Problem A. 从前有棵神樱树

知识点：图论

对于任意一条边来说，产生的贡献为断开所有与这条边元素印记相同的边后，该边两个端点所在的连通块大小的乘积，而该数据在题目中已给出。

于是，本题答案为 $\sum_{i=1}^{n} d_i \cdot e_i$。

Solution

```
#include <cstdint>
#include <iostream>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

int main() {
    i64 n;
    cin >> n;
    i64 ans = 0;
    for (auto i = 1; i < n; ++i) {
        i64 d, e;
        cin >> d >> e;
        ans += d * e;
    }
    cout << ans << endl;
    return 0;
}
```

Problem B. 提瓦特大陆·电气传导之谜

知识点：可撤销并查集

放置与移除继电石都可能改变连通性并改变总能量，且移除的顺序严格遵循"后放先移"的栈顺序，因此可以利用「可撤销并查集」的数据结构维护"加入边"和"撤销加入边"的操作。

遇到放置操作：并查集合并，并把合并前的必要信息压栈，计算本次新边贡献的能量，累加到当前总能量里。

遇到移除操作：从栈中弹出刚才合并时存的信息，进行回滚，恢复到移除前的状态。同时减去被移除的继电石对应的能量贡献。

开数组记录每个时刻的总能量，最后根据 T，输出答案即可。

Solution

```
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

template <class T>
constexpr bool chmax(T& x, T y) {
    if (y > x) {
        x = y;
        return true;
    }
    return false;
}

template <class T>
constexpr bool chmin(T& x, T y) {
    if (y < x) {
        x = y;
        return true;
    }
    return false;
}

struct RevocableDSU {
    vector<int>             par, siz;
    vector<pair<int, int>> stk;

    explicit RevocableDSU(int n) {
        par.resize(n + 1);
        siz.assign(n + 1, 1);
        iota(par.begin(), par.end(), 0);
        stk.clear();
    }

    int find(int x) {
```

```
        while (x != par[x]) {
            x = par[x];
        }
        return x;
    }

    bool same(int x, int y) { return find(x) == find(y); }

    bool merge(int x, int y, i64& sum) {
        x = find(x), y = find(y);
        if (x == y) {
            stk.emplace_back(-1, -1);
            return false;
        }
        if (siz[x] < siz[y]) {
            std::swap(x, y);
        }
        sum    += siz[x] - siz[y];
        siz[x] += siz[y];
        par[y]  = x;
        stk.emplace_back(x, y);
        return true;
    }

    int size(int x) { return siz[find(x)]; }

    size_t version() const { return stk.size(); }

    void rollback(i64& sum) {
        auto [x, y] = stk.back();
        stk.pop_back();
        if (x != -1 && y != -1) {
            siz[x] -= siz[y];
            sum    -= siz[x] - siz[y];
            par[y]  = y;
        }
    }
};

int main() {
    std::ios::sync_with_stdio(false);
```

```
    std::cin.tie(nullptr);

    int n, m, q, r;
    std::cin >> n >> m >> q >> r;

    std::vector<std::pair<int, int>> edges(m + 1);
    for (int i = 1; i <= m; i += 1) {
        auto& [u, v] = edges[i];
        std::cin >> u >> v;
    }

    i64                sum = 0;
    RevocableDSU       dsu(n);
    std::vector<i64> ans(q + 1);
    for (int i = 1; i <= q; i += 1) {
        int o, p;
        std::cin >> o >> p;
        if (o == 1) {
            auto [u, v] = edges[p];
            dsu.merge(u, v, sum);
        } else {
            auto [u, v] = edges[p];
            dsu.rollback(sum);
        }
        ans[i] = sum;
    }

    for (int i = 1; i <= r; i += 1) {
        int t;
        std::cin >> t;
        std::cout << ans[t] << '\n';
    }
    return 0;
}
```

Problem C. qfl_zzz的数组询问

知识点：dp，矩阵乘法，拆平方贡献

先考虑对于一个数组 a，如何求出 $\sum_{i=1}^{n}\sum_{j=i}^{n}(\sum_{k=i}^{j}a_k)^2$ 的值。

根据式子不难发现，我们要去计算所有"连续子数组和"的平方和；$O(n^2)$ 暴力的方法是显然的：枚举子

数组的左右端点 l,r，那么可以用前缀和 $O(1)$ 地去计算总和，平方之后累加。

考虑优化，如果只枚举其中一个端点（右端点），那么我们需要将答案拆分成"以每个位置 i 结尾的子数组平方和"之和，接下来考虑怎么去对于一个位置 i 求出"以 i 结尾的子数组平方和"。

假设一个区间 $[j,i-1]$ 的平方和是 x^2，那么将 a_i 加入后，区间 $[j,i]$ 的平方和就是：

$$(x+a_i)^2 = x^2 + 2a_ix + a_i^2$$

对于所有以 $i-1$ 结尾的区间的平方和，记为 $\sum x^2$，将 a_i 加入这些区间后，所有以 i 结尾的区间的平方和就是：

$$(\sum x^2) + 2a_i(\sum x) + a_i^2c_i$$

其中，c_i 表示以 i 结尾的区间个数。

定义 dp_i 表示以 i 结尾的区间平方和，f_i 表示以 i 结尾的区间和，那么按照上述变化可以得到转移方程：

$$\begin{cases} dp_{i-1} + 2a_if_{i-1} + a_i^2(c_{i-1}+1) & \to dp_i \\ f_{i-1} + a_i(c_{i-1}+1) & \to f_i \\ c_{i-1}+1 & \to c_i \end{cases}$$

至此，可以 $O(n)$ 回答每个询问，答案是 $\sum_{i=1}^{n} dp_i$。

考虑多个询问怎么快速回答，可以发现每次删除掉区间 $[l,r]$，相当于就是将前缀 $[1,l-1]$ 与后缀 $[r+1,n]$ 合并在一起；问题在于如何在每次询问的时候快速合并前后缀信息，即快速地进行 dp 的转移。

定义 s_i 表示 $\sum_{j=1}^{i} dp_j$，将 s_i 加入到转移方程中：

$$\begin{cases} s_{i-1} + dp_{i-1} + 2a_if_{i-1} + a_i^2(c_{i-1}+1) & \to s_i \\ dp_{i-1} + 2a_if_{i-1} + a_i^2(c_{i-1}+1) & \to dp_i \\ f_{i-1} + a_i(c_{i-1}+1) & \to f_i \\ c_{i-1}+1 & \to c_i \end{cases}$$

将转移方程写成矩阵的形式：

$$\begin{bmatrix} 1 & 1 & 2a_i & a_i^2 & a_i^2 \\ 0 & 1 & 2a_i & a_i^2 & a_i^2 \\ 0 & 0 & 1 & a_i & a_i \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} s_{i-1} \\ dp_{i-1} \\ f_{i-1} \\ c_{i-1} \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} s_i \\ dp_i \\ f_i \\ c_i \\ 1 \end{bmatrix}$$

可以发现，对于每一个位置 i，对应的转移矩阵只和 a_i 的值有关；至此，我们可以预处理每个前缀的矩阵乘积 pre_i，以及每个后缀的矩阵乘积 suf_i；然后对于每个询问 l, r，将前缀矩阵积 pre_{l-1} 与后缀矩阵积 suf_{r+1} 相乘即可快速得到答案。

记矩阵的大小为 $k \times k$，其中 $k = 5$，一次矩阵乘法的时间复杂度是 $O(k^3)$。

总时间复杂度 $O(k^3 n)$。

Solution

```
#include <array>
#include <cstdint>
#include <iostream>


using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

constexpr i64 p = 998244353;

struct mtx {
    std::array<std::array<int, 5>, 5> a{};

    mtx() = default;
    void eye() {
        for (int i = 0; i < 5; ++i) {
            a[i][i] = 1;
        }
    }
    void set(int x) {
        eye();
        a[0][1] = a[3][4] = 1;
        a[2][3] = a[2][4] = x % p;
        a[0][2] = a[1][2] = 2 * x % p;
        a[0][3] = a[0][4] = a[1][3] = a[1][4] = 1ll * x * x % p;
```

```
    }
};
mtx operator*(mtx& x, mtx& y) {
    mtx res;
    for (int i = 0; i < 5; ++i) {
        for (int j = i; j < 5; ++j) {
            for (int k = 0; k < 5; ++k) {
                res.a[i][j] += 1ll * x.a[i][k] * y.a[k][j] % p;
                res.a[i][j] %= p;
            }
        }
    }
    return res;
}

int n, q, x, l, r;
mtx pre[500005], suf[500005];
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> x;
        pre[i].set(x);
        suf[i].set(x);
    }

    pre[0].eye();
    for (int i = 1; i <= n; ++i) {
        pre[i] = pre[i] * pre[i - 1];
    }

    suf[n + 1].eye();
    for (int i = n; i >= 1; --i) {
        suf[i] = suf[i + 1] * suf[i];
    }

    cin >> q;
    while (q--) {
        cin >> l >> r;
```

```
        cout << (suf[r + 1] * pre[l - 1]).a[0][4] << "\n";
    }
}
```

Problem D. 回文询问

知识点：回文自动机

首先对字符串构建回文自动机，在构建的同时记录每个右端点对应的最长回文串在回文自动机上的位置。虽然我们记录了右端点的最长回文串，但题目要求限制了左端点，实际限制了长度的范围。例如某次询问 3 到 10，实际要求的是右端点在 10 且长度小于等于 8 的最长回文串。所以仅仅记录最长的回文串还是无法解决问题。

暴力的做法是不断地去跳 $fail$ 指针，直到长度符合条件，但是这种做法容易被卡成 $O(n^2)$，一个简单的例子就是 $aaaaaaa...$ 这样的字符串。为了解决不断跳 $fail$ 指针导致的时间复杂度过大的问题，我们需要对回文自动机的 $fail$ 指针建一颗 $fail$ 树，在 $fail$ 树上进行倍增即可解决。

创建回文自动机的时间复杂度为 $O(n)$，对于单次询问的时间复杂度为 $O(\log n)$，总的时间复杂度为 $O(n + q \cdot \log n)$。

Solution

```
#include <cstdint>
#include <iostream>
#include <vector>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

struct PAM {
    std::vector<int>          Last, len, fail;
    std::vector<vector<int>> fa;
    std::vector<vector<int>> son;
    std::vector<char>         s;
    int                       tot{}, last{}, cnt{};
    int                       length{};

    int node(int l) {
        tot++;
        len[tot] = l;
        return tot;
    }
```

```
void clear() {
    last = cnt = 0;
    tot        = -1;
    s.resize(length + 7);
    Last.resize(length + 7);
    len.resize(length + 7);
    fail.resize(length + 7);
    son  = vector(length + 7, vector(26, 0));
    fa   = vector(length + 7, vector(23, -1));
    s[0] = '#';
    node(0);
    node(-1);
    fail[0] = 1;
}
int getfail(int x) {
    while (s[cnt - len[x] - 1] != s[cnt]) x = fail[x];
    return x;
}
void insert(char c) {
    s[++cnt] = c;
    int now  = getfail(last);
    if (!son[now][c - 'a']) {
        int x = node(len[now] + 2), y = son[getfail(fail[now])][c - 'a'];
        fail[x]           = y;
        fa[x][0]          = y;
        son[now][c - 'a'] = x;
        for (int i = 1; i <= 20; i++) {
            int half = fa[x][i - 1];
            if (half != -1) fa[x][i] = fa[half][i - 1];
        }
    }
    last = son[now][c - 'a'];
}
void build(string t) {
    length = t.size();
    clear();
    t = "#" + t;
    for (int i = 1; i <= length; i++) {
        insert(t[i]);
        Last[i] = last;
    }
```

```
    }
    int query(int l, int r) {
        int u = Last[r];
        if (len[u] <= r - l + 1) return len[u];
        for (int i = 20; i >= 0; i--) {
            int v = fa[u][i];
            if (v != -1 && len[v] > r - l + 1) u = v;
        }
        u = fa[u][0];
        return len[u];
    }
};

constexpr auto N = 1000010;
int            l[N], r[N];

void solve() {
    PAM    pam;
    int    n, q;
    string s;
    cin >> n >> q >> s;
    for (int i = 1; i <= q; i++) cin >> l[i] >> r[i];
    pam.build(s);
    for (int i = 1; i <= q; i++) {
        int res = pam.query(l[i], r[i]);
        cout << res << "\n";
    }
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    int T;
    T = 1;
    while (T--) solve();
}
```

Problem E. 七彩珍珠

知识点：线段树

首先，题目进行区间修改以及区间询问的操作，所以可以想到线段树。同时，题目中的颜色种类并不会太多，所以可以考虑对每种颜色建一颗线段树。每一种颜色的线段树主要是对 01 序列进行维护，这里的 01 序列指，如果某个位置的珍珠的颜色为这一种颜色，那么这一位为 1，否则为 0。当然，要注意我们并不是把 01 序列存在了线段树节点中，每个线段树节点我们只需要存一下左右儿子是哪个节点，以及这个区间中 1 的个数 sum 即可。

对每一种颜色都创建一棵线段树的时间和空间复杂度为 $O(4 \cdot n \cdot m)$，对于每次修改和询问的时间复杂度是 $O(\log n)$，所以总的时间复杂度为 $O(4 \cdot n \cdot m + q \cdot \log n)$。

我们拿样例进行解释：

原数组：

1 1 4 5 1 4

对于颜色 1 而言的 01 序列为：

1 1 0 0 1 0

对于颜色 4 而言的 01 序列为：

0 0 1 0 0 1

对于颜色 5 而言的 01 序列为：

0 0 0 1 0 0

那么每一次的交换操作实际上就是交换两颗颜色的线段树中的某些节点。

解决了交换操作，区间询问操作实际上就相当于求 01 序列中某一段的和，和普通线段树求区间和的做法类似。

Solution

```
#include <cstdint>
#include <iostream>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

constexpr int N = 100010, M = N * 4 * 30, K = 30;
struct Node {
    int l, r, sum;
};

int  a[N], idx, root[K];
```

```
Node tr[M];

void pushup(int u) { tr[u].sum = tr[tr[u].l].sum + tr[tr[u].r].sum; }

int build(int l, int r, int k) {
    if (l == r) {
        tr[idx] = {-1, -1, 0};
        if (a[l] == k) tr[idx].sum = 1;
        return idx++;
    }
    int u   = idx++;
    tr[u]   = {-1, -1, 0};
    int mid = (l + r) >> 1;
    tr[u].l = build(l, mid, k);
    tr[u].r = build(mid + 1, r, k);
    pushup(u);
    return u;
}

void modify(int l, int r, int u, int v, int L, int R) {
    int mid = (L + R) >> 1;
    if (l <= mid) {
        if (L >= l && mid <= r) swap(tr[u].l, tr[v].l);
        else modify(l, r, tr[u].l, tr[v].l, L, mid);
    }
    if (r > mid) {
        if (mid + 1 >= l && R <= r) swap(tr[u].r, tr[v].r);
        else modify(l, r, tr[u].r, tr[v].r, mid + 1, R);
    }
    pushup(u);
    pushup(v);
}

int query(int u, int l, int r, int L, int R) {
    if (L >= l && R <= r) return tr[u].sum;
    int mid = (L + R) >> 1;
    if (r <= mid) return query(tr[u].l, l, r, L, mid);
    if (l > mid) return query(tr[u].r, l, r, mid + 1, R);
    return query(tr[u].l, l, r, L, mid) + query(tr[u].r, l, r, mid + 1, R);
}
```

```
void solve() {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= m; i++) root[i] = build(1, n, i);
    int q;
    cin >> q;
    while (q--) {
        int op;
        cin >> op;
        if (op == 1) {
            int l, r, x, y;
            cin >> l >> r >> x >> y;
            if (l == 1 && r == n) swap(root[x], root[y]);
            else modify(l, r, root[x], root[y], 1, n);
        } else {
            int l, r, k;
            cin >> l >> r >> k;
            cout << query(root[k], l, r, 1, n) << endl;
        }
    }
}

int main() {
    int t = 1;
    while (t--) solve();
}
```

Problem F. 公平分配

知识点：构造

正常展示我们要计算的目标分配 $X = (X_1, \ldots, X_n)$ 用数学语言直接表示的话是

$$\forall i, j \in N, e \in X_j, v_i(X_i) \geq v_i(X_j \setminus \{e\}).$$

首先可以考虑这个式子为什么长得这么奇怪，分析出题人的心路历程。不妨考虑计算：

$$\forall i, j \in N, e \in X_j, v_i(X_i) \geq v_i(X_j).$$

实际上，这个问题存在无解情况：考虑两个人，一个物品，无论物品分给谁，另一个都满足不了上述的式子。因此，我们需要简单地弱化一下原问题，于是允许在对方的集合中去掉一个物品。

下面介绍一种算法计算满足该题目的分配。 **不妨想象 n 个人现在轮流去拿物品，每个人拿剩下物品里它觉得价值最大的，那么这个分配就满足题目的要求**。我们可以用优先队列和 set 等数据结构在 $O(nm\log m)$ 的时间复杂度下做到该模拟。以下是粗略地证明：定义第 i 个人拿自己第 j 个物品的时间戳 t_{ij}，现在我们先只考虑第一个物品，只有 $t_{k1} < t_{i1}$，i 才有可能觉得 k 拿的物品比他大。由于我们可以去掉对方的一个物品，所以我们不妨去掉对方第一次拿的物品。那么现在考虑 i 手头物品和 k 物品的时间戳对比，一定有 $t_{i1} < t_{k2}, t_{i2} < t_{k3}, \dots$ 这意味着 $v_i(e_i^1) \geq v_i(e_k^2), v_i(e_i^2) \geq v_i(e_k^3), \dots$，即 i 第一次拿的不会比 k 第二次拿的更差，即 i 第二次拿的不会比 j 第三次拿的更差, ... 综上，这样模拟得到的分配满足

$$\forall i, j \in N, e \in X_j, v_i(X_i) \geq v_i(X_j \setminus \{e\}).$$

Solution

```cpp
#include <cstdint>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

void solve() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int n, m;
    cin >> n >> m;
    vector<vector<int>> v(n, vector(m, 0));

    vector<int> state(m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) cin >> v[i][j];

    vector<priority_queue<pair<int, int>>> q(n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) q[i].emplace(v[i][j], j);

    vector<vector<int>> ans(n);
    int                 T = m, idx = 0;
    while (T--) {
        while (state[q[idx].top().second]) q[idx].pop();
        auto t = q[idx].top();
```

```
            q[idx].pop();
            state[t.second] = 1;
            ans[idx].push_back(t.second);
            idx = (idx + 1) % n;
        }

        for (int i = 0; i < n; i++) {
            cout << ans[i].size() << ' ';
            for (auto t : ans[i]) cout << t + 1 << ' ';
            cout << "\n";
        }
}

int main() {
    int t;
    t = 1;
    while (t--) solve();
}
```

Problem G. 史蒂夫玩跑酷

知识点：图论，最短路

建图，每个位置向可以跳跃到的位置连边，可以用（需放置方块，跳跃数）表示距离。从一个位置跳到另一个位置时，若目标位置没有方块，则需放置方块加 1，无论是否有方块，跳跃数都加 1，跑 Dijkstra 算法即可。时间复杂度为 $O(nmk^2 \log(nm))$

Solution

```
#include <cstdint>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

constexpr int maxn = 1e9 + 7;
int           mx[10];

struct node {
```

```
    int  x, y;
    int  block, step;
    bool operator<(const node& a) const { return block > a.block || (block == a.block
};

struct dis {
    int          b = maxn, s = maxn;
    friend bool le(dis a, dis b) {
        if (a.b != b.b) {
            return a.b < b.b;
        }
        return a.s < b.s;
    }
};
int main() {
    i64 n, m, k;
    int tx, ty;
    cin >> n >> m >> k;
    vector<string> vt(n);
    for (auto& x : vt) cin >> x;

    mx[0] = k;
    for (int i = 0; i <= k; i++) {
        for (int j = 0; j <= k; j++) {
            if (i * i + j * j > k * k) {
                mx[i] = j - 1;
                break;
            }
        }
    }
    vector<vector<bool>> vis(n, vector(m, false));
    vector<vector<dis>>  dist(n, vector(m, dis()));

    priority_queue<node, vector<node>> q;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (vt[i][j] == 's') {
                q.push({i, j, 0, 0});
                dist[i][j] = {0, 0};
                vt[i][j]   = '1';
            } else if (vt[i][j] == 't') {
```

```
                tx = i, ty = j;
                vt[i][j] = '1';
            }
        }
    }
    while (!q.empty()) {
        i64 nx = q.top().x, ny = q.top().y;
        q.pop();
        if (vis[nx][ny]) {
            continue;
        }
        vis[nx][ny] = true;
        for (auto i = max(0ll, nx - k); i <= min(n - 1, nx + k); i++) {
            int d = mx[abs(i - nx)];
            for (auto j = max(0ll, ny - d); j <= min(m - 1, ny + d); j++) {
                if (i == nx && j == ny || vis[i][j]) {
                    continue;
                }
                int add = 1 - (vt[i][j] - '0');
                dis u   = {dist[nx][ny].b + add, dist[nx][ny].s + 1};
                if (le(u, dist[i][j])) {
                    dist[i][j] = u;
                    q.emplace(i, j, dist[nx][ny].b + add, dist[nx][ny].s + 1);
                }
            }
        }
    }
    cout << dist[tx][ty].b << " " << dist[tx][ty].s << "\n";
}
```

Problem H. 史蒂夫的粘液块

知识点：数学，二分

注意到

1. $h_{cur} - \lfloor \frac{a \times h_{cur}}{b} \rfloor$是单调不减函数，即当$h$减小时，弹起后减少的高度不会增加
2. 最多只有 $\sqrt{h}$ 种不同的减少高度

因此只要每次通过二分查找找出最小的 h_0 使得 $h_{cur} - \lfloor \frac{a \times h_{cur}}{b} \rfloor = h_0 - \lfloor \frac{a \times h_0}{b} \rfloor$，以此在 $O(logh)$ 的时间复杂度下计算出每种减少高度会进行多少次，重复 $O(\sqrt{h})$ 次直到 h_{cur} 为 0，时间复杂度为 $O(t \times \sqrt{h} \times logh)$

Solution

```
#include <iostream>
#include <vector>
using namespace std;
typedef long long ll;
ll sub(ll n, ll a, ll b)
{
    return n - n * a / b;
}
int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        ll n, a, b;
        cin >> n >> a >> b;
        ll cnt = 0;
        while (n)
        {
            ll l = 0, r = n;
            ll s = sub(n, a, b);
            while (l + 1 < r)
            {
                ll mid = (l + r) / 2;
                ll s1 = sub(mid, a, b);
                if (s1 == s)
                {
                    r = mid;
                }
                else
                {
                    l = mid;
                }
            }
            ll d = (n - r) / s + 1;
            n -= d * s;
            cnt += d;
        }
        cout << cnt - 1 << "\n";
    }
```

}

Problem I. 幸运数字

知识点：数论，因数分解

考虑到对于某一数字，在询问中可能被多次判断，故思路为预处理数字是否为"幸运数字"，结合前缀和。

一个数字为"幸运数字"，应该满足以下条件：

1. 所有真因数中，是否存在一个其数位出现 1 至 9 的次数与该数一样；
2. 出现 0 的次数要小于等于该数（因为这个数字转换后会去除前导零，0 的次数只会减少或不变）；

考虑一般做法，枚举每个数字，每个数字再枚举所有因数，总时间复杂度为 $\sum_{x=1}^{n}\sqrt{x}\approx O(n^{3/2})$，不足以通过此题。

发现枚举因数的过程中，会有因数被重复枚举。我们可以转换顺序，先枚举因数，再枚举每个倍数。这样的时间复杂度为 $\sum_{x=1}^{n}\frac{n}{x}\approx O(n\ln n)$。

Solution

```
#include <cstdint>
#include <iostream>
#include <vector>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

int main() {

    int          A = 1000000;
    vector<int> pre(A + 5, 0);
    vector<int> cnt(10, 0), temp(10, 0);

    for (int i = 1; i <= A; i++) {
        if (pre[i]) continue;

        // count of 0-9
        for (int o = 0; o < 10; o++) cnt[o] = 0;
```

```
        for (int o = i; o; o /= 10) cnt[o % 10]++;

        // factor
        for (int j = 2 * i; j <= A; j += i) {
            if (pre[j]) continue;

            // count of 0-9 on j
            for (int o = 0; o < 10; o++) temp[o] = 0;
            for (int o = j; o; o /= 10) temp[o % 10]++;

            bool ok = true;
            for (int l = 1; l < 10; l++) {
                if (temp[l] != cnt[l]) ok = false;
            }

            if (ok && temp[0] >= cnt[0]) pre[j] = 1;
        }
    }

    for (int i = 1; i <= A; i++) pre[i] += pre[i - 1];

    int T, l, r;
    cin >> T;

    while (T--) {
        cin >> l >> r;
        cout << pre[r] - pre[l - 1] << '\n';
    }

    return 0;
}
```

Problem J. 安达与岛村(1)

知识点：模拟

观察规则：

1. 点运动的时候可以自由穿过球台和障碍点（可以从第三象限向上穿过球台），在规则中提及，这也算是"触及"，但是如果"触及"发生的位置是原点，那就不能计入了。
2. 端点问题：球台和球网端点都是要计入计算的。例如，球台长为 10，球网高为 5，那么 $(-5,0)$，

$(5,0)$，$(0,5)$，$(0,0)$ 这三个点都算作与对应物体发生了触及。特殊的，尽管 $(0,0)$ 是球台与球网的重合点，规则优先级限制在这个点发生的触及不计入最终答案。

明确这两点后，思路便很明确了：

由于点是从"左边"向"右边"移动的，x 值小的点必然更早与运动点发生"触及"事件。可以使用堆对点坐标进行维护。

具体解法：维护运动点运动方向 (x,y) 和运动点当前位置 (X,Y)，当前所有点集合 set。

首先，将所有点加入 set。

然后维护堆：只要 set 非空，执行操作：将点 (X,Y) 和向量 (x,y) 引出的射线与 x 轴的交点加入到集合，取出当前 set 中的首元素，判断是否可以到达，如果 Y 为 0，符合所有条件后，可以考虑增加触及次数。将该点删除，然后对 (x,y) 和 (X,Y) 进行更新。此过程需要注意结合规则进行特判。

由于向加入点的操作只能在两个点之间发生，加入的点的数量一定与障碍点的数量相当，即集合最多有 $O(n)$ 个点，n 为障碍点的数量。最终复杂度为 $O(n\log n)$，可以通过本题。

Solution

```cpp
#include <cmath>
#include <cstdint>
#include <iostream>
#include <set>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

const double eps = 1e-8;

int sgn(double x) {
    if (fabs(x) < eps) return 0;
    if (x < 0) return -1;
    else return 1;
}

int dcmp(double x, double y) {
    if (fabs(x - y) < eps) return 0;
    if (x > y) return 1;
    return -1;
}
```

```
inline double sqr(double x) { return x * x; }
struct Point {
    double x{}, y{};
    Point() = default;
    Point(double _x, double _y) {
        x = _x;
        y = _y;
    }
    void   input() { cin >> x >> y; }
    bool   operator==(Point b) const { return sgn(x - b.x) == 0 && sgn(y - b.y) == 0
    Point  operator-(const Point& b) const { return Point(x - b.x, y - b.y); }
    double operator^(const Point& b) const { return x * b.y - y * b.x; }
    double distance(Point p) { return hypot(x - p.x, y - p.y); }
    Point  operator+(const Point& b) const { return Point(x + b.x, y + b.y); }
};

struct Line {
    Point s, e;

    Line() = default;
    Line(Point _s, Point _e) {
        s = _s;
        e = _e;
    }

    void input() {
        s.input();
        e.input();
    }

    double length() { return s.distance(e); }
    int    relation(Point p) {
        int c = sgn((p - s) ^ (e - s));
        if (c < 0) return 1;
        else if (c > 0) return 2;
        else return 3;
    }

    Point crosspoint(Line v) {
        double a1 = (v.e - v.s) ^ (s - v.s);
        double a2 = (v.e - v.s) ^ (e - v.s);
```

```
        return Point((s.x * a2 - e.x * a1) / (a2 - a1), (s.y * a2 - e.y * a1) / (a2 -
    }
    int linecrossseg(Line v) {
        int d1 = sgn((e - s) ^ (v.s - s));
        int d2 = sgn((e - s) ^ (v.e - s));
        if ((d1 ^ d2) == -2) return 2;
        return (d1 == 0 || d2 == 0);
    }
};

struct cmp {
    bool operator()(const pair<Point, int>& a, const pair<Point, int>& b) const {
        if (sgn(a.first.x - b.first.x) != 0) return a.first.x < b.first.x;
        else if (sgn(a.first.y - b.first.y) != 0) return a.first.y < b.first.y;
        return a.second > b.second;
    }
};

void solve() {
    set<pair<Point, int>, cmp> p;
    double                     H, W;
    Point                      ball, v;
    int                        n;
    cin >> H >> W;
    ball.input();
    v.input();
    cin >> n;
    for (int i = 0; i < n; i++) {
        Point o;
        o.input();
        p.insert({o, 0});
    }
    p.insert({ball, 0});

    Line l;
    Line table(Point(-H / 2, 0.0), Point(H / 2, 0.0));
    Line web(Point(0.0, 0.0), Point(0.0, W));
    auto getpoint = [&]() {
        l = Line(ball, ball + v);
        if (l.linecrossseg(table) != 0) {
            Point tablep = l.crosspoint(table);
```

```
            if (sgn(tablep.x) != 0 || sgn(tablep.y) != 0) {
                p.emplace(tablep, 1);
            }
        }
        if (l.linecrossseg(web) != 0) {
            p.emplace(l.crosspoint(web), 2);
        }
    };


    bool ok  = false;
    int  ans = 0;
    for (auto& k : p) {
        if (k.first == ball) {
            getpoint();
            ok = true;
        } else {
            if (!ok) continue;
            if (l.relation(k.first) == 3) {
                if (k.second == 0) v.y = -fabs(v.y);
                if (k.second == 1) {
                    v.y = fabs(v.y);
                    ans++;
                }
                if (k.second == 2) break;
                ball = k.first;
                getpoint();
            }
        }
    }
    cout << ans << '\n';
}

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

Problem K. 安达与岛村(2)

知识点：费用流，二分图

事实上，本题考察的初衷并非为 gcd 的性质，而是对图论的考察。但是引入 gcd 后可能会导致有不同的解法。以下是两种图论解法：

解法一：流建模

采用最大费用最大流建模（实际上对费用进行取反，与最小费用最大流无异），此处的费用等价于牌的数量，在满流的情况下，若总费用的大小与牌的总数相当，即可换完。

点集合：

1. 将牌 $\{1,2,3,\ldots,n\}$ 视为 n 个节点，将每个节点拆成两个节点：$a = \{1,2,3,\ldots,n\}$ 和 $b = \{1+n,2+n,\ldots,n+n\}$。
2. 建立超级源点 S 和汇点 T。
3. 建立超级源点限制点 s。

边集合（以下操作默认建立反向边）：

1. 对于超级源点限制点 S，执行操作：S 向 s 连接一条容量为 m 的，费用为 0 的有向边，代表安达的起始手牌数量最多为 m。
2. 对于超级源点限制点 s，执行操作：向 a 中所有的元素 x 连接一条容量大于等于 A_x，费用为 0 的有向边，代表对相应类型的牌的限制。
3. 对于集合 a 中的每个元素 x，执行操作：x 向 $x+n$ 连接一条容量为 A_x，费用为 -1 的有向边，代表类型为 x 的牌一共只有 A_x 张牌。
4. 对于集合 b 中的每个元素 $x+n$，执行操作：$x+n$ 向汇点连接一条容量大于等于 A_x，费用为 0 的有向边，代表对相应类型的牌的限制。同时，假设 i 为集合 a 中不为 x 的元素，若 $\gcd(x,i)$ 不等于 1，则 $x+n$ 向 i 连接一条容量大于等于 $\min(A_x,A_i)$，费用为 0 的有向边，代表 x 类型牌换成 i 类型牌的限制。

对该图模型跑一次最小费用最大流即可。注意此处的费用需要取相反数，最终最大费用记为 sum。若 sum 等于 $A_1+A_2+\cdots+A_n$，则岛村可以换完所有牌。当然上述建模方法存在冗余，并不是最简单建模方法，但是比较直观。时间复杂度为网络流时间复杂度，但是由于图模型复杂，可能会超时，需要对最大费用最小流过程进行优化才能通过。

解法二：二分图+最大权匹配（使用最大流）

对换牌机制进行分析：将 a 类型牌换成 b 类型牌表示为 $a \to b$，那么观察易得 $a \to b \to c \to d \to \cdots \to z$ 实际上也可以表示为 25 步过程：$a \to b, b \to c, c \to d, \ldots, y \to z$。也就是说，换牌机制实际

上是将原本需要多步骤的过程缩减为 1 步，假设元素集合 $\{obj_1, obj_2, \ldots, obj_n\}$ 中的元素存在关系：$obj_1 \to obj_2, obj_2 \to obj_3, \ldots, obj_{n-1} \to obj_n$，则可以缩减为 $obj_1 \to obj_2 \to \cdots \to obj_n$。反过来说，如果 n 对关系 $a \to b$ 可以通过彼此之间的关系联合起来，那么就可以用 1 步关系完成转换，即节省 $n-1$ 步。对应到二分图上，就是匹配关系（两两配对的关系）。

用一个简单的例子来解释：假设此时牌的可重集合为 $\{2,2,4,4,8\}$，那么最优的换牌链应该为 $8 \to 4 \to 2, 4 \to 2$，即两张初始手牌即可完成。上述过程同时也可以拆解为 $8 \to 4, 4 \to 2, 4 \to 2$ 三条匹配关系，观察到：点总数 = 最大匹配关系数 + 最优换牌链数，即 $5 = 3 + 2$。换句话说，由于最优换牌链的数量非常难以通过直接求解得到，我们可以转换成求解匹配关系对，间接得到最优换牌链的数量。当然，由于本题的牌类型和换牌关系具有特殊的性质，求解最大匹配关系数可能会有更加优秀的解法。以下是图论解法：

点建立：

1. 将牌 $\{1,2,3,\ldots,n\}$ 视为 n 个节点，将每个节点拆成两个节点：$a = \{1,2,3,\ldots,n\}$ 和 $b = \{1+n, 2+n, \ldots, n+n\}$。$a$ 集合作为二分图左部分，b 集合作为二分图右部分。
2. 建立超级源点 S 和汇点 T。

边集合：

1. 对于超级源点 S，执行操作：S 向集合 a 中的每个元素 x，连接一条容量为 A_x 的有向边，代表 x 类型的牌只有 A_x 张。
2. 对于集合 b 的每个元素 x，执行操作：x 向汇点 T 连接一条容量为 A_x 的有向边，代表以 x 类型牌作为最后一张牌的换牌链最多只有 A_x 条。
3. 对于集合 a 的每个元素 x，执行操作：假设 i 为集合 a 中不为 x 元素，若 $\gcd(x,i)$ 不等于 1，则 x 向 $i+n$ 连接一条权为 $A_1 + A_2 + \cdots + A_n$ 的有向边，代表 x 类型牌换成 i 类型牌的限制。

对该二分图跑一次最大权匹配，将最大权匹配记作 sum，若牌的总数大于最大权匹配与最大初始手牌数的和，则岛村无法完成游戏，反之则可以完成。

由于二分图在网络流模型的执行中具有优良的性质，该解法会比解法一快非常多，非常容易通过本题。

Solution

```
#include <cmath>
#include <cstdint>
#include <iostream>
#include <numeric>
#include <queue>
#include <vector>
```

```
using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

const int N = 500 + 5, M = 1e9 + 7;

template <typename T>
struct Flow_ {

    const int n;
    const T   inf = numeric_limits<T>::max();

    struct Edge {
        int to;
        T   w;
        Edge(int to, T w) : to(to), w(w) {}
    };

    vector<Edge>          ver;
    vector<vector<int>> h;
    vector<int>           cur, d;

    explicit Flow_(int n) : n(n + 1), h(n + 1) {}

    void add(int u, int v, T c) {
        h[u].push_back(ver.size());
        ver.emplace_back(v, c);
        h[v].push_back(ver.size());
        ver.emplace_back(u, 0);
    }

    bool bfs(int s, int t) {
        d.assign(n, -1);
        d[s] = 0;
        queue<int> q;
        q.push(s);

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

```
            for (auto it : h[x]) {
                auto [y, w] = ver[it];

                if (w && d[y] == -1) {
                    d[y] = d[x] + 1;

                    if (y == t) return true;

                    q.push(y);
                }
            }
        }
        return false;
    }

    T dfs(int u, int t, T f) {
        if (u == t) return f;

        auto r = f;

        for (int& i = cur[u]; i < h[u].size(); i++) {
            auto j         = h[u][i];
            auto& [v, c]   = ver[j];
            auto& [u, rc] = ver[j ^ 1];

            if (c && d[v] == d[u] + 1) {
                auto a  = dfs(v, t, std::min(r, c));
                c       -= a;
                rc      += a;
                r       -= a;

                if (!r) return f;
            }
        }

        return f - r;
    }

    T work(int s, int t) {
        T ans = 0;
        while (bfs(s, t)) {
```

```
            cur.assign(n, 0);
            ans += dfs(s, t, inf);
        }
        return ans;
    }
};
using Flow = Flow_<int>;

int  a[N];
void solve() {
    int n, m;
    cin >> n >> m;
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        sum += a[i];
    }

    Flow flow(2 * n + 5);
    int  S = 2 * n + 1, T = 2 * n + 2;

    for (int i = 1; i <= n; i++) {
        if (!a[i]) continue;

        flow.add(S, i, a[i]);
        flow.add(n + i, T, a[i]);
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j < i; j++) {
            if (gcd(i, j) != 1 && a[j]) {
                flow.add(i, n + j, sum);
            }
        }
    }

    sum -= flow.work(S, T);
    cout << (sum <= m ? "Adachi" : "Shimamura");
}
```

```
int main() {
    int t = 1;
    while (t--) solve();
}
```

Problem L. 寻找GZHU

知识点：字符串处理

逐行读入，并对偶数行翻转处理，拼接并统计出现次数。

Solution

```
#include <algorithm>
#include <cstdint>
#include <iostream>
#include <string>

using namespace std;
using i64 = int64_t;
using u64 = uint64_t;

int main() {
    int n, m;
    cin >> n >> m;

    string s;
    s.reserve(n * m);
    for (int i = 0; i < n; ++i) {
        std::string row;
        cin >> row;
        if (i % 2 == 1) std::reverse(row.begin(), row.end());
        s += row;
    }

    string target = "GZHU";
    int64_t ans = 0;
    int ind = 0;
    for (char i : s) {
        if (i == target[ind]) {
            ind++;
            if (ind == 4) {
```

```
                ans++;
                ind = 0;
            }
        } else {
            ind = 0;
        }
    }
    cout << ans << endl;

}
```