAXN + 10];

int head, tail, q[MAXN + 10];

void solve() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &A[i]);
    // 为了方便得到最终答案，可以令 A[n + 1] = 0，然后要求 [n + 1, n + 1] 里建设一座基站，
    // 这样答案就是 f[n + 1] 了
    A[++n] = 0;

    int m; scanf("%d", &m);
    // B[i] 是一个 vector，
    // 里面的负数 -j 表示有一个需求区间是 [i, j]，
    // 里面是正数 j 表示有一个需求区间是 [j, i]，
    // 方便我们等下用双指针算 p_i
    for (int i = 1; i <= n; i++) B[i].clear();
    for (int i = 1; i <= m; i++) {
        int l, r; scanf("%d%d", &l, &r);
        B[l].push_back(-r);
        B[r].push_back(l);
    }
    B[n].push_back(-n);
    B[n].push_back(n);

    // now 记录了双指针区间 [j, i] 中有几个完整的需求区间
    int now = 0;
    for (int i = 1, j = 1; i <= n; i++) {
        // 双指针右端点移动一步，增加右端点为 i 且位于 [j, i] 里的需求区间
        for (int x : B[i]) if (x > 0 && x >= j) now++;
        // 求出 j = p_i + 1
        while (now > 0 && j <= i) {
            // 双指针左端点移动一步，减少左端点为 j 且位于 [j, i] 里的需求区间
            for (int x : B[j]) if (x < 0 && -x <= i) now--;
            j++;
        }
```

```
        assert(now == 0);
        // LIM[i] = p_i
        LIM[i] = j;
    }

    // dp 初值
    f[0] = 0;
    f[1] = A[1];
    // 用 dp 初值初始化单调队列
    head = tail = 1;
    q[tail++] = 0;
    q[tail++] = 1;

    for (int i = 2; i <= n; i++) {
        // 要求上一个基站的位置 >= p_{i - 1} - 1
        int lim = LIM[i - 1] - 1;
        while (q[head] < lim) head++;
        f[i] = f[q[head]] + A[i];
        while (head < tail && f[q[tail - 1]] >= f[i]) tail--;
        q[tail++] = i;
    }
    printf("%lld\n", f[n]);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

C - 市场交易

题解

每次应该从价格最便宜的商店购买货物，并卖给价格最贵的商店。用双指针模拟这一贪心策略即可，具体实现详见参考代码。

复杂度 O(n log n)，主要是排序的复杂度。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 1e5)
using namespace std;
typedef pair<int, int> pii;

int n;
pii A[MAXN + 10];
long long ans;

void solve() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d%d", &A[i].first, &A[i].second);
    // 将商店按价格从低到高排序
```

```
    sort(A + 1, A + n + 1);

    ans = 0;
    // 维护两个指针，i 指向最便宜的商店，j 指向最贵的商店
    for (int i = 1, j = n; i < j; ) {
        // 交易的次数为两个商店交易次数的最小值
        int mn = min(A[i].second, A[j].second);
        // 计算利润
        ans += 1LL * (A[j].first - A[i].first) * mn;
        // 减少两个商店的交易次数
        A[i].second -= mn;
        A[j].second -= mn;
        // 如果商店的交易次数用完了，则指针指向下一个商店
        if (A[i].second == 0) i++;
        if (A[j].second == 0) j--;
    }
    printf("%lld\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

D - 新居规划

题解

如果已知 k（2 <= k <= n）个人有邻居，剩下的人没有邻居，怎样选择有邻居的人才能使总满意度最大化?

这是一个经典问题。先假设所有人都是没邻居的，得到总满意度

$$\sum_{i=1}^{n} b_i$$

当第 i 个人从没邻居变成有邻居时，总满意度将增加 (a_i - b_i)。因此选择 (a_i - b_i) 最大的 k 个人变成有邻居的即可。排序后可以在 O(n) 的复杂度内一次性算出 k = 2, … , n 的最大总满意度。

如果 k 个人有邻居，剩下的人没有邻居，这样的布局至少需要 k + 2(n - k) = 2n - k 栋房子（即有邻居的人都住在最左边，然后每隔一栋房子住一个没邻居的人）。因此只有满足 2n - k <= m 才能考虑。

最后，别忘了考虑所有人都没有邻居的情况。这要求 m >= 2n - 1。

复杂度 O(n log n)。主要是排序的复杂度。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 5e5)
using namespace std;

int n, m, A[MAXN + 10], B[MAXN + 10];

```

```
void solve() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) scanf("%d%d", &A[i], &B[i]);

    // 将 (A[i] - B[i]) 排序，vector 里存的是从小到大的顺序
    vector<int> vec;
    for (int i = 1; i <= n; i++) vec.push_back(A[i] - B[i]);
    sort(vec.begin(), vec.end());

    long long ans = 0, now = 0;
    for (int i = 1; i <= n; i++) now += B[i];
    // 特殊情况：所有人都没有邻居
    if (m >= 2 * n - 1) ans = now;

    now += vec[n - 1];
    for (int i = 2; i <= n; i++) {
        // 计算有 i 个人有邻居时的最大总满意度
        now += vec[n - i];
        if (2 * n - i <= m) ans = max(ans, now);
    }
    printf("%lld\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

E - 新怀质问

题解

本题名称来自日文“新しい、でも懐かしい質問”（意为新的，但是令人怀念的问题），因为有人说命题人的题目都很有年代感...

考虑从左到右确定答案的每一位。我们举个例子来介绍这一过程。

假设我们已经确定了答案的前三位是 abc，接下来要确定第四位。为了让字典序尽量小，我们需要从 a 到 z 枚举第四位。

假设我们枚举到了 d，我们考虑已知答案为 abcd* 时，与已知答案为 abc* 时相比，能选择的字符串数量如何变化。

- 所有 abc[a-d]* 都能选择，因为它们的最长公共前缀肯定小于等于 abcd*。
- 所有 abc[e-z]* 的字符串，原来答案是 abc* 的时候都能选择，现在每种字母只能选一个，否则比如 abce* 选了两个，那答案就至少是 abce > abcd* 了。
- 剩下的字符串可选情况维持不变。
  如果我们能选至少 k 个字符串，那么答案的第四位就是 d，否则我们要继续枚举 e、f、...

确定了答案的第四位以后，我们还要确定答案是不是只有四位。考虑已知答案为 abcd 时，与已知答案为 abcd* 时相比，能选的字符串数量如何变化。

- 所有 abcd[a-z]* 的字符串，原来答案是 abcd* 时都能选择，现在每种字母只能选一个，否则答案大于 abcd。

- 剩下的字符串可选情况维持不变。
  如果我们能选至少 k 个字符串，那么答案就只有四位，否则继续枚举第五位。复杂度 O(26 * ∑|s|)。

参考代码

```
#include <bits/stdc++.h>
#define MAXLEN ((int) 1e6)
using namespace std;

int n, K;
char s[MAXLEN + 10];

// sz[i]：有几个字符串恰好被节点 i 代表
// tot[i]：以节点 i 为根的子树里一共有几个字符串
int nCnt, sz[MAXLEN + 10], tot[MAXLEN + 10], ch[MAXLEN + 10][26];

// 新建一个 trie 节点，返回节点编号
int newNode() {
    nCnt++;
    sz[nCnt] = tot[nCnt] = 0;
    memset(ch[nCnt], 0, sizeof(ch[nCnt]));
    return nCnt;
}

// 将 s 里保存的字符串加入 trie
void add() {
    int now = 1;
    tot[now]++;
    for (int i = 1; s[i]; i++) {
        int &c = ch[now][s[i] - 'a'];
        if (!c) c = newNode();
        tot[now = c]++;
    }
    sz[now]++;
}

void solve() {
    // 新建 trie 根节点，编号为 1
    nCnt = 0; newNode();

    scanf("%d%d", &n, &K);
    // 将每个字符串加入 trie
    for (int i = 1; i <= n; i++) {
        scanf("%s", s + 1);
        add();
    }

    // 逐位确定答案，now 表示目前已确定的答案前缀在 trie 上是哪个节点
    int now = 1;
    while (true) {
        // 判断答案是否就是 now 所在的节点
        // 首先，now 代表的字符串一定都可以选
        int t = sz[now];
        // 以 now 子节点为根的子树里，每棵子树最多选一个字符串
```

```
        for (int i = 0; i < 26; i++) if (tot[ch[now][i]]) t++;
        if (t >= K) {
            // 答案就是 now 所在的节点
            if (now == 1) printf("EMPTY");
            printf("\n");
            return;
        }

        // 枚举答案的下一个字母
        for (int i = 0; i < 26; i++) if (tot[ch[now][i]]) {
            // 现在以 now 子节点为根的子树里，每棵子树可以选任意个字符串
            t = t - 1 + tot[ch[now][i]];
            if (t >= K) {
                // 确定了下一个字母为 i
                K -= t - tot[ch[now][i]];
                now = ch[now][i];
                printf("%c", i + 'a');
                break;
            }
        }
    }
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

F - 格子旅行

题解

显然旅行的范围是包含起点的连续区间。每次询问，我们通过二分找出旅行的左右端点，然后询问该区间的权值之和即可。

为了通过二分找出旅行的端点，我们需要快速求出一个区间里所有的颜色是否都在 A 里。也就是说，求 A 中所有颜色在区间中出现次数之和，是否等于区间长度。我们维护一个线段树 / 树状数组，每个节点保存一个哈希表，表示该区间中出现了哪些颜色，以及每种颜色出现了几次即可。

如果先二分答案，然后再算颜色出现次数之和的复杂度是 $O(\sum k \log^2 n)$ 的，需要较小的常数才能通过本题。正确的做法应该是 直接在线段树 / 树状数组上进行二分 / 倍增，复杂度 $O(\sum k \log n)$。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 3e5)
using namespace std;

// 比 unordered_map 更快的哈希表
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
const int RANDOM =
chrono::high_resolution_clock::now().time_since_epoch().count();
```

```
struct chash {
    int operator()(int x) const { return x ^ RANDOM; }
};
typedef gp_hash_table<int, int, chash> hash_t;

int n, q, C[MAXN + 10], V[MAXN + 10];

hash_t colTree[MAXN + 10];
long long smTree[MAXN + 10];

int lb(int x) { return x & (-x); }

// val == -1：把位置 pos 的颜色 c 删掉
// val == 1：把位置 pos 的颜色设为 c
void addCol(int pos, int c, int val) {
    for (; pos <= n; pos += lb(pos)) colTree[pos][c] += val;
}

// 查询 vec 里的所有颜色在前 pos 个位置中一共出现了几次
int queryCol(int pos, vector<int> &vec) {
    int ret = 0;
    for (; pos; pos -= lb(pos)) for (int c : vec) {
        auto it = colTree[pos].find(c);
        if (it != colTree[pos].end()) ret += it->second;
    }
    return ret;
}

// 树状数组上倍增，
// 返回值 l 满足 vec 里所有颜色在区间 [l, lim] 中出现的总次数等于区间长度，且 l 最小
int gao1(int lim, vector<int> &vec) {
    int base = queryCol(lim, vec);
    if (base == lim) return 1;

    int b;
    for (b = 1; b <= n; b <<= 1);

    int now = 0, cnt = 0;
    for (b >>= 1; b; b >>= 1) {
        int nxt = now | b, tmp = 0;
        for (int c : vec) {
            auto it = colTree[nxt].find(c);
            if (it != colTree[nxt].end()) tmp += it->second;
        }
        if (nxt > lim || base - (cnt + tmp) == lim - nxt) {
            // do nothing
        } else {
            now = nxt; cnt += tmp;
        }
    }
    return now + 2;
}

// 树状数组上倍增，
// 返回值 r 满足 vec 里所有颜色在区间 [lim, r] 中出现的总次数等于区间长度，且 r 最大
```

```
int gao2(int lim, vector<int> &vec) {
    int base = queryCol(lim, vec);

    int b;
    for (b = 1; b <= n; b <<= 1);

    int now = 0, cnt = 0;
    for (b >>= 1; b; b >>= 1) {
        int nxt = now | b, tmp = 0;
        for (int c : vec) {
            auto it = colTree[nxt].find(c);
            if (it != colTree[nxt].end()) tmp += it->second;
        }
        if (nxt < lim || (cnt + tmp) - base == nxt - lim) {
            now = nxt; cnt += tmp;
        } else {
            // do nothing
        }
    }
    return now;
}

// 位置 pos 的权值增加 val
void addSm(int pos, long long val) {
    for (; pos <= n; pos += lb(pos)) smTree[pos] += val;
}

// 求前 pos 个位置的权值之和
long long querySm(int pos) {
    long long ret = 0;
    for (; pos; pos -= lb(pos)) ret += smTree[pos];
    return ret;
}

void solve() {
    scanf("%d%d", &n, &q);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &C[i]);
        addCol(i, C[i], 1);
    }
    for (int i = 1; i <= n; i++) {
        scanf("%d", &V[i]);
        addSm(i, V[i]);
    }

    while (q--) {
        int op, x, y; scanf("%d%d%d", &op, &x, &y);
        if (op == 1) {
            addCol(x, C[x], -1);
            addCol(x, y, 1);
            C[x] = y;
        } else if (op == 2) {
            addSm(x, y - V[x]);
            V[x] = y;
        } else {
```

```
            bool ok = false;
            vector<int> vec;
            for (int i = 1; i <= y; i++) {
                int z; scanf("%d", &z);
                if (C[x] == z) ok = true;
                vec.push_back(z);
            }
            if (!ok) { printf("0\n"); continue; }

            int L = gao1(x, vec), R = gao2(x, vec);
            printf("%lld\n", querySm(R) - querySm(L - 1));
        }
    }

    for (int i = 1; i <= n; i++) colTree[i].clear();
    for (int i = 1; i <= n; i++) smTree[i] = 0;
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

G - 交换操作

题解

假设已经确定了分界点 k，考虑交换哪两个数才有意义。

令 f(i, j) = a_i & a_{i + 1} & ... & a_j，称满足 f(1, i) ≠ f(1, i - 1) 的下标 i 为"前缀关键点"。可以发现，关键点只有 log a_i 个。同理，称满足 f(i, n) ≠ f(i + 1, n) 的下标 i 为"后缀关键点"，关键点也只有 log a_i 个。

因此，交换可以被分为三类情况。

- 交换两个非关键点：如果从一个前缀里拿走一个非关键点，这个前缀的 & 值不会改变；同理，如果从一个后缀里拿走一个非关键点，这个后缀的 & 值也不会改变。交换后，相当于原来前（后）缀的 & 值又多 & 了一个数。由于多 & 一个数不会让值变大，因此这样的交换没有意义。
- 交换两个关键点：前后缀关键点分别只有 log a_i 种，直接枚举即可。这一类的总复杂度为 O(n log^2 a_i)。
- 交换一个关键点和一个非关键点：不妨假设交换的是前缀关键点 i 和后缀非关键点 j。与第一类情况的分析类似，因为从后缀拿走的是非关键点，所以交换之后，后缀的 & 值为 f(k + 1, n) & a_i。也就是说，只要选定了 i，无论选哪个 j 都不影响后缀的 & 值，那么我们选择让交换之后，前缀的 & 值最大的 j 即可。即最大化 f(1, i - 1) & f(i + 1, k) & a_j。

注意到对于一个固定的关键点 i，f(i + 1, k) 的值只有 log a_i 种，而关键点 i 也只有 log a_i 种，那么 v = f(1, i - 1) & f(i + 1, k) 的值只有 log^2 a_i 种。因此对于每种 v，O(n) 计算 g(v, k) 表示 k + 1 <= j <= n 里，v & a_j 的最大值即可。这一类的总复杂度也为 O(nlog^2 a_i)。

因此本题复杂度 O(nlog^2 a_i)，涉及到对 f 值的计算可以通过 RMQ 在 O(n log n) 的复杂度内预处理，后续每次查询只要 O(1) 的复杂度。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 1e5)
#define MAXP 18
using namespace std;

int n, ans, A[MAXN + 10];

int rmq[MAXP][MAXN + 10], lg[MAXN + 10];
unordered_map<int, vector<int>> f, g;

// 询问 A[l] & A[l + 1] & ... & A[r]
int query(int l, int r) {
    if (l > r) return (1 << 30) - 1;
    int p = lg[r - l + 1];
    return rmq[p][l] & rmq[p][r - (1 << p) + 1];
}

// 对于特定的 val，求满足 j <= lim 的 A[j] 中，val & A[j] 的最大值
int gaoPre(int val, int lim) {
    if (f.count(val) == 0) {
        // 根据题解分析，val 只有 log^2 种，所以下面的 for 枚举只会进行 log^2 次
        vector<int> &vec = f[val];
        vec.resize(n + 2);
        for (int i = 1; i <= n; i++) vec[i] = max(vec[i - 1], val & A[i]);
    }
    return f[val][lim];
}

// 对于特定的 val，求满足 j >= lim 的 A[j] 中，val & A[j] 的最大值
int gaoSuf(int val, int lim) {
    if (g.count(val) == 0) {
        // 根据题解分析，val 只有 log^2 种，所以下面的 for 枚举只会进行 log^2 次
        vector<int> &vec = g[val];
        vec.resize(n + 2);
        for (int i = n; i > 0; i--) vec[i] = max(vec[i + 1], val & A[i]);
    }
    return g[val][lim];
}

void solve() {
    scanf("%d", &n);
    // 读入数据并预处理 rmq
    for (int i = 1; i <= n; i++) scanf("%d", &A[i]), rmq[0][i] = A[i];
    for (int i = 1, half = 1; i < MAXP; i++, half *= 2) for (int j = 1; j + half
* 2 - 1 <= n; j++)
        rmq[i][j] = rmq[i - 1][j] & rmq[i - 1][j + half];
    lg[1] = 0;
    for (int i = 2; i <= n; i++) lg[i] = lg[i >> 1] + 1;

    // 计算前后缀关键点，分别保存在 pre 和 suf 里
    vector<int> pre, suf;
    int now = (1 << 30) - 1;
    for (int i = 1; i <= n; i++) {
        int nxt = now & A[i];
        if (now != nxt) pre.push_back(i);
```

```
        now = nxt;
    }
    now = (1 << 30) - 1;
    for (int i = n; i > 0; i--) {
        int nxt = now & A[i];
        if (now != nxt) suf.push_back(i);
        now = nxt;
    }

    // 计算不进行任何交换的答案
    ans = 0;
    for (int k = 1; k < n; k++) ans = max(ans, query(1, k) + query(k + 1, n));

    // 枚举交换两个关键点的情况
    for (int k = 1; k < n; k++)
        for (int i : pre) if (i <= k)
            for (int j : suf) if (j > k)
                ans = max(ans, (query(1, i - 1) & A[j] & query(i + 1, k)) +
(query(k + 1, j - 1) & A[i] & query(j + 1, n)));

    // 枚举交换前缀关键点 + 后缀非关键点的情况
    g.clear();
    for (int k = 1; k < n; k++)
        for (int i : pre) if (i <= k) {
            int val = query(1, i - 1) & query(i + 1, k);
            // 对于特定的 val，求满足 j >= k + 1 的 A[j] 中，val & A[j] 的最大值
            int best = gaoSuf(val, k + 1);
            ans = max(ans, best + (query(k + 1, n) & A[i]));
        }

    // 枚举交换后缀关键点 + 前缀非关键点的情况
    f.clear();
    for (int k = 1; k < n; k++)
        for (int j : suf) if (j > k) {
            int val = query(k + 1, j - 1) & query(j + 1, n);
            // 对于特定的 val，求满足 j <= k 的 A[j] 中，val & A[j] 的最大值
            int best = gaoPre(val, k);
            ans = max(ans, best + (query(1, k) & A[j]));
        }

    printf("%d\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

H - 流画溢彩

题解

本题 idea 来自同名桌游《流画溢彩》。

因为后面的操作会覆盖前面的操作，思考起来比较麻烦。我们不妨把操作顺序倒过来，这样一旦某一列的值确定了，后续操作就再也不会更改这一列的值。以下题解中操作的"先后"，是按操作顺序逆转之后的"先后"来说的。

首先，如果某个操作把两个数都改成 2，那么这个操作肯定最优，最先考虑；相应地，如果某个操作把两个数都改成 1，那么这个操作肯定最差，最后考虑。剩下的就是一个数改成 1，一个数改成 2 的操作。

本题的关键在于把题目转化为图论问题。把每一列看成一个点，把每个操作从改成 1 的那一列向改成 2 的那一列连一条有向边。

考虑选择一条边 $u \to v$，进行它代表的操作。此时 a_u 的值将被锁定为 1，而 u 能直接或间接到达的所有点的值将被锁定为 2（只要从 u 出发，按 dfs 序走一遍 u 能到达的边即可）。我们要做的就是让锁定为 1 的列尽量少，也就是选择尽量少的点，让它们能到达的点的并集等于整张图的点集。

容易发现，将强连通分量缩点以后，我们将得到一个有向无环图。有向无环图上每一个入度为 0 的点我们都必须选择，才能让它们本身以及它们的后继覆盖整张图。也就是说，我们每次从一个入度为 0 的强连通分量中选择一个点，按 dfs 顺序输出边的编号即可。

这里有一个细节：如果一个入度为 0 的强连通分量被一个 (l_i, 2, r_i, 2) 操作所影响，那么应该选择被影响的点为 dfs 的起始点，因为被影响的点代表的列已经被锁定为 2。

复杂度 $O(n + m)$。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 5e5)
#define MAXM ((int) 5e5)
using namespace std;

int n, m, OP[MAXM + 10][4];
vector<int> ans;

vector<int> e[MAXN + 10], v[MAXN + 10];
// bel[i]：点 i 属于哪个强连通分量
int clk, bCnt, dfn[MAXN + 10], low[MAXN + 10], bel[MAXN + 10];
bool inStk[MAXN + 10];
stack<int> stk;

// deg[i]：强连通分量 i 的入度
int deg[MAXN + 10];
// vis[i]：dfs 构造答案的过程中，节点 i 是否被访问过
bool vis[MAXN + 10];

// tarjan 求强连通分量
void tarjan(int sn) {
    low[sn] = dfn[sn] = ++clk;
    stk.push(sn); inStk[sn] = true;
    for (int fn : e[sn]) {
        if (!dfn[fn]) {
            tarjan(fn);
            low[sn] = min(low[sn], low[fn]);
        } else if (inStk[fn]) {
            low[sn] = min(low[sn], dfn[fn]);
        }
```

```
    }
    if (dfn[sn] == low[sn]) {
        ++bCnt;
        while (stk.top() != sn) {
            bel[stk.top()] = bCnt;
            inStk[stk.top()] = false;
            stk.pop();
        }
        bel[stk.top()] = bCnt;
        inStk[stk.top()] = false;
        stk.pop();
    }
}

// 从节点 sn 开始 dfs，并按 dfs 序将访问过的每条边加入 vec 里
void dfs(int sn, vector<int> &vec) {
    vis[sn] = true;
    for (int i = 0; i < e[sn].size(); i++) {
        int fn = e[sn][i];
        int val = v[sn][i];
        vec.push_back(val);
        if (!vis[fn]) dfs(fn, vec);
    }
}

void solve() {
    scanf("%d%d", &n, &m);

    vector<int> one, two;
    for (int i = 1; i <= n; i++) e[i].clear(), v[i].clear();
    for (int i = 1; i <= m; i++) {
        for (int j = 0; j < 4; j++) scanf("%d", &OP[i][j]);
        if (OP[i][1] == 1 && OP[i][3] == 1) {
            // 两个数都改成 1，最差的操作
            one.push_back(i);
        } else if (OP[i][1] == 2 && OP[i][3] == 2) {
            // 两个数都改成 2，最好的操作
            two.push_back(i);
        } else if (OP[i][1] == 1) {
            // 图中加一条从 1 指向 2 的边
            e[OP[i][0]].push_back(OP[i][2]);
            v[OP[i][0]].push_back(i);
        } else {
            // 图中加一条从 1 指向 2 的边
            e[OP[i][2]].push_back(OP[i][0]);
            v[OP[i][2]].push_back(i);
        }
    }

    // 顺序输出的答案中，两个数都改成 1 的最差操作最先输出
    ans.clear();
    for (int x : one) ans.push_back(x);

    memset(dfn, 0, sizeof(int) * (n + 3));
    clk = bCnt = 0;
```

```
    for (int sn = 1; sn <= n; sn++) if (!dfn[sn]) tarjan(sn);

    memset(deg, 0, sizeof(int) * (n + 3));
    for (int sn = 1; sn <= n; sn++) {
        for (int fn : e[sn]) if (bel[sn] != bel[fn]) deg[bel[fn]]++;
    }

    vector<int> vec;
    memset(vis, 0, sizeof(bool) * (n + 3));
    // 如果一个入度为 0 的强连通分量受一个 (l_i, 2, r_i, 2) 操作影响，那么需要从 l_i 或
r_i 开始 dfs
    for (int x : two) for (int j = 0; j < 4; j += 2) {
        int sn = OP[x][j];
        if (deg[bel[sn]] == 0 && !vis[sn]) dfs(sn, vec);
    }
    // 剩下的入度为 0 的强连通分量，随便找一个点开始 dfs
    for (int sn = 1; sn <= n; sn++) {
        if (deg[bel[sn]] == 0 && !vis[sn]) dfs(sn, vec);
    }
    // dfs 序是答案的倒序
    reverse(vec.begin(), vec.end());
    ans.insert(ans.end(), vec.begin(), vec.end());

    // 顺序输出的答案中，两个数都改成 2 的最好的操作最后输出
    for (int x : two) ans.push_back(x);

    // 计算序列最终的和
    unordered_map<int, int> mp;
    for (int x : ans) {
        mp[OP[x][0]] = OP[x][1];
        mp[OP[x][2]] = OP[x][3];
    }
    int tot = 0;
    for (auto it = mp.begin(); it != mp.end(); it++) tot += it->second;
    printf("%d\n", tot);
    for (int i = 0; i < m; i++) printf("%d%c", ans[i], "\n "[i + 1 < m]);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

I - 路径规划

题解

假设答案为 x，那么存在一条路径，使得从 0 到 (x - 1) 的每个整数都在路径上。这一条件满足二分性，因此我们可以二分答案 x，并检查是否存在这样的路径。

由于每一步只能往右或者往下走，因此将路径上每个格子的坐标按行为第一关键字，列为第二关键字排序后，排在前面的坐标的列编号，一定小于等于排在后面的坐标的列编号。

因此，将从 0 到 (x - 1) 的每个整数所在的格子的坐标排序，并检查列编号是否满足以上条件，即可判断是否存在一条路径，使得这些整数都在路径上。实际实现时，不需要使用排序函数。只要依此枚举每个格子，若格子里的整数小于 x 则把格子加入 vector，这样得到的 vector 就已经按枚举的顺序排序了。

复杂度 O(nmlog(nm))。

参考代码

```
#include <bits/stdc++.h>
#define MAXPROD ((int) 1e6)
using namespace std;

int n, m, A[MAXPROD];

// 将从 0 到 LIM - 1 所在的格子坐标“排序”，检查前面的列坐标是否小于等于后面的列坐标
bool check(int LIM) {
    // 实际实现时，不需要使用排序函数，
    // 直接按顺序枚举每个格子，若格子里的整数小于 $x$ 则把格子加入 vector，
    // 这样得到的 vector 就已经按枚举的顺序排序了
    //
    // 而且甚至连 vector 也不用真的维护，
    // 因为我们只关心 vector 最后一个元素的列坐标，和当前列坐标的大小关系，
    // 直接用变量 last 维护最后一个元素的列坐标即可
    int last = 0;
    for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) if (A[i * m + j] <
LIM) {
        if (last > j) return false;
        last = j;
    }
    return true;
}

void solve() {
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) scanf("%d", &A[i * m
+ j]);
    // 二分答案
    int head = 0, tail = n * m;
    while (head < tail) {
        int mid = (head + tail + 1) >> 1;
        if (check(mid)) head = mid;
        else tail = mid - 1;
    }
    printf("%d\n", head);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

J - X 等于 Y

题解

当序列长度为 1 时，要求 x = y，此时 a = b = 2 即可。

当序列长度为 2 时，设最高位为 t，有 t <= sqrt{x} 且 t <= sqrt{y}（否则如果 t > sqrt{x}，因为 t < a，x >= ta > x 矛盾）。有以下式子：

- 1 <= t < a，1 <= t < b：高位不能超过进制基数。
- 0 <= x - ta < a，0 <= y - tb < b：低位不能超过进制基数。
- x - ta = y - tb：低位也要一样。
- 2 <= a <= A，2 <= b <= B：题目要求。
  利用第三条等式，把所有 b 代换成 b = (y - x)/t + a，就能得到 a 的范围。因为 b 也要是整数，所以 (y - x) mod t = 0 的 t 才能检测。这种情况的复杂度为 O(sqrt{x})。

当序列长度大于等于 3 时，a <= sqrt{x}，b <= sqrt{y}，计算每种 a 和每种 b 的答案，看是否有一样的即可。可以用哈希表记录，但是常数比较大，也可以用双指针的方式判断。这种情况的复杂度为 O(sqrt{x} log x)。

参考代码

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

long long x, y, A, B;

long long ceil(long long a, long long b) {
    return (a + b - 1) / b;
}

// 把 x 的 a 进制表示当成一个 b 进制数
long long gao(long long a, long long b) {
    long long ret = 0;
    for (long long t = x, p = 1; t; t /= a, p *= b) {
        // 注意！t % a >= b 是有可能的，在调用函数的地方检查
        ret += p * (t % a);
        // 超过 y 就退出，防止溢出
        if (ret > y) break;
    }
    return ret;
}

// 检查 x 的 a 进制和 y 的 b 进制表示是否相等
bool check(long long a, long long b) {
    long long xx = x, yy = y;
    while (xx && yy) {
        if (xx % a != yy % b) return false;
        xx /= a; yy /= b;
    }
    return xx == yy;
}

void solve() {
    scanf("%lld%lld%lld%lld", &x, &y, &A, &B);

    // 序列长度为 1 的情况
```

```
    if (x == y) { printf("YES\n2 2\n"); return; }

    // 序列长度为 2 的情况
    for (long long t = 1; t * t <= max(x, y); t++) if ((y - x) % t == 0) {
        long long L = 2, R = A;
        L = max(L, ceil(2 * t + x - y, t));
        L = max(L, x / (t + 1) + 1);
        L = max(L, t + 1);
        L = max(L, ((t + 1) * x - y) / (t * (t + 1)) + 1);
        L = max(L, (t * t + x - y) / t + 1);
        R = min(R, (t * B + x - y) / t);
        R = min(R, x / t);
        if (L <= R) { printf("YES\n%lld %lld\n", L, (t * L - x + y) / t); return; }
    }

    // 序列长度为 3 的情况，用双指针判断
    for (long long a = 2, b = 2; a * a <= x && a <= A; a++) {
        // 把 x 的 a 进制表示当成一个 b 进制数，如果这个数不够 y 说明 b 进制不够大
        while (gao(a, b) < y && b * b <= y && b <= B) b++;
        if (b * b > y || b > B) break;
        // 必须检查两种表示是否真的相等，因为 gao 函数中，x 的 a 进制表示里的某一位可能大于等于 b
        if (gao(a, b) == y && check(a, b)) { printf("YES\n%lld %lld\n", a, b); return; }
    }

    printf("NO\n");
}
```

K - 独立钻石

题解

因为每个棋子只能被横向或者纵向跳过，因此当棋盘上存在 k 枚棋子时，共有 2k 种操作可以选择。又因为每一步都将减少 1 枚棋子，因此至多执行 (k - 1) 步。

直接通过 dfs 进行搜索的复杂度为

$$O(T * nm * \prod_{i=2}^{k} 2i) \approx 1.7 * 10^7,$$

无需任何优化即可在时限内通过。

参考代码

```
#include <bits/stdc++.h>
#define MAXN 6
#define MAXM 6
using namespace std;

int n, m, K, ans;
int MAP[MAXN + 5][MAXM + 5];

void dfs(int now) {
```

```
    ans = min(ans, now);
    for (int i = 1; i <= n; i++) for (int j = 1; j <= m; j++) if (MAP[i][j]) {
        if (i > 1 && i < n) {
            // 向下跳
            if (MAP[i - 1][j] && !MAP[i + 1][j]) {
                MAP[i - 1][j] = MAP[i][j] = 0;
                MAP[i + 1][j] = 1;
                dfs(now - 1);
                MAP[i - 1][j] = MAP[i][j] = 1;
                MAP[i + 1][j] = 0;
            }

            // 向上跳
            if (!MAP[i - 1][j] && MAP[i + 1][j]) {
                MAP[i + 1][j] = MAP[i][j] = 0;
                MAP[i - 1][j] = 1;
                dfs(now - 1);
                MAP[i + 1][j] = MAP[i][j] = 1;
                MAP[i - 1][j] = 0;
            }
        }

        if (j > 1 && j < m) {
            // 向右跳
            if (MAP[i][j - 1] && !MAP[i][j + 1]) {
                MAP[i][j - 1] = MAP[i][j] = 0;
                MAP[i][j + 1] = 1;
                dfs(now - 1);
                MAP[i][j - 1] = MAP[i][j] = 1;
                MAP[i][j + 1] = 0;
            }

            // 向左跳
            if (!MAP[i][j - 1] && MAP[i][j + 1]) {
                MAP[i][j + 1] = MAP[i][j] = 0;
                MAP[i][j - 1] = 1;
                dfs(now - 1);
                MAP[i][j + 1] = MAP[i][j] = 1;
                MAP[i][j - 1] = 0;
            }
        }
    }
}

void solve() {
    scanf("%d%d%d", &n, &m, &K);
    memset(MAP, 0, sizeof(MAP));
    for (int i = 1; i <= K; i++) {
        int x, y; scanf("%d%d", &x, &y);
        MAP[x][y] = 1;
    }
    ans = K; dfs(K);
    printf("%d\n", ans);
}
```

```
int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

L - 经典问题

题解

求完全图的最小生成树通常使用 Boruvka 算法。

称特殊边连接的节点为特殊点，其它节点为一般点。可以发现，2m 个特殊点将一般点分成了 (2m + 1) 段。一般点和其它点的连边权值至少为 1。因此最后肯定存在一种最小生成树，使得每段一般点 l, (l + 1), … , r 内部是从小到大依次相连的。

连接后，我们可以把图缩为 (2m + 1) 个"连续点"（每个连续点代表连续的一段一般点）以及 2m 个特殊点的完全图。接下来我们需要在这张图上运行 Boruvka 算法，问题变为：快速维护每个点向其它连通块连边的最小边权。

每一轮 Boruvka 刚开始时，计算 pre[i] 表示点 i 左边最近的，且和它不在同一连通块的点是哪个。同理，计算 nxt[i] 表示点 i 右边最近的，且和它不在同一连通块的点是哪个。接下来从左到右枚举每个点：

- 如果当前是连续点，选择 pre[i] 和 nxt[i] 中距离最近的即可。
- 如果当前是特殊点，则需要向左 / 右枚举到第一个和它没有连特殊边，且不在同一连通块内的点。另外还要考虑与它相邻的所有特殊边。这一步总体是 O(m) 的。实现详见参考代码的注释。Boruvka 算法执行 O(log点数) 轮，因此总体复杂度为 O(m log m)。

参考代码

```
#include <bits/stdc++.h>
#define MAXM ((int) 1e5)
using namespace std;
typedef pair<int, int> pii;

int n, m;
long long ans;

// 缩点后图上的一个节点
struct Vert {
    // spec == true：特殊点；spec == false：连续点
    bool spec;
    // 对于特殊点，L = R = 点的编号；对于连续点，L 是所代表区间的左端点，R 是右端点
    int L, R;
    // 只对特殊点有用，记录与该特殊点相邻的所有特殊边，key 是终点编号，value 是权值
    unordered_map<int, int> e;
};
vector<Vert> V;

int root[MAXM * 4 + 10], pre[MAXM * 4 + 10], nxt[MAXM * 4 + 10];

int findroot(int x) {
    if (root[x] != x) root[x] = findroot(root[x]);
```

```
    return root[x];
}

void solve() {
    scanf("%d%d", &n, &m);
    map<int, vector<pii>> e;
    for (int i = 1; i <= m; i++) {
        int x, y, z; scanf("%d%d%d", &x, &y, &z);
        e[x].push_back(pii(y, z)); e[y].push_back(pii(x, z));
    }

    // 用特殊点把一般点分成一段段区间
    V.clear();
    int last = 0;
    for (auto &entry : e) {
        int pos = entry.first;
        if (last + 1 != pos) V.push_back(Vert{false, last + 1, pos - 1, {}});
        V.push_back(Vert{true, pos, pos, {}});
        last = pos;
    }
    if (last != n) V.push_back(Vert{false, last + 1, n, {}});
    n = V.size();

    // 把原图的点编号转换成缩点后图的点编号
    unordered_map<int, int> mp;
    for (int i = 0; i < n; i++) if (V[i].spec) mp[V[i].L] = i;
    auto it = e.begin();
    for (auto &v : V) if (v.spec) {
        for (auto &[fn, val] : it->second) v.e[mp[fn]] = val;
        it++;
    }

    // 把每个连续点内部连起来
    ans = 0;
    for (auto &v : V) if (!v.spec) ans += v.R - v.L;

    for (int i = 0; i < n; i++) root[i] = i;
    int comp = n;
    // boruvka
    while (comp > 1) {
        // pre[i] 表示点 i 左边最近的，且和它不在同一连通块的点是哪个
        pre[0] = -1;
        for (int i = 1; i < n; i++) pre[i] = (findroot(i - 1) == findroot(i) ?
pre[i - 1] : i - 1);
        // nxt[i] 表示点 i 右边最近的，且和它不在同一连通块的点是哪个
        nxt[n - 1] = n;
        for (int i = n - 2; i >= 0; i--) nxt[i] = (findroot(i + 1) == findroot(i)
? nxt[i + 1] : i + 1);

        vector<int> best(n, -1), len(n, 2e9);
        for (int i = 0; i < n; i++) {
            int r = findroot(i);
            if (V[i].spec) {
                // 考虑与该特殊点相连的所有特殊边
                // 特殊边一共 m 条，所以每一轮 boruvka 这一段总共执行 m 次
```

```
                for (auto &[fn, val] : V[i].e)
                    if (r != findroot(fn) && val < len[r]) best[r] = fn, len[r] = val;
                // 寻找左边第一个和它没有连特殊边，且不在同一连通块内的点
                //
                // 简单分析一下这段代码的复杂度。找的时候共有 3 种情况：
                //   1. 碰到有特殊边相连的点，直接继续往前考虑一个点。
                //   2. 碰到同一连通块内的点，通过 pre[j] 直接跳到前一个不在同一连通块内的点。
                //   3. 找到了，直接退出。
                //
                // 情况 1 可能会跳到情况 1，2，3 的任意一种。情况 2 只会跳到情况 1 和 3。
                // 因为情况 1 每轮 boruvka 最多出现 m 次，而情况 2 只能从情况 1 跳过去，
                // 因此情况 2 每轮也最多出现 m 次。
                // 所以这一段代码的复杂度也是每轮 O(m) 的。
                for (int j = pre[i]; j >= 0; ) {
                    // 同一连通块内的点，通过 pre[j] 直接跳
                    if (r == findroot(j)) j = pre[j];
                    // 和 j 有特殊边相连，不能考虑这个点
                    else if (V[i].e.count(j)) j--;
                    else {
                        // 找到了
                        int val = V[i].L - V[j].R;
                        if (val < len[r]) best[r] = j, len[r] = val;
                        break;
                    }
                }
                // 寻找右边第一个和它没有连特殊边，且不在同一连通块内的点，与上面类似的逻辑
                for (int j = nxt[i]; j < n; ) {
                    if (r == findroot(j)) j = nxt[j];
                    else if (V[i].e.count(j)) j++;
                    else {
                        int val = V[j].L - V[i].R;
                        if (val < len[r]) best[r] = j, len[r] = val;
                        break;
                    }
                }
            } else {
                // 连续点，考虑 pre[i] 和 nxt[i] 哪个更好即可
                int j = pre[i];
                if (j >= 0) {
                    int val = V[i].L - V[j].R;
                    if (val < len[r]) best[r] = j, len[r] = val;
                }
                j = nxt[i];
                if (j < n) {
                    int val = V[j].L - V[i].R;
                    if (val < len[r]) best[r] = j, len[r] = val;
                }
            }
        }

        // 把这一轮我们选上的边连接一下
        for (int i = 0; i < n; i++) if (best[i] >= 0) {
            int x = findroot(i), y = findroot(best[i]);
```

```
            if (x == y) continue;
            root[x] = y;
            ans += len[i];
            comp--;
        }
    }
    printf("%lld\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```

M - 计算几何

题解

枚举用于切开大多边形的顶点 i 和 j，问题变为如何快速计算两个小多边形的直径。显然两个小多边形也都是凸多边形。

因为凸多边形的直径一定是某两个顶点的连线（初中几何证明题，请读者自行完成），维护 f(i, j) 表示第 i 个顶点到第 j 个顶点之间，两个顶点之间的最大距离的平方（如果 i > j 那就是顶点 i, i + 1, ... , n, 1, 2, ... , j 之间的最大距离），令 \text{dis}(i, j) 表示顶点 i 和 j 之间的距离，容易得到区间 dp 方程

[f(i, j) = \max { f(i + 1, j), f(i, j - 1), \text{dis}^2(i, j) }]
初值 f(i, i + 1) = \text{dis}^2(i, i + 1)。

两个小多边形的直径平方和即为 f(i, j) + f(j, i)，取最小值为答案即可。

当然，要求顶点 i 和 j 的连线切到凸多边形内部。根据凸多边形的性质，这等价于顶点 j 不能在顶点 i 和 (i + 1) 的连线上，也不能在顶点 i 和 (i - 1) 的连线上。算出叉积进行判断即可。

复杂度 O(n^2)。

参考代码

```
#include <bits/stdc++.h>
#define MAXN ((int) 5e3)
using namespace std;

int n;
long long ans, X[MAXN], Y[MAXN];

long long f[MAXN][MAXN];

long long cross(long long x1, long long y1, long long x2, long long y2) {
    return x1 * y2 - x2 * y1;
}

long long dis2(int i, int j) {
    return (X[i] - X[j]) * (X[i] - X[j]) + (Y[i] - Y[j]) * (Y[i] - Y[j]);
}
```

```
void solve() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%lld%lld", &X[i], &Y[i]);

    // 区间 dp
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        f[i][j] = dis2(i, j);
    }
    for (int len = 3; len <= n; len++) for (int i = 0; i < n; i++) {
        int j = (i + len - 1) % n;
        f[i][j] = max({f[(i + 1) % n][j], f[i][(j - 1 + n) % n], dis2(i, j)});
    }

    ans = 9e18;
    for (int i = 0; i < n; i++) {
        int nxt = (i + 1) % n, pre = (i - 1 + n) % n;
        for (int j = 0; j < n; j++) if (i != j) {
            // 点 j 不能在连接 i 和 (i + 1) 的直线上，否则这条线无法切到多边形内部
            long long c1 = cross(X[j] - X[i], Y[j] - Y[i], X[nxt] - X[i], Y[nxt]
- Y[i]);
            // 点 j 不能在连接 i 和 (i - 1) 的直线上，否则这条线无法切到多边形内部
            long long c2 = cross(X[j] - X[i], Y[j] - Y[i], X[pre] - X[i], Y[pre]
- Y[i]);
            if (c1 == 0 || c2 == 0) continue;
            ans = min(ans, f[i][j] + f[j][i]);
        }
    }
    printf("%lld\n", ans);
}

int main() {
    int tcase; scanf("%d", &tcase);
    while (tcase--) solve();
    return 0;
}
```