Tutorial

The 2024 FJNU Programming Contest

FJNUACM Problem Setting Team
2024.05.22

E题为原题，来自CF 1174F，略微改动了数据范围

H题为CF 1609D的数据范围加强版

难度预估

Easy: AL

Easy-Mid: DFJ

Mid: CIK

Mid-Hard: GH

Hard: BE

实际比赛过程中 Mid 档题目全都被参赛队伍跳过了（可能是看着不可做导致的）

A. Crazy Yesterday

模拟。

取模一下即可，或者分讨。

时间复杂度: $O(1)$

Std (implementation)

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int w;
    cin >> w;
    cout << (w + 5) % 7 + 1 << '\n';
}

int main(){
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *题面的暗示是疯狂星期四*

B. Solo Leveling

离散化，前缀和，`dp` / `bfs`，思维。

理论上贪心是全错的，这题最诈骗的地方就在一看就能乱贪（数据应该够强）。

不妨考虑将两个属性转换为坐标系上的两个坐标轴。那么，我们就可以将怪物视为坐标系上的点，玩家也是一样的。

在此基础上，我们可以得出一些结论：

结论1

如果玩家的属性是 A, B，那么他可以杀死满足 $x \le A$，$y \le B$ 的所有怪物 (x, y)。

结论2

如果玩家的属性是 A, B，那么不妨令 D 为 $(10, 10)$ 和 (A, B) 之间的曼哈顿距离，即 $D = |A - 10| + |B - 10|$；再令 $S = \sum c_i$ 为满足 $x \le A, y \le B$ 的所有怪物 (x, y) 的 c_i 总和。此时，可分配点数为 $S - D$。

上面的 S 可以通过预处理二维前缀和得到。

结论3

杀死所有怪物时，玩家的属性为 $(a_{\max}, b_{\max})$。至于为什么不可以更大，因为点数是可以浪费的，所以无所谓。

关于本结论的正确性，反证法可证。

总结

如上，问题就转化为：检查能否从 $(10, 10)$ 转移到 $(a_{\max}, b_{\max})$。

我们可以在每个点上画一条垂直线和一条水平线，以此得到一个表格。这在每个坐标轴上看来更像是一种去重，或者正式地说是离散化。

令数组 p，q 表示去重后的数组 a，b。

我们需要确保可分配点数足够，才能从 (p_{x-1}, q_y) 转移到 (p_x, q_y)。那么不妨令 $dp[x][y]$ 为玩家状态为 (p_x, q_y) 时的可分配点数。此时如果需要进行转移，就必须满足：$dp[x-1][y] \ge p_x - p_{x-1}$。

从 (p_x, q_{y-1}) 转移到 (p_x, q_y) 的条件也是类似的。

那么我们只需以上面的 dp 式子为条件，用 结论2 的式子，从 $(10, 10)$ 转移到 $(a_{\max}, b_{\max})$ 即可。转移完成后，我们只需检查最后的状态的值是否为非负数。

需要注意的是，因为初始状态是 $(10, 10)$，所以我们不能从比这个更小的状态转移过来，也就是满足 $x < 10$ 或 $y < 10$ 的点 (x, y)。

当然，你也可以使用类似于 `bfs` 的搜索算法。

时间复杂度: $O(n^2)$

Std (dp, hashing, prefix sum)

```cpp
#include <bits/stdc++.h>

using namespace std;
using i64 = long long;

const i64 inf = 0x3f3f3f3f3f3f3f3f;

void solve() {
    int n;
    cin >> n;
    vector<array<int, 3>> points(n);
    vector<int> p, q;
    for (auto &[a, b, c] : points) {
        cin >> a >> b >> c;
        p.emplace_back(a), q.emplace_back(b);
    }
    p.emplace_back(10), q.emplace_back(10);

    sort(p.begin(), p.end()), sort(q.begin(), q.end());
    n = unique(p.begin(), p.end()) - p.begin();
    int m = unique(q.begin(), q.end()) - q.begin();

    vector<vector<i64>> pre(n + 1, vector<i64>(m + 1));
    for (auto &[a, b, c] : points) {
        int x = lower_bound(p.begin(), p.begin() + n, a) - p.begin() + 1,
            y = lower_bound(q.begin(), q.begin() + m, b) - q.begin() + 1;
        pre[x][y] += c;
```

```cpp
    }

    vector<vector<i64>> dp(n + 1, vector<i64>(m + 1, -inf));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            pre[i][j] += pre[i - 1][j] + pre[i][j - 1] - pre[i - 1][j - 1];
            if (p[i - 1] <= 10 && q[j - 1] <= 10) dp[i][j] = pre[i][j];
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (p[i - 1] <= 10 && q[j - 1] <= 10) continue;
            if ((dp[i - 1][j] >= p[i - 1] - p[i - 2] && p[i - 2] >= 10 && q[j - 1] >= 10) ||
                (dp[i][j - 1] >= q[j - 1] - q[j - 2] && p[i - 1] >= 10 && q[j - 2] >= 10)) {
                    dp[i][j] = pre[i][j] - p[i - 1] - q[j - 1] + 20;
                }
        }
    }
    if (dp[n][m] >= 0) cout << "Yes\n";
    else cout << "No\n";
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *如果一开始就想着用贪心做，这道题就完蛋了（x*

C. Chain Reaction

数学。

考虑将所有按钮按下，容易证明只有编号为完全平方数的灯会被打开。

证明如下：

对于 x，每个小于 $\sqrt{x}$ 的因子都会存在一个大于 $\sqrt{x}$ 的与其对应的因子。在所有按钮按下后，若 x 不是完全平方数，则该灯会熄灭；若 x 是完全平方数，则等价于只按了 $\sqrt{x}$ 这个按钮，该灯会被打开。

时间复杂度: $O(n)$

Std (math)

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m;
    cin >> n >> m;
    vector<pair<int, int>> a(m + 1);
    while (m --) {
        int u, v;
        cin >> u >> v;
        a.emplace_back(u, v);
    }
    cout << n << ' ';
    for (int i=1;i<=n;i++) cout << i << " \n"[i == n];
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

D. XOR Pairing

数论，暴力。

一个异或的小性质：$a \oplus b = c$，则 $a \oplus c = b$。

那么，对于每个 a_i，计算有多少个 $j\ (j < i)$ 使得 $a_i \oplus a_j = k$。注意，由于上述性质，在 a_i 和 k 确定的情况下，a_j 的值是唯一确定的。

可以使用 map，或排序后二分查找等方式。

时间复杂度: $O(n \log n)$

Std (number theory, brute force, hashing)

```cpp
#include <bits/stdc++.h>

using namespace std;
using i64 = long long;

void solve() {
    int n, k;
    cin >> n >> k;
    map<int, int> cnt;
    i64 ans = 0;
    for(int i=1;i<=n;i++) {
        int x;
        cin >> x;
        if(cnt.count(x ^ k)) {
            ans += cnt[x ^ k];
        }
        cnt[x]++;
    }
    cout << ans << '\n';
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
```

```
    }
    return 0;
}
```

> *不要试图使用* $O(n^2)$ *的算法通过本题（*

E. Cyber Hide-and-Seek

图论，分治，树的重心/树链剖分。

注意到 $\log_2 3.9 \times 10^5 \approx 18.573 < 19$，而 $39 > 19 \times 2$。只需设法在 2 次询问内将问题规模减小一半，即可在 $2 \times \log_2 n$ 次询问内找到 x，而这个式子的值在 $n \leq 3.9 \times 10^5$ 时是小于 39 的。

由此可以得出两种思想类似但实现不同的解法：

解法1

利用树的重心的性质：当一棵树以重心为根时，其所有子树的大小不大于原树的一半（向下取整）。

首先用一次询问 1 得出 x 的深度 D。在原树上跑一遍 `dfs`，计算出每个结点的深度 $depth_i$，以及子树的大小 $size_i$。对于深度大于 x 的点，显然这些点对询问没有价值，在 `dfs` 时可以去掉。

下面考虑进行分治。设当前子树根为 p 的重心为 q。如果 $size_p = 1$，p 就是答案。否则对 q 做一次询问 1，求出 q 到 x 的距离 dis_q。如果 $dis_q = 0$，显然 $x = q$，可以直接回答。

如果 $dis_q + depth_q = D$，说明 x 是 q 的后代。对 q 做一次询问 2，得出 x 在 q 的某棵子树内，将 q 作为新的根 p 递归。由于 q 是重心，这棵子树的大小一定不超过当前树的一半。

如果 $dis_q + depth_q \neq D$，说明 x 不是 q 的后代；而 x 与 q 的最近公共祖先在 s 到 q 的路径上。容易算出其深度为 $depth_q - (dis_q + depth_q - D)/2$，故只需从 q 向上跳到其公共祖先，然后做一次询问 2 并递归。同样地，由于 q 是重心，新子树的大小也不会超过当前树的一半。

因 此 ， 每 做 两 次 询 问 后 ， 原 问 题 规 模 至 少 减 小 一 半 。 从 而 可 以 在 至 多 $1 + 2 \times \lfloor \log_2 3.9 \times 10^5 \rfloor = 37$ 次询问内得出答案。

解法2

利用重链剖分的性质：从树根到任意一个结点的路径经过的重链不超过 $\lfloor log_2 n \rfloor$ 条。

用两遍 `dfs` 对原树进行重链剖分，保存每个结点的深度 $depth_i$。同样地用一次询问 1 得出 x 的深度 D 后，从结点 1 开始递归分治。

设当前子树根结点为 p（这里的 p 一定是其父节点的轻儿子），每次询问 p 所在重链的最后一个结点 q（也就是该链深度最大的一个结点）到 x 的距离 dis_q。如果 $dis_q = 0$，显然 $x = q$，可以直接回答。

如果 $depth_q + dis_q = D$，说明 x 是 q 的后代。对 q 做一次询问 2 并递归。

如果 $depth_q + dis_q \neq D$，说明 x 不是 q 的后代，且 x 与 q 最近公共祖先的深度为 $depth_q - (dis_q + depth_q - D)/2$。所以直接从 q 往上跳 $(dis_q + depth_q - D)/2$ 个结点后做一次询问 2，然后递归。

在最坏情况下，从结点 1 到 x 的每条重链上都在头尾各执行一次询问。而从 1 到 x 至多只经过 $\lfloor log_2 n \rfloor$ 条重链，因此询问次数同样不会超过 $1 + 2 \times \lfloor \log_2 3.9 \times 10^5 \rfloor = 37$，因而可以在题目限制的 39 次询问内得出答案。

时间复杂度: $O(n)$

Std (dfs, centroid decomposition)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 4e5;

void solve() {
    auto query1 = [&](int x) -> int {
        cout << "1 " << x << endl;
        fflush(stdout);
        int ans;
        cin >> ans;
        return ans;
    };
    auto query2 = [&](int x) -> int {
        cout << "2 " << x << endl;
        fflush(stdout);
        int ans;
        cin >> ans;
        return ans;
    };
    auto answer = [&](int x) -> void {
        cout << "! " << x << endl;
        exit(0);
    };
```

```

    vector<int> e[N];
    int n, dis;
    cin >> n;
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        e[u].emplace_back(v);
        e[v].emplace_back(u);
    }
    dis = query1(1);
    if (!dis) answer(1);
    vector<int> dep(n + 1), size(n + 1), son(n + 1), fa(n + 1);
    auto dfs = [&](auto self, int cur, int f, int depth) -> void {
        dep[cur] = depth;
        fa[cur] = f;
        size[cur] = 1;
        if (depth >= dis) return;
        for (auto i : e[cur]) {
            if (i == f) continue;
            self(self, i, cur, depth + 1);
            size[cur] += size[i];
            if (!son[cur] || size[i] > size[son[cur]]) son[cur] = i;
        }
    };
    dfs(dfs, 1, 0, 0);
    int cur = 1;
    while (true) {
        if (dep[cur] == dis) {
            answer(cur);
            return;
        }
        if (size[cur] >= size[son[cur]] * 2) {
            cur = query2(cur);
            continue;
        }
        int p = cur;
        while (size[son[p]] * 2 > size[cur]) p = son[p];
        int q = query1(p);
        if (q == 0) answer(p);
        if (dis == dep[p] + q) {
```

```cpp
                cur = query2(p);
                continue;
            }
            while (dep[p] + q != dis) p = fa[p], --q;
            cur = query2(p);
        }
}

int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

Std (dfs, heavy-light decomposition)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 4e5;

void solve() {
    auto query1 = [&](int x) -> int {
        cout << "1 " << x << endl;
        fflush(stdout);
        int ans;
        cin >> ans;
        return ans;
    };
    auto query2 = [&](int x) -> int {
        cout << "2 " << x << endl;
        fflush(stdout);
        int ans;
        cin >> ans;
        return ans;
    };
    auto answer = [&](int x) -> void {
```

```
        cout << "! " << x << endl;
        exit(0);
    };

    int n, dis;
    cin >> n;
    vector<int> e[N + 1];
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        e[u].emplace_back(v);
        e[v].emplace_back(u);
    }
    dis = query1(1);
    if (!dis) answer(1);
    vector<int> dep(n + 1), size(n + 1), son(n + 1), fa(n + 1);
    auto dfs1 = [&](auto self, int cur, int f, int depth) -> void {
        dep[cur] = depth;
        fa[cur] = f;
        size[cur] = 1;
        if (depth >= dis) return;
        for (auto i : e[cur]) {
            if (i == f) continue;
            self(self, i, cur, depth + 1);
            size[cur] += size[i];
            if (!son[cur] || size[i] > size[son[cur]]) son[cur] = i;
        }
    };
    dfs1(dfs1, 1, 0, 0);
    vector<vector<int>> heavy(n + 1);
    auto dfs2 = [&](auto self, int cur, int t) -> void {
        heavy[t].emplace_back(cur);
        if (dep[cur] == dis) return;
        if (son[cur]) self(self, son[cur], t);
        for (auto i : e[cur]) {
            if (i != son[cur] && i != fa[cur])
                self(self, i, i);
        }
    };
    dfs2(dfs2, 1, 1);
    int p = 1, q, d;
```

```
    while (true) {
        if (dep[p] == dis) answer(p);
        q = heavy[p].back();
        d = query1(q);
        if (!d) answer(q);
        d = (dep[q] + d - dis) / 2;
        p = query2(*(heavy[p].end() - d - 1));
    }
}

int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *事实上这个 37 次询问的上界还可以更低。因为如果要保证每次递归后都需要询问两次，递归次数就卡不到 $\lfloor log_2 n \rfloor$。此外还有一些剪枝可以减少询问次数。*

F. Double Holding

模拟，离散化，双指针，差分。

本题有两个解法。

解法1

因为一个序列上不会出现重叠，所以将两个序列合并后，重叠区域的长度就是答案。

因为值域很大，不妨考虑用 map 进行离散化，然后在 map 上维护一个差分数组。统计答案时，正着跑一下前缀和即可。

解法2

在两个序列上维护两个指针，进行模拟。这是一个比较经典的线段覆盖问题。

当然也有将双轨合并来计算重叠区域的做法，但是不如直接在两个轨道上各维护一个指针好写。

时间复杂度: $O((n+m)\log(n+m))$

Std (hashing, difference)

```
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m, E;
    cin >> n >> m >> E;
    map<int, int> d;
    for (int i = 1; i <= n + m; i++) {
        int l, r;
        cin >> l >> r;
        d[l] ++, d[r + 1] --;
    }
    int sum = 0, ans = E;
    int pre = 0;
    for (auto &[x, y] : d) {
        if (sum == 2) ans -= x - pre - 1;
```

```cpp
        sum += y;
        pre = x;
    }
    cout << max(-1, ans) << '\n';
}

int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

Std (sorting, two pointers, implementation)

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m, E;
    cin >> n >> m >> E;
    vector<pair<int, int>> s(n + 1), t(m + 1);
    for (int i = 1; i <= n; i++) {
        auto &[l, r] = s[i];
        cin >> l >> r;
    }
    for (int i = 1; i <= m; i++) {
        auto &[l, r] = t[i];
        cin >> l >> r;
    }
    sort(s.begin(), s.end()), sort(t.begin(), t.end());
    int x = 1, ans = E;
    for (int y = 1; y <= m; y++) {
        if (t[y].second <= s[x].first) continue;
        while (x <= n && s[x].second <= t[y].second) {
            ans -= max(0, min(s[x].second, t[y].second) - max(s[x].first, t[y].first));
            x ++;
```

```
        }
        if (x > n) break;
        ans -= max(0, min(s[x].second, t[y].second) - max(s[x].first, t[y].first));
    }
    cout << max(-1, ans) << '\n';
}

int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *劲爆 2K 音游等你来战（划掉*

G. Color Contagion

组合数学，图论，树型dp，搜索。

令 $f[x]$ 表示以 x 为根的子树的染色方案数，$siz[x]$ 表示以 x 为根的子树大小。

考虑根为 rt 的两个子树如何合并。实际上，染色方案可以抽象成一个序列，合并两个序列，保持它们的相对顺序不变，可以将一个序列按顺序插入到另一个序列的间隔中，使用隔板法容易算出方案数。

$$f[rt] = f[son_1] \cdot f[son_2] \cdot \binom{sz[son_1] + siz[son_2]}{sz[son_1]}$$

多个子树的情况按顺序合并即可。

Std (combinatorics, dp, graph, dfs)

```cpp
#include <bits/stdc++.h>

using namespace std;
using i64 = long long;

int power(int a, i64 b, int p) {
    //快速幂
    int res = 1;
    for (; b; b /= 2, a = 1LL * a * a % p) {
        if (b % 2) {
            res = 1LL * res * a % p;
        }
    }
    return res;
}

const int N = 1e5 + 10, mod = 998244353;

int fac[N + 1], inv[N + 1];

void init() {
    fac[0] = 1;
    for (int i = 1; i <= N; i++) {
        fac[i] = 1LL * fac[i - 1] * i % mod;
```

```
    }
    inv[N] = power(fac[N], mod - 2, mod);
    for (int i = N - 1; i >= 0; i--) { //预处理逆元
        inv[i] = 1LL * inv[i + 1] * (i + 1) % mod;
    }
}

int C(int n, int m) { //组合数
    return 1LL * fac[n] * inv[m] % mod * inv[n - m] % mod;
}

void solve() {
    int n;
    cin >> n;
    vector<vector<int>> adj(n + 1);
    vector<i64> sz(n + 1), dp(n + 1);
    for (int i = 1; i < n; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].emplace_back(v);
        adj[v].emplace_back(u);
    }
    auto dfs = [&](auto self, int u, int p) -> void {
        sz[u] = 1;
        dp[u] = 1;
        i64 t = 0;
        for (auto v : adj[u]) {
            if (v == p) continue;
            self(self, v, u);
            dp[u] = dp[u] * C(t + 1 + sz[v] - 1, t) % mod * dp[v] % mod;
            t += sz[v];
        }
        sz[u] = t + 1;
    };
    dfs(dfs, 1, 0);
    cout << (dp[1] % mod + mod) % mod << '\n';
}

int main() {
    init();
    int T;
```

```
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

H. Seeking Allies

思维，并查集，图论，数据结构，模拟。

本题可以拆分为两个子问题。

问题1

首先，我们将这 n 个人视为图上的点，那么一开始图中没有边。

如果两个人之间认识，那么他们就处于同一个连通块中。

如果一个连通块中有 x 个点，那么边数的最小值一定是 $x-1$，即形成一颗树。

于是我们可以用给定的条件依次连边，如果被连接的两个点位于同一连通块，那么操作是无意义的。这可以用并查集来判断。

不妨统计无意义的操作数量 lf，那么我们就可以用这些多出来的操作，使最大的连通块尽可能大。

显然，将最大的前 $lf+1$ 个连通块连接起来是最优的。

问题2

如何维护前 $lf+1$ 个连通块的大小之和 sum?

显然不可以 $O(n)$ 暴力统计，我们不妨考虑使用数据结构，用 $O(\log n)$ 的复杂度解决。

观察到 lf 是单调递增的，且每次只会 $+1$。

那么，我们只需维护这个边界的左右两端即可。这里有两个方案。

方案 1 是使用对顶堆，即用两个堆分别维护左侧和右侧的有序集合。也可以用两个 `set` 实现。

方案 2 是使用平衡树，即一个可以获取 第 k 大的元素是什么 以及 比指定元素大的元素数量 的有序集合。平衡树可以直接用 `pb_ds` 库。

连通块合并时，维护 sum 需要进行一些分讨，不妨画图模拟这个过程。

时间复杂度: $O(n+d\log d)$

Std (dsu, pbds, treap, data structure, implementation)

```cpp
#include <bits/extc++.h>

using namespace std;

template<typename Type, typename Comp = less<Type>>
using treap = __gnu_pbds::tree<Type, __gnu_pbds::null_type, Comp, __gnu_pbds::rb_tree_tag, __gnu_pbds::tree_order_statistics_node_update>;

void solve() {
    int n, d;
    cin >> n >> d;
    vector<int> pa(n + 1), cnt(n + 1, 1);
    iota(pa.begin(), pa.end(), 0ll);
    auto find = [&](auto self, int x) -> int{
        return pa[x] == x ? x : pa[x] = self(self, pa[x]);
    };
    int lf = 1;
    treap<pair<int, int>> st;
    for (int i = 1; i <= n; i++) st.insert({-1, i});
    int sum = 1;
    while (d--) {
        int x, y;
        cin >> x >> y;
        int f1 = find(find, x), f2 = find(find, y);
        if (f1 != f2) {
            pa[f2] = f1;
            int ind1 = st.order_of_key({-cnt[f1], f1}), ind2 = st.order_of_key({-cnt[f2], f2});
            if (ind1 < lf) sum -= cnt[f1];
            if (ind2 < lf) sum -= cnt[f2];
            st.erase({-cnt[f1], f1});
            st.erase({-cnt[f2], f2});
            cnt[f1] += cnt[f2];
            st.insert({-cnt[f1], f1});
            int n_ind = st.order_of_key({-cnt[f1], f1});
            if (ind1 < lf && ind2 < lf) { //区间内有两个，补偿代价
                sum += cnt[f1]; //变大后肯定在区间内
                if (st.size() >= lf) sum += -(*st.find_by_order(lf - 1)).first;
```

```cpp
            } else if (ind1 >= lf && ind2 >= lf) { //区间内一个都没有，但可能会进入区间
                if (n_ind < lf) { //进入区间，顶出来一个
                    sum += cnt[f1];
                    sum -= -(*st.find_by_order(lf)).first;
                }
            } else {
                if (n_ind < lf) { //进入区间，1换1
                    sum += cnt[f1];
                }
            }
        } else {
            lf ++;
            if (st.size() >= lf) sum += -(*st.find_by_order(lf - 1)).first;
        }
        cout << sum - 1 << '\n';
    }
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

Std (dsu, set, data structure, implementation)

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, d;
    cin >> n >> d;
    vector<int> fa(n + 1), sz(n + 1);
    auto find = [&](auto self, int x) -> int{
        return fa[x] = x ? x : fa[x] = self(self, fa[x]);
```

```cpp
    };
    for (int i = 1; i <= n; i++) fa[i] = i, sz[i] = 1;
    set<pair<int, int>> minn, maxx;
    for (int i = 1; i < n; i++) minn.insert({1, i});
    maxx.insert({1, n});
    int ans = 1, cnt = 1;
    for (int i = 1; i <= d; i++) {
        int u, v;
        cin >> u >> v;
        u = find(find, u), v = find(find, v);
        if (u == v) {
            cnt ++;
            ans += minn.rbegin() -> first;
            maxx.insert(*minn.rbegin());
            minn.erase(*minn.rbegin());
        } else {
            if (sz[u] < sz[v] || (sz[u] = sz[v] && u < v)) swap(u, v);
            int num = minn.count({sz[u], u}) + minn.count({sz[v], v});
            switch (num) {
                case 0: {
                    fa[v] = u;
                    maxx.erase({sz[u], u});
                    maxx.erase({sz[v], v});
                    sz[u] += sz[v];
                    ans += minn.rbegin() -> first;
                    maxx.insert({sz[u], u});
                    maxx.insert(*minn.rbegin());
                    minn.erase(*minn.rbegin());
                    break;
                }
                case 1: {
                    if (minn.count({sz[u], u}) > minn.count({sz[v], v}))
                        swap(u, v);
                    fa[v] = u;
                    maxx.erase({sz[u], u});
                    minn.erase({sz[v], v});
                    sz[u] += sz[v];
                    ans += sz[v];
                    maxx.insert({sz[u], u});
                    break;
                }
```

```cpp
                case 2: {
                    fa[v] = u;
                    minn.erase({sz[u], u});
                    minn.erase({sz[v], v});
                    sz[u] += sz[v];
                    if (sz[u] < maxx.begin() -> first)
                        minn.insert({sz[u], u});
                    else {
                        ans += sz[u] - maxx.begin() -> first;
                        minn.insert(*maxx.begin());
                        maxx.erase(*maxx.begin());
                        maxx.insert({sz[u], u});
                    }
                    break;
                }
            }
        }
        cout << ans - 1 << endl;
    }
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *实现的方法很多，关键还是在于能否分讨清楚*

I. Aeroplane Chess

dp，数学，前缀和。

考虑期望 dp，用 dp_i 表示离终点 i 格的地方期望上需要几步可以到达终点。先考虑计算 dp_1 到 dp_n 的值。显然，如果棋子已经在 $[1,n]$ 范围内，无论如何其也不会离开这个范围，因此以下讨论建立在棋子始终在 $[1,n]$ 范围的基础上。

容易列出 dp 方程（考虑在 i 位置分别掷出了 $1,2,\ldots,n$）：

$$dp_1 = 1 + \frac{0 + dp_1 + dp_2 + \ldots\ldots + dp_{n-1}}{n}$$

$$dp_2 = 1 + \frac{dp_1 + 0 + dp_1 + \ldots\ldots + dp_{n-2}}{n}$$

$\vdots$

$$dp_n = 1 + \frac{dp_{n-1} + dp_{n-2} + \ldots\ldots + 0}{n}$$

观察可得，dp_i 都是 $1 + \dfrac{0 + (n-1)\text{个未知数}}{n}$ 的形式。

由于这个形式相当对称，dp_1 到 dp_n 都相等时方程组会有解，可以解出 $i \in [1,n]$ 时，$dp_i = n$。或者，可以通过打表（高斯消元或者 `dfs`）得出这一结论。

此处给出较为数学的证明方法：

对于位置 i，其可能情况是：投一次到达终点，投两次到达终点，$\ldots$，投 ∞ 次到达终点。

那么：$dp_i = \sum_{i=1}^{\infty} \frac{i}{n} \cdot (\frac{n-1}{n})^{i-1}$。其中，前一项的 i 表示投了 i 次到达， 分母表示这次投出来了，后一项表示前面几次都没到达。通过一些简单的高等数学知识，我们也可以计算出 $dp_i = n$。

对于 $i > n$ 的部分，$dp_i = 1 + \frac{\sum_{j=i-n}^{i-1} dp_j}{n}$。考虑使用前缀和优化 dp 即可。

时间复杂度: $O(m)$

Std (dp, math, prefix sum)

```cpp
#include <bits/stdc++.h>

using namespace std;
```

```cpp
using i64 = long long;

const int mod = 998244353;

int power(int a, i64 b, int p) {
    int res = 1;
    for (; b; b /= 2, a = 1LL * a * a % p) {
        if (b % 2) {
            res = 1LL * res * a % p;
        }
    }
    return res;
}

i64 inv(int n) {
    return power(n, mod - 2, mod);
}

void solve() {
    int n, q;
    cin >> n >> q;
    vector<i64> dp(1e6 + 1);
    i64 s = 0;
    for (int i = 1; i <= n; i++) {
        dp[i] = n;
        s += dp[i];
    }
    i64 v = inv(n);
    for (int i = n + 1; i <= 1e6; i++) {
        dp[i] = (s * v + 1) % mod;
        s += dp[i];
        s -= dp[i - n];
        s = (s % mod + mod) % mod;
    }
    while (q--) {
        int x;
        cin >> x;
        cout << dp[x] << '\n';
    }
}

```

```
int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

J. Shifting Tournament

思维，模拟，数学。

本题可以用两种不同的观察得到同样的结论。

观察1

任意两轮的选择之间互不干扰，或者说，如果某一轮的选择不一样，那么无论剩下的轮怎么选择，最后胜者的结果都会不一样。

观察2

我们记获胜者的编号为 x。

可以发现，$x-1$ 的长为 k 二进制表示进行左右翻转后，和字符串一一对应。

结论

答案等于规则的方案数。

那么我们只需维护 $?$ 的数量 cnt，然后输出 2^{cnt} 即可。

2^x 可以预处理，也可以用快速幂，注意取模。

时间复杂度: $O(n+q)$

Std (implementation, math)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int mod = 998244353;

void solve(){
    int k;
    cin >> k;
    string s;
    cin >> s;
    s = " " + s;
```

```cpp
    int cnt = count(s.begin() + 1, s.end(), '?');
    vector<int> p2(k + 1);
    p2[0] = 1;
    for (int i = 1; i <= k; i++) p2[i] = (p2[i - 1] * 2) % mod;
    int q;
    cin >> q;
    while (q--) {
        int p;
        string c;
        cin >> p >> c;
        if(s[p] == '?') cnt--;
        s[p] = c[0];
        if(s[p] == '?') cnt++;
        cout << p2[cnt] << '\n';
    }
}

int main() {
    int T;
    T = 1;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *本题的一大难点是有耐心读完题目（追真*

K. Uniform Dispersion

贪心，二分，模拟。

考虑贪心寻找划分的线。

首先点数如果不能均等划分，即 $n\%(k+1)^2 \neq 0$ 时，一定无解。

分别考虑对横纵坐标做以下操作：设 $len = \frac{n}{k+1}$，假设答案存在的话，每条线一定会把 n 个点均等分为 $k+1$ 份，每份有 len 个点。因此，我们对横（纵）坐标排序后，检查 len 与 $len+1$ 至 $k \cdot len$ 与 $k \cdot len+1$ 间能否加入划分的线，不能则一定无解。

完成以上操作，我们得到了 k 条横线和 k 条竖线，所以便有了 $(k+1)^2$ 个区域。对于每个点，我们二分查找其会落在哪个区域，计算每个区域中会有几个点。最后再检查每个区域内的点数是否相同，若相同，则有解（事实上，我们也找到了一个合法解），反之亦然。

时间复杂度: $O(n \log k)$

Std (greedy, binary search, implementation)

```cpp
#include <bits/stdc++.h>

using namespace std;

struct point {
    int x = 0, y = 0;
};

void solve() {
    int n, k;
    cin >> n >> k;
    vector<point> a(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> a[i].x >> a[i].y;
        a[i].x *= 2;
        a[i].y *= 2;
    }
    if (n % ((k + 1) * (k + 1))) {
        cout << "No\n";
        return;
    }
```

```cpp
    vector<int> x(n + 1), y(n + 1);
    for (int i = 1; i <= n; i++) {
        x[i] = a[i].x, y[i] = a[i].y;
    }
    sort(x.begin() + 1, x.end());
    sort(y.begin() + 1, y.end());
    vector<int> kx, ky;
    int len = n / (k + 1);
    for (int i = 1; i <= k; i++) {
        if (x[i * len] == x[i * len + 1] || y[i * len] == y[i * len +
1]) {
            cout << "No\n";
            return;
        }
        kx.emplace_back(x[i * len] + 1);
        ky.emplace_back(y[i * len] + 1);
    }
    kx.emplace_back(INT32_MAX);
    ky.emplace_back(INT32_MAX);
    vector<vector<int>> cnt(k + 1, vector<int>(k + 1));
    for (int i = 1; i <= n; i++) {
        int dx = lower_bound(kx.begin(), kx.end(), a[i].x) - kx.begin();
        int dy = lower_bound(ky.begin(), ky.end(), a[i].y) - ky.begin();
        cnt[dx][dy]++;
    }
    int p = n / (k + 1) / (k + 1);
    for (int i = 0; i <= k; i++) {
        for (int j = 0; j <= k; j++) {
            if (cnt[i][j] != p) {
                cout << "No\n";
                return;
            }
        }
    }
    cout << "Yes\n";
}

int main() {
    int T;
    cin >> T;
    while (T--) {
```

```
        solve();
    }
    return 0;
}
```

> *由于合法情况下* $(k+1)^2 \le n$，$\log k$ *并不大。*

L. Terabyte Connection

暴力，模拟。

记 $f_i = p_i + t_i$，则答案为 $p_{\max}$ 和 $f_{\max}$。

时间复杂度: $O(n)$

Std (implementation, brute force)

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n;
    cin >> n;
    vector<int> p(n), f(n);
    for(int i = 0; i < n; i++) {
        cin >> p[i] >> f[i];
        f[i] += p[i];
    }
    cout << *max_element(p.begin(), p.end()) << ' ' << *max_element(f.begin(), f.end()) << '\n';
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        solve();
    }
    return 0;
}
```

> *日常之头尾俩签到*