Tutorial

The 2025 FJNU Programming Contest

FJNUACM Problem Setters Team
2025.05.18

这套题难度和去年差不多，但因为各种原因，K2 没放入正式赛。

难度预估

Easy: A K1 L

Easy-Mid: D F M

Mid: G J

Mid-Hard: B C E H

Hard: I K2

A. One Must Imagine Time Tight,

签到，模拟。

如题，读取输入后输出最小值即可。

时间复杂度: $O(1)$

Std (implementation)

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

using namespace std;

void solve() {
    int a, b, c;
    cin >> a >> b >> c;
    cout << min({a, b, c}) << '\n';
}

int main() {
    solve();
    return 0;
}
```

> *日常第一题签到*

B. Uchiage Hanabi

dp，单调队列。

令 $dp_{i,j}$ 为只考虑前 i 次烟花，并且第 i 次烟花结束时停在位置 j 时的答案。记停在位置 j 时第 i 次烟花的贡献为 $val_{i,j}$，不难得出转移方程：

$dp_{i,j} = val_{i,j} + max\{dp_{i-1,k} \mid |j-k| \leq d(t_i - t_{i-1})\}$。

这样单次转移是 $O(n)$ 的，总状态数有 $O(nm)$ 种，无法通过。但我们发现右边这一项实际上就是求 dp_{i-1} 的一个区间上的最大值；而且当 j 增大的时候，区间的左端点和右端点刚好都右移一格。这就是一个经典的滑动窗口问题。所以可以用单调队列维护区间最大值进行加速，转移复杂度变为均摊 $O(1)$，可以通过。

实际操作时可以通过滚动数组把第一维压掉，这样空间复杂度就只有 $O(n)$。不滚的话因为空间只有 $512M$，只能开大小 5×10^7 左右的数组，大概率会MLE。

时间复杂度: $O(nm)$

Std (dp, monotonous queue)

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

using namespace std;

const long long inf = 0x3f3f3f3f3f3f3f3f;

void solve() {
    long long n, m, d;
    cin >> n >> m >> d;
    vector<long long> dp(n + 1, -inf), g(n + 1, -inf), t(m + 1, 0);
    g[0] = g[n] = 0;
    t[0] = 1;
    vector<int> q(n + 2);
    for (int i = 1; i <= m; i++) {
        int a, b;
        cin >> a >> b >> t[i];
        long long dis = (t[i] - t[i - 1]) * d;
        int hh = 0, tt = -1;
        for (int j = 1; j <= min(dis, n); j++) {
            while (hh <= tt && g[q[tt]] < g[j]) tt--;
```

```
            q[++tt] = j;
        }
        for (int j = 1; j <= n; j++) {
            if (j + dis <= n) {
                while (hh <= tt && g[q[tt]] < g[j + dis]) tt--;
                q[++tt] = j + dis;
            }
            if (hh <= tt && q[hh] < j - dis) hh++;
            if (hh <= tt && g[q[hh]] != -inf)
                dp[j] = g[q[hh]] + 1LL * b * abs(a - j);
        }
        for (int j = 1; j <= n; j++) g[j] = dp[j];
    }
    cout << *max_element(dp.begin() + 1, dp.end()) << '\n';
}

int main() {
    solve();
    return 0;
}
```

> *今年这个位置又放了一道 dp，又是 Hard*

C. Sequential Sequence

数据结构，哈希，前缀和。

首先，我们考虑这样来判断是否合法：

1. 记区间最小值为 A，区间最大值为 B，那么 $B-A+1$ 等于区间长度；
2. 区间内不包含重复数字。

对于条件 1，用线段树维护即可。

对于条件 2，做法很多，比如说离线后用莫队查询。此处给出一种在线的做法。

由于值域很小，不妨考虑将值域内的所有数进行随机哈希映射。可以认为任意一段数值区间 $A\sim B$ 内所有数的哈希值的异或和都是唯一的，哈希碰撞概率比选手明早被陨石砸中的概率低。

我们可以用线段树查询出区间的最大值 A 和最小值 B，如果 区间内所有数 的哈希值的异或和 与 数值区间 $A\sim B$ 内所有数 的哈希值的异或和 相等，那么说明该区间为 $A\sim B$ 内所有数字的任意排列，满足条件 2。

区间异或和可以用线段树顺便维护，或者使用树状数组。数值区间的异或和查询可用前缀和解决。

时间复杂度: $O(n\log n)$

Std (data structure, hashing, prefix sums)

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

using namespace std;

constexpr int N = 1e5 + 10;
constexpr long long inf = 0x3f3f3f3f3f3f3f3f;

//单点修改线段树
template<class Info>
struct SegmentTree {
    int n;
    vector<Info> info;
    SegmentTree() : n(0) {}
    explicit SegmentTree(int _n, Info _v = Info()) {
```

```
        init(_n, _v);
    }
    template<class T>
    explicit SegmentTree(vector<T> _init) {
        init(_init);
    }
    void init(int _n, Info _v = Info()) {
        init(vector(_n, _v));
    }
    template<class T>
    void init(vector<T> _init) {
        n = _init.size();
        info.assign(4 << __lg(n), Info());
        auto build = [&](auto self, int p, int l, int r) -> void{
            if (r - l == 1){
                info[p] = _init[l];
                return;
            }
            int mid = l + r >> 1;
            self(self, 2 * p, l, mid);
            self(self, 2 * p + 1, mid, r);
            pull(p);
        };
        build(build, 1, 0, n);
    }
    void pull(int p){
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void modify(int p, int l, int r, int x, const Info &v){
        if (r - l == 1){
            info[p] = v;
            return;
        }
        int mid = l + r >> 1;
        if (x < mid) modify(2 * p, l, mid, x, v);
        else modify(2 * p + 1, mid, r, x, v);
        pull(p);
    }
    void modify(int p, const Info &v){
        modify(1, 0, n, p, v);
    }
```

```
    Info rangeQuery(int p, int l, int r, int x, int y){
        if (l ⩾ y || r ⩽ x){
            return Info();
        }
        if (l ⩾ x && r ⩽ y){
            return info[p];
        }
        int mid = l + r >> 1;
        return rangeQuery(2 * p, l, mid, x, y) + rangeQuery(2 * p + 1,
mid, r, x, y);
    }
    Info rangeQuery(int l, int r){
        return rangeQuery(1, 0, n, l, r);
    }
};

struct Info {
    long long mn = inf, mx = -inf;
    unsigned long long sum = 0;
};

Info operator+(const Info &a, const Info &b) {
    Info c;
    c.sum = a.sum ^ b.sum;
    c.mn = min(a.mn, b.mn);
    c.mx = max(a.mx, b.mx);
    return c;
}

mt19937_64
rng(std::chrono::steady_clock::now().time_since_epoch().count());

unsigned long long msk[N], pre[N];

void init() {
    for(int i = 1; i ⩽ 1e5; i++) msk[i] = rng();
    for(int i = 1; i ⩽ 1e5; i++) pre[i] = pre[i - 1] ^ msk[i];
}

void solve() {
    int n, q;
```

```
    cin >> n >> q;
    vector<int> a(n + 1);
    SegmentTree<Info> seg(n + 1);
    for (int i = 1; i ⩽ n; i++) {
        cin >> a[i];
        seg.modify(i, {a[i], a[i], msk[a[i]]});
    }
    while (q--) {
        int op;
        cin >> op;
        if (op == 1) {
            int x, y;
            cin >> x >> y;
            seg.modify(x, {y, y, msk[y]});
        } else {
            int l, r;
            cin >> l >> r;
            auto [mn, mx, p] = seg.rangeQuery(l, r + 1);
            if (mx - mn + 1 == r - l + 1 && (pre[mx] ^ pre[mn - 1]) ==
p) {
                cout << "YES\n";
            } else {
                cout << "NO\n";
            }
        }
    }
}

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

> *妙妙哈希来一点*

D. Where's My Money?

签到，模拟。

因为不要求剔除无意义转账，所以本题有很多做法，下面给出两个可参考思路。

首先我们记三个人的总花费分别为 x, y, z。

解法1

我们拿 A 举例，实际上他只需付 $\frac{x}{3}$ 元，剩下的 $\frac{2x}{3}$ 由另外两个人 B, C 平摊，也就是说 $B \to A = C \to A = \frac{x}{3}$。

显然，这对剩下两个人都是成立的，那么就结束了。

解法2

考虑剔除无意义转账后的结果。

不难发现，此时最多出现 2 笔转账。我们记 $AA = \frac{x+y+z}{3}$，那么只可能出现下面几种情况：

1. 只有一个人付的钱超过 AA，那么剩下两个人将钱打给这个人即可；
2. 只有一个人付的钱小于 AA，那么这个人将钱打给另外两个人即可；
3. 三人刚好 AA，不需要转账。

分类讨论即可。

时间复杂度: $O(n)$

Std (1, implementation)

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

using namespace std;

void solve() {
    int n, m, p;
    cin >> n >> m >> p;
    double oc = 0, kp = 0, xw = 0;
    while (n --) {
```

```
        double x;
        cin >> x;
        oc += x;
    }
    while (m --) {
        double x;
        cin >> x;
        kp += x;
    }
    while (p --) {
        double x;
        cin >> x;
        xw += x;
    }
    cout << fixed << setprecision(3);
    cout << kp / 3 << ' ' << xw / 3 << ' ' << oc / 3 << ' ' << xw / 3 <<
' ' << oc / 3 << ' ' << kp / 3 << '\n';
}

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

Std (2, implementation)

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

using namespace std;

void solve() {
    int n, m, p;
    cin >> n >> m >> p;
    double oc = 0, kp = 0, xw = 0;
    while (n--) {
        double x;
        cin >> x;
```

```
        oc += x;
    }
    while (m--) {
        double x;
        cin >> x;
        kp += x;
    }
    while (p--) {
        double x;
        cin >> x;
        xw += x;
    }
    double aa = (oc + kp + xw) / 3;
    cout << fixed << setprecision(2);
    if (oc ⩾ aa && kp ⩾ aa) {
        cout << 0 << ' ' << 0 << ' ' << 0 << ' ' << 0 << ' ' << oc - aa
<< ' ' << kp - aa << '\n';
    } else if (oc ⩾ aa && xw ⩾ aa) {
        cout << 0 << ' ' << 0 << ' ' << oc - aa << ' ' << xw - aa << ' '
<< 0 << ' ' << 0 << '\n';
    } else if (kp ⩾ aa && xw ⩾ aa) {
        cout << kp - aa << ' ' << xw - aa << ' ' << 0 << ' ' << 0 << ' '
<< 0 << ' ' << 0 << '\n';
    } else if (oc ⩾ aa) {
        cout << 0 << ' ' << 0 << ' ' << aa - kp << ' ' << 0 << ' ' << aa
- xw << ' ' << 0 << '\n';
    } else if (kp ⩾ aa) {
        cout << aa - oc << ' ' << 0 << ' ' << 0 << ' ' << 0 << ' ' << 0
<< ' ' << aa - xw << '\n';
    } else {
        cout << 0 << ' ' << aa - oc << ' ' << 0 << ' ' << aa - kp << ' '
<< 0 << ' ' << 0 << '\n';
    }
}

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

```
47      return 0;
48  }
```

> *样例 1 实际上是第一种做法的优化版，考虑了两者之间的冗余*

E. Binary Banter: Counting Combinatorial Bits

数论，Lucas 定理，数位 dp。

数据范围很大，暴力肯定会T飞，首先考虑怎么快速求第二重求和里面的值，即 $\sum_{j=0}^{i}\left(C_i^j \bmod 2\right)$。我们有结论：令 $f(i)$ 为 i 的二进制表示中 1 的个数，则有 $\sum_{j=0}^{i}\left(C_i^j \bmod 2\right)=2^{f(i)}$，题解的后面附上了证明（也可以通过打表得出，前提是注意力足够好）。那么原式化为 $\sum_{i=0}^{n}2^{f(i)}$。

下面考虑化简后怎么做第一重求和。我们记设最高位为第 k 位（对应的值为 2^k），最低位为第 0 位。

先考虑怎么计算 $\sum_{i=0}^{2^k-1}2^{f(i)}$，即 i 的第 0 到 $k-1$ 位任取 $0,1$，剩下高位全 0。考虑枚举 $f(i)$，那么在 $[0,2^k-1]$ 中，共有 $C_k^{f(i)}$ 个 i 满足条件（即从 k 位中任选 $f(i)$ 位的方案数）。对所有可能的 $f(i)$ 求和，得 $\sum_{i=0}^{2^k-1}2^{f(i)}=\sum_{i=0}^{k}2^{f(i)}\times C_k^{f(i)}$。熟悉二项式定理的话会发现这个式子又能转化成 $\sum_{i=0}^{k}2^{f(i)}\times C_k^{f(i)}=3^k$（没看出这个化简的话也能跑过去，但是要多预处理一下这个式子的前缀和）。

但 n 不一定能表示成 2^k-1 的形式。所以接下来考虑如果第 k 位以上有 1 的情况。固定高位有 $f(p)$ 个 1，低 k 位任取 0 或 1。我们发现，这就相当于低 k 位每种取值的贡献乘了一个 2^{cnt}。也就是说 $p\in Z^+$ 时，有 $\sum_{i=p2^k}^{(p+1)2^k-1}2^{f(i)}=2^{f(p)}\times 3^k$。由于这个乘式里的两项是独立的，故可以考虑数位dp：从高位往低位枚举，每次计算 $[p2^k,(p+1)2^k)$ 区间内所有的贡献之和。由于 $[0,n]$ 总可以拆成至多 $\lfloor\log n\rfloor+1$ 个不相交区间，每个区间的贡献计算是 $O(1)$ 的（预处理一下 3 的幂即可），故单次询问复杂度为 $O(\log n)$，总复杂度 $O(t\log n)$。

证明

考虑 C_n^m 在模 2 意义下的值。我们使用 Lucas 定理，该定理表明：

若 $n=(n_kn_{k-1}\dots n_0)_2$，$m=(m_km_{k-1}\dots m_0)_2$ 为 n 和 m 的二进制表示，则有：

$$C_n^m\equiv\prod_{i=0}^{k}C_{n_i}^{m_i}\pmod 2$$

由于模 2 意义下 $C_1^1 = C_1^0 = C_0^0 = 1$，而 $C_0^1 = 0$。因此：
$C_{n_i}^{m_i} \equiv 1 \pmod 2 \iff m_i \leq n_i$

我们知道 0 乘上任何数都得 0，从而：
$C_n^m \equiv 1 \pmod 2 \iff \forall i,\ m_i \leq n_i$

这等价于 m 的所有 1 位只出现在 n 的 1 位上。这意味着若要使 C_n^m 取到奇数，对 n 的二进制表示为 0 的位，m 的二进制表示对应位上必须为 0；而对 n 的二进制表示为 1 的位，m 的二进制表示对应位上可以任取 0 或 1。从而若 n 的二进制表示中有 $f(n)$ 个 1，根据乘法原理，m 可有 $2^{f(n)}$ 种取值可以使 C_n^m 为奇数。得证。

时间复杂度: $O(t \log n)$

Std (number theory, lucas theorem, dp)

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

using namespace std;

constexpr int mod = 998244353;

void solve() {
    long long n;
    cin >> n;
    long long ans = 0, cnt = 1;
    vector<long long> p3(61, 1ll);
    for (int i = 1; i <= 60; i++) p3[i] = p3[i - 1] * 3 % mod;
    for (int i = 60; i >= 0; i--) {
        if ((1ll << i) & n) {
            ans = (ans + cnt * p3[i] % mod);
            cnt = cnt * 2 % mod;
        }
    }
    ans = (ans + cnt % mod) % mod;
    cout << ans << '\n';
}

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

```
28      }
29      return 0;
30  }
```

> *CCB领域大神*

F. Imbalanced Bracket

构造，思维，前缀和，模拟。

显然，如果序列中存在一对不一定相邻的 ()，那么条件一定不满足。

思路1

考虑构造，只需避免出现 ()，而满足条件的形式为若干个连续的) 与若干个连续的 (拼接。

合法构造总共有 $n+1$ 种，取决于)(出现的位置，或者只有一种字符。我们可以从左到右枚举分割点的位置，简单统计计算即可在线性复杂度内确定这种构造下需要修改几次。当然，前缀和也是可以的。

思路2

考虑让所有括号对失配，只需模拟括号配对，统计可以配对的括号数量即可。

时间复杂度: $O(n)$

Std (1, construction)

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

using namespace std;

constexpr int inf = 0x3f3f3f3f;

void solve() {
    int n;
    cin >> n;
    string s;
    cin >> s;
    int cnt_l = count(s.begin(), s.end(), '(');
    int now_r = 0, ans = inf;
    for(int i=0;i<=n;i++) {  // 从 i 开始全是左括号
        int now_l = cnt_l - (i - now_r);
        ans = min(ans, i - now_r + n - i - now_l);
        if(i < n && s[i] == ')') now_r ++;
    }
    cout << ans << '\n';
```

```
}

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

Std (2, implementation)

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

using namespace std;

void solve() {
    int n;
    cin >> n;
    string s;
    cin >> s;
    int cnt = 0, ans = 0;
    for (int i = 0; i < n; i++) {
        if (s[i] == '(') cnt ++;
        else if(cnt > 0) cnt --, ans ++;
    }
    cout << ans << '\n';
}

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

> *一开始甚至没想到第二种做法（*

G. Still No Money?

博弈论。

首先如果 $x \leq k$，那先手开局直接就能全拿光，然后后手没得拿，所以此时先手必胜。

因此如果 $x > k$，双方一定都会尽量避免拿完之后出现 $x' \leq k'$ 的情况，否则立刻输掉。而这一点总是可以做到的，此时拿的人只需选择只拿 1 根即可：这样 $x' = x - 1, k' = k - 1$，所以拿完之后必然有 $x' > k'$。也就是说这种情况下永远不会有 $x \leq k$ 然后竹签被拿完的情况，所以最终胜利条件就由 k 决定： k 为奇数时先手拿到最后一根，k 为偶数时后手拿到最后一根。

综上，先手必胜当且仅当 $x \leq k$ 或 k 为奇数。

时间复杂度: $O(1)$

Std (games)

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

using namespace std;

void solve() {
    long long x, k;
    cin >> x >> k;
    cout << ((x <= k || k % 2 == 1) ? "OC\n" : "KP\n");
}

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

> *骗骗更健康*

H. Yet Another Tree Problem

数学，前缀和，二分。

先把方差表达式里面那个完全平方乘开，得 $\frac{k\sum_{i=1}^{k}b_i^2-(\sum_{i=1}^{k}b_i)^2}{k^2}$。

先把 a_i 从小到大排序，考虑枚举 k。注意到 "先砍光最矮的 $n-k$ 棵树，再去修其他树的高度" 的策略能涵盖所有 "砍得剩下 k 棵树" 的情况。因为假设两棵树的高度分别为 a_i,a_j 且 $a_i<a_j$，如果说某个策略下 a_i 砍成了 a_i' 而 a_j 砍成了 0，那同样砍了 $a_i+a_j-a_i'$ 次的情况下，我们也可以把 a_i 砍成 0 然后把 a_j 砍成 a_i'，效果是一样的。也就是说枚举 k 时，我们只需要考虑保留后 k 棵树的情况就足够了。

现在考虑怎么求 k 确定时的答案。观察方差表达式中被减掉的一项，我们发现 $(\sum_{i=1}^{k}b_i)^2$ 是个定值（即 $(\sum a_i-s)^2$），所以我们只需要最小化前一项，即最小化剩下的 $n-k$ 个数的平方和。由于对 a_i 砍一次之后平方和会减小 $2a_i-1$，所以每次贪心地砍最大的 a_i 就能得到最优解。

但这题 s 和 n 都很大，所以暴力砍会T飞，需要设法快速求出砍完之后的平方和。

砍完之后的 a_i 可以分为两部分，较小的一部分是完全没砍过的，较大的一部分是砍过的，并且由于贪心时每次都砍最大的数，所以这些数一定是平均分配的（但可能有余数）。我们的目标是快速找到这两部分的分界点，并分别计算两部分的平方和，然后加起来即可得出答案。

预处理出 a_i 的前缀和 pre_i。我们知道剩余能砍的次数是 $L=s-\sum_{i=1}^{n-k}pre_i$。考虑二分找最后一个没被砍的位置 mid，因为按照贪心策略每次都砍最大的数，这个数没砍意味着 $[mid+1,mid+n]$ 这些位置上的每一个数最后都不能小于 a_{mid}，否则与贪心策略矛盾。于是考虑求出 $[mid+1,mid+n]$ 这些数的和（即 pre_n-pre_{mid}），看它砍 L 次之后能不能让每一个数都大于 a_{mid}，也就是判断总和是不是大于等于 $a_{mid}\times(n-mid)+L$。如果大于等于，说明右边够砍，分界点可能小了，令 $l=mid$；如果小于，说明右边不够砍，分界点一定大了，令 $r=mid-1$。

找到分界点之后就很好做了：左边的部分没砍过，预处理一下平方的前缀和即可；右边的部分一定是均分的（因为每一项都砍过，换言之这个过程中每一项都当过最大值，所以最小值与最大值相差必定不超过 1），剩余的这些数之和为 $pre_n-pre_{mid}-L$，平均一下就得到剩下这些数最小值为 $\lfloor\frac{pre_n-pre_{mid}-L}{n-mid}\rfloor$，最大值为 $\lfloor\frac{pre_n-pre_{mid}-L}{n-mid}\rfloor+1$，由于余数为 $(pre_n-pre_{mid}-L)\bmod(n-mid)$，所以取到最大值的数也就有

$(pre_n - pre_{mid} - L) \bmod (n - mid)$ 个，计算一下加起来就行。复杂度的瓶颈在二分上，所以总的时间复杂度是 $O(n \log n)$。

时间复杂度: $O(n \log n)$

Std (math, prefix sums, binary search)

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

using namespace std;

void solve() {
    int n;
    long long s;
    cin >> n >> s;
    vector<long long> a(n + 1), pre1(n + 1), pre2(n + 1);
    for (int i = 1; i ⩽ n; i++) cin >> a[i];
    sort(a.begin() + 1, a.end());
    for (int i = 1; i ⩽ n; i++) {
        pre1[i] = pre1[i - 1] + a[i];
        pre2[i] = pre2[i - 1] + a[i] * a[i];
    }
    long double ans = 1.0e30l;
    for (int i = 0; i < n; i++) {
        //枚举最后一棵被彻底砍掉的树
        if (pre1[i] > s) break;
        long long lst = s - pre1[i];
        int l = i + 1, r = n;
        while (l < r) {
            // 贪心 用二分找在贪心策略下最后一棵被砍过的树
            int mid = (l + r) >> 1;
            if ((n - mid) * a[mid] + lst ⩽ pre1[n] - pre1[mid])l = mid + 1;
            else r = mid;
        }
        long long sq1 = pre2[l - 1] - pre2[i];  // 没被砍的树高平方和
        long long tot = pre1[n] - pre1[l - 1] - lst;  // 被砍过的树的总高度
        long long height = tot / (n - l + 1);  // 被砍过的树的最小高度
        long long sq2 = (height + 1) * (height + 1) * (tot % (n - l + 1)) + height * height * (
                        (n - l + 1) - tot % (n - l + 1));  //被砍过的树高平方和
```

```
        long double ave_sq = (long double) (pre1[n] - s) * (pre1[n] - s) / (n - i) / (n - i);
        ans = min(ans, (long double) (sq1 + sq2) / (n - i) - ave_sq);
    }
    cout << fixed << setprecision(6) << ans << '\n';
}

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

> *虽然看起来有点吓人 但其实算清楚了就不难写*

I. DJ Mr. Spin

计算几何，离散化，差分，STL。

这里给出两种解法。

解法1

首先，对于圆中的每一个点，我们都可以分别算出 P 开始移动的时刻 t 的可行时间范围。这需要考虑：

1. 向量 OP 足够长，可以 "碰到" 这个点；
2. 向量 OP 不会和圆相交

如果我们能求出这些范围，那么我们只需离散化后差分，就能找到可以到达最多点数的对应开始时刻。

我们将逆时针旋转的圆固定，视为点 P 在顺时针旋转，不妨考虑处理时刻 t 对应于向量 OP 与坐标轴的夹角，找到合适夹角后除掉 v_1 即可。

由于题目限制，P 开始移动后，旋转的圈数一定 < 1。换句话说，向量 OP 的夹角区间不会超过 360°。

题目需要求出满足条件的最小时刻，那么我们就得让夹角区间位于 $[0^\circ, 360^\circ)$ 内。通过精细的计算，我们可以将区间的左端点控制在 $[0^\circ, 360^\circ)$ 内，右端点控制在 $[0^\circ, 720^\circ)$，如果右端点超出边界，那么将区间拆分为两段即可。简单来说，类比一下取模即可。

解法2

按照极角序给圆中的每个点排序。考虑经典的拆环做法，也就是把原数组复制一份，变成长度 $2n$ 的链。

同样假定圆是固定的而向量 OP 在转，枚举向量 OP 碰到的第一个点，这样一共有至多 $2n$ 种可能的出发时间（碰到第 $n+1 \sim 2n$ 个点相当于出发前等了一会，出发后转了一圈又回来了）。

把这 $2n$ 种可能的时间离线下来从小到大排序。每个出发时间能扫到的极角是一个固定长度的区间；而对于每个点，知道其在极坐标系下的 θ 和 r 坐标后就能求出碰到这个点的 点 P 最晚的出发时间。那么求这些向量 OP 能碰到的点的个数就相当于处理 $2n$ 个询问，每次询问需要求出给定区间中大于给定值的数有多少。

由于排完序后询问值单调递增且询问区间单调右移，用双指针加一个小根堆维护当前询问的向量 OP 能扫到的点即可。然后我们发现如果向量 OP 碰到最后一个点时还没有碰到圆，那向量 OP 就可以再提前一点出发。所以再用一个大根堆维护向量 OP 能扫到的最后一个点，在更新答案时把出发时间设成恰好碰到这个点时结束的最早时间。记得判负数时间，否则会 WA（

时间复杂度: $O(n \log n)$

Std (1, computational geometry, hashing, difference)

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

using namespace std;

const double PI = acos(-1.0), eps = 1e-9;

void solve() {
    double r, v1, v2;
    cin >> r >> v1 >> v2;
    int n;
    cin >> n;
    vector<pair<int, int>> p(n);
    for (auto &[x, y] : p) cin >> x >> y;
    vector<pair<double, double>> lr;
    for (auto &[x, y] : p) {
        double len = sqrt(1.0 * (1LL * x * x + 1LL * y * y));
        double angle;
        if (x == 0) angle = y > 0 ? PI / 2 : PI * 3 / 2;
        else if (y >= 0) angle = atan2(y, x);
        else angle = 2 * PI + atan2(y, x);
        angle = 2 * PI - angle;  // 角处理成反的（与 x 轴正半轴的顺时针夹角）
        angle -= len / v2 * v1;
        double angle_len = (r - len) / v2 * v1;
        if (angle < -eps) angle += 2 * PI;
        if (angle - angle_len < -eps) {
            lr.emplace_back(0.0, angle);
            lr.emplace_back(2 * PI + angle - angle_len, 2 * PI);
        }else {
            lr.emplace_back(angle - angle_len, angle);
        }
    }
    vector<double> v;
```

```
    for (auto &[l, r] : lr) v.emplace_back(l), v.emplace_back(r);
    sort(v.begin(), v.end());
    int m = unique(v.begin(), v.end()) - v.begin();
    auto idx = [&](double x){
        return lower_bound(v.begin(), v.begin() + m, x) - v.begin();
    };
    vector<int> d(m + 2);
    for (auto &[l, r] : lr) d[idx(l)] ++, d[idx(r) + 1] --;
    int mx = d[0];
    double ans = v[0];
    for (int i = 1; i ⩽ m; i++) {
        d[i] += d[i - 1];
        if (d[i] > mx) {
            ans = v[i];
            mx = d[i];
        } else if (d[i] == mx) {
            ans = min(ans, v[i]);
        }
    }
    cout << fixed << setprecision(3) << ans / v1 << '\n';
}

int main() {
    solve();
    return 0;
}
```

Std (2, computational geometry, two-pointers, data structure)

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

using namespace std;

constexpr long double PI = acos(-1.0), eps = 1e-9;

void solve() {
    int n;
    long double R, v1, v2;
    cin >> R >> v1 >> v2 >> n;
    R /= v2;
    long double rd = 2 * pi / v1;
```

```
    vector<long double> x(n + 1), y(n + 1), t(2 * n + 1), len(2 * n + 1);
    for (int i = 1; i ⩽ n; i++) {
        cin >> x[i] >> y[i];
        len[i] = sqrt(x[i] * x[i] + y[i] * y[i]);
        t[i] = atan2(y[i], x[i]);
        if (t[i] > -eps)
            t[i] -= 2 * pi;
        t[n + i] = t[i] - 2 * pi;
        len[i] /= v2;
        len[n + i] = len[i];
        t[i] /= -v1;
        t[n + i] /= -v1;
    }
    int l = 0, r = 0;
    vector<pair<long double, long double>> vec(2 * n + 1);
    for (int i = 1; i ⩽ 2 * n; i++) vec[i] = {t[i], len[i]};
    sort(vec.begin() + 1, vec.end());
    vector<pair<long double, int>> query(2 * n + 1);
    for (int i = 1; i ⩽ 2 * n; i++) {
        query[i] = {vec[i].first - vec[i].second, i};
        if (query[i].first < -eps) query[i].first += rd;
    }
    sort(query.begin() + 1, query.end());
    int score = 0, cnt = 0;
    long double ans = 0;
    priority_queue<pair<long double, int>, vector<pair<long double, int>>, greater<pair<long double, int>>> q;
    priority_queue<pair<long double, int>> q2;
    vector<int> ok(2 * n + 1);
    for (int i = 1; i ⩽ 2 * n; i++) {
        long double ti = query[i].first;
        while (r < 2 * n && vec[r + 1].first - R - ti < eps) {
            ++r;
            q.push({vec[r].first - vec[r].second, r});
            q2.push({vec[r].first, r});
            ok[r] = 1;
            ++cnt;
        }
        while (!q.empty() && q.top().first - ti < -eps) {
            cnt -= ok[q.top().second];
```

```
            ok[q.top().second] = 0;
            q.pop();
        }
        while (!q2.empty() && !ok[q2.top().second]) q2.pop();
        ti = max(0.0l, q2.top().first - R);
        if (cnt > score) {
            score = cnt;
            ans = ti;
        } else if (cnt == score)
            ans = min(ans, ti);
    }
    cout << fixed << setprecision(3) << ans;
}

int main() {
    solve();
    return 0;
}
```

> *旋转吧，旋转吧，旋转吧！*

J. Spirit of Cola.

BFS。

考虑到状态数较小（每个杯子最多 301 种状态，合法状态只有 301^3 个，而且很多状态根本搜不到），考虑用 BFS。用三元组 (a, b, c) 表示当前三个杯子中可乐的含量，令 $dp_{a,b,c}$ 为到达该状态所需的最小步数。转移只有两种：

- 倒可乐操作：枚举被操作的杯子 (i, j)，计算可以倒的最小值 $\min(w_i, c_j - w_j)$，更新新状态。
- 清空操作：找到当前含量最少的杯子，将其清空。

BFS 搜索，直至到达目标状态（至少一个杯子里有 t 升可乐）或搜完所有可达状态都没找到为止。

由于本题需要输出方案，因此转移时需要记录一下当前状态是从哪里转移过来的，输出时递归一下倒着输出即可。

时间复杂度: $O(n^3)$

Std (BFS)

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

using namespace std;

void solve() {
    array<int, 3> c, w;
    int t;
    cin >> c[0] >> c[1] >> c[2] >> w[0] >> w[1] >> w[2] >> t;
    vector dp(c[0] + 1, vector(c[1] + 1, vector<int>(c[2] + 1, -1)));
    vector last(c[0] + 1, vector(c[1] + 1, vector<array<int, 3>>(c[2] + 1, {-1, -1, -1})));

    dp[w[0]][w[1]][w[2]] = 0;
    queue<array<int, 3>> q;
    q.push({w[0], w[1], w[2]});

    while (q.size()) {
        auto x = q.front();
        q.pop();
```

```
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (i == j)continue;
                int d = min(x[i], c[j] - x[j]);
                array<int, 3> y = x;
                y[i] -= d;
                y[j] += d;
                if (dp[y[0]][y[1]][y[2]] == -1) {
                    dp[y[0]][y[1]][y[2]] = dp[x[0]][x[1]][x[2]] + 1;
                    last[y[0]][y[1]][y[2]] = x;
                    q.push(y);
                }
            }
        }
        int mi = 1e9;
        for (int i = 0; i < 3; i++) if (x[i]) mi = min(mi, x[i]);
        for (int i = 0; i < 3; i++) {
            if (x[i] == mi) {
                array<int, 3> y = x;
                y[i] = 0;
                if (dp[y[0]][y[1]][y[2]] == -1) {
                    dp[y[0]][y[1]][y[2]] = dp[x[0]][x[1]][x[2]] + 1;
                    last[y[0]][y[1]][y[2]] = x;
                    q.push(y);
                }
            }
        }
    }
    int cnt = 1e9;
    array<int, 3> res;
    for (int i = 0; i <= c[0]; i++) {
        for (int j = 0; j <= c[1]; j++) {
            for (int k = 0; k <= c[2]; k++) {
                if ((i == t || j == t || k == t) && dp[i][j][k] != -1 &&
cnt > dp[i][j][k]) {
                    cnt = dp[i][j][k];
                    res = {i, j, k};
                }
            }
        }
    }
```

```
59
60      if (cnt == 1e9) cout << -1 << '\n';
61      else {
62          cout << cnt << '\n';
63          auto pr = [&](auto self, int x, int y, int z) -> void {
64              if (x == w[0] && y == w[1] && z == w[2]) return;
65              auto it = last[x][y][z];
66              self(self, it[0], it[1], it[2]);
67              cout << x << " " << y << " " << z << '\n';
68          };
69          pr(pr, res[0], res[1], res[2]);
70      }
71  }
72
73  int main() {
74      solve();
75      return 0;
76  }
```

> *：你说得对，但可乐真不能无糖*
>
> *：这么好喝，居然无糖？*
>
> *：无糖也好喝，不信你逝逝（*

K1. MST(a.k.a. Most Shortened Terms)

签到，模拟。

读取输入后判断三个单词是否分别以M,S,T开头即可。

时间复杂度: $O(1)$

Std (implementation)

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

using namespace std;

void solve() {
    int n, m, p;
    cin >> n >> m >> p;
    string a, b, c;
    cin >> a >> b >> c;
    cout << (a[0] == 'M' && b[0] == 'S' && c[0] == 'T' ? "YES\n" :
"NO\n");
}

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

> *多一题签到，你开心我开心*

K2. MST(a.k.a. Most Stupid Technique)

最小生成树，DFS，LCA，树上差分。

首先因为每条边的边权唯一，所以最小生成树一定是唯一的。那不妨先把MST跑出来。

接下来考虑把原问题反过来问：何时这个最小生成树是一棵合法的DFS树？

因为给定的是无向图，而无向图上的DFS树一定满足所有的非树边都是返祖边（即非树边的两个端点在树上是祖先-后代关系），所以如果以某个点为根的时候某条非树边是横叉边，那这个点就不可能是答案。

另一方面，如果所有的非树边都是返祖边（即最小生成树是合法的DFS树），那最小生成树一定是可以通过给定的这个算法跑出来的。因为根据最小生成树的性质，任何非树边与树上简单路径组成的环中，非树边的边权一定是最大的（否则把边权最大的边从生成树里删掉然后把这条非树边加进去，得到的生成树边权总和会变小，与最小生成树的性质矛盾）。

所以对任何一条非树边 (u, v)，在遍历这条边之前算法一定会先把最小生成树上面 $u \to v$ 的这条路径搜完，然后 u 和 v 连通之后这条边就加不进去了，也就是加进去的全是树边。换言之，以 i 为根时算法求出最小生成树的充要条件即 i 为根时所有的非树边都是返祖边。

但暴力判断所有边是不是返祖边的复杂度至少是 $O(n^2)$ 的，需要优化。考虑对每一条非树边，维护加入这条边之后还有哪些点符合条件。可以发现对 (u, v) 这条非树边，满足条件的点一定是树上没有 "夹在" u 和 v 之间的那些点。随便取一个点作为最小生成树的根，手玩一下可以得出结论：

1. 如果 u 和 v 在树上是祖先后代关系，不妨设 u 是 v 的祖先。令 w 是从 u 到 v 路径上的第二个点，则满足条件的是在 v 子树或不在 w 子树内的所有点；
2. 如果 u 和 v 在树上没有祖先后代关系，则满足条件的是在 u 子树或在 v 子树内的所有点。

注意到维护的时候是以子树为单位进行的，所以树上差分一下，找到加入所有非树边之后仍然满足条件的点即可。找 w 时需要多写一个倍增/树剖快速跳祖先。

时间复杂度: $O(n \log n + m)$

Std (min spanning tree MST, dfs, lca, differential on tree)

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

using namespace std;

```

```
5  void solve() {
6      int n, m;
7      cin >> n >> m;
8      vector<pair<int, int>> e(m + 1);
9      vector<vector<int>> e2(n + 1);
10     int u, v;
11     for (int i = 1; i ⩽ m; i++) {
12         cin >> u >> v;
13         if (u > v) swap(u, v);
14         e[i] = {u, v};
15     }
16     vector<int> fa(n + 1);
17     for (int i = 1; i ⩽ n; i++) fa[i] = i;
18     auto find = [&](auto self, int x) → int {
19         if (fa[x] == x) return fa[x];
20         return fa[x] = self(self, fa[x]);
21     };
22     set<pair<int, int>> st;
23     for (int i = 1; i ⩽ m; i++) {
24         u = e[i].first;
25         v = e[i].second;
26         if (find(find, u) ≠ find(find, v)) {
27             fa[fa[u]] = fa[v];
28             e2[u].push_back(v);
29             e2[v].push_back(u);
30             st.insert({u, v});
31         }
32     }
33     vector<vector<int>> ffa(n + 1, vector<int>(20));
34     vector<int> dep(n + 1);
35     auto dfs = [&](auto self, int cur, int fa) → void {
36         dep[cur] = dep[fa] + 1;
37         ffa[cur][0] = fa;
38         for (int j = 1; (1 << j) ⩽ dep[cur]; j++) ffa[cur][j] =
   ffa[ffa[cur][j - 1]][j - 1];
39         for (auto j : e2[cur]) {
40             if (j == fa) continue;
41             self(self, j, cur);
42         }
43     };
44     auto lca = [&](int u, int v) → int {
```

```
        if (dep[u] == dep[v]) return -1;
        int dif = dep[u] - dep[v] - 1;
        for (int j = 19; j >= 0; j--) if (dif & (1 << j)) u = ffa[u][j];
        if (ffa[u][0] == v) return u;
        return -1;
    };
    dep[0] = -1;
    dfs(dfs, 1, 0);
    vector<int> d(n + 1);
    for (int i = 1; i <= m; i++) {
        if (st.count(e[i])) continue;
        u = e[i].first, v = e[i].second;
        if (dep[u] < dep[v]) swap(u, v);
        int k = lca(u, v);
        if (k == -1) ++d[u], ++d[v];
        else ++d[1], --d[k], ++d[u];
    }
    vector<int> ans(n + 1);
    auto dfs2 = [&](auto self, int cur, int fa) -> void {
        ans[cur] = ans[fa] + d[cur];
        for (auto j: e2[cur]) {
            if (j == fa) continue;
            self(self, j, cur);
        }
    };
    dfs2(dfs2, 1, 0);
    for (int i = 1; i <= n; i++) {
        ans[i] = (ans[i] == m - n + 1);
        cout << ans[i];
    }
}

int main() {
    solve();
    return 0;
}
```

> *比较综合的一道题，难点在于反过来思考*

L. We Luv Stamina

模拟。

是的，没别的了，就是纯模拟。照着题面给的流程做就行了。

从下往上枚举每一行，判断底下一行左右手分别有没有键即可。

时间复杂度: $O(n)$

Std (implementation)

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

using namespace std;

void solve() {
    int n;
    cin >> n;
    vector<string> s(n);
    for (int i = 0; i < n; i++) {
        cin >> s[i];
    }
    for (int i = n - 2; i >= 0; i--) {
        if (s[i][0] == '1' && s[i + 1][1] == '1') swap(s[i][0], s[i + 1][0]);
        if (s[i][1] == '1' && s[i + 1][0] == '1') swap(s[i][1], s[i + 1][1]);
        if (s[i][2] == '1' && s[i + 1][3] == '1') swap(s[i][2], s[i + 1][2]);
        if (s[i][3] == '1' && s[i + 1][2] == '1') swap(s[i][3], s[i + 1][3]);
    }
    for (int i = 0; i < n; i++) {
        cout << s[i] << '\n';
    }
}

int main() {
    solve();
```

```
25      return 0;
26  }
```

> *不怕你不会做，就怕你不敢写（笑）*

M. The Journey Onwards...

排序，贪心。

先把给出的点从小到大排序，从 0 开始往后跳，每次检查下一个点与当前点的距离。如果不交技能就能跳过去，那就直接跳；如果交了技能才能过，那我们肯定希望这次技能开完之后跳的距离尽可能长。所以此时往后扫，找到最后一个开技能可以跳到的点，然后跳上去并更新答案。当然不管开不开技能都跳不过去的话就无解。

注意三种情况判定的顺序：数据里的 x 是可以大于 y 的，所以一定要先判断不交技能能不能过，不能的话再考虑交技能的情况。

时间复杂度: $O(n \log n)$

Std (sortings, greedy)

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

using namespace std;

void solve() {
    int n, x, y;
    cin >> n >> x >> y;
    vector<int> a(n + 1);
    for (int i = 1; i ⩽ n; i++) cin >> a[i];
    sort(a.begin(), a.end());
    int ans = 0;
    for (int i = 1; i ⩽ n; i++) {
        int d = a[i] - a[i - 1];
        if (d ⩽ x) continue;
        if (d > y) {
            cout << -1 << '\n';
            return;
        }
        int cur = i - 1;
        while (i ⩽ n && a[i] - a[cur] ⩽ y) ++i;
        --i;
        ++ans;
    }
    cout << ans << '\n';
```

```
}

int main() {
    solve();
    return 0;
}
```

> *代码不长，细节不少*