T1 产品序列

10%的做法：

对于 q=1，直接选取两个时间最小的机器

额外20%的做法：

首先考虑只有机器A的情况，可以想到，完成的越快的机器应该被优先使用，并且每个机器可以多次来使用。这个过程可以使用优先队列来维护。

再考虑机器A+机器B的影响，可以采用枚举的方式，对每个产品分配到哪个机器B上进行加工，取出其中最优的方案即可。

时间复杂度：$O(n * m!)$

60% - 100%的做法：

由于第一道工序和第二道工序除了完成顺序以外，没有其他的限制。也就是说，可以分解问题，对两个工序可以分开进行考虑。

考虑将第一道工序和第二道工序分开进行考虑，对两个加工过程分别使用贪心得到对应完成的时间 $f[i]$ 和 $g[i]$ ：完成的越快的机器应该被优先使用，并且每个机器可以多次来使用。使用优先队列维护。

再考虑将两个工序拼接起来，对于 B 机器来说，输入的产品相当于多了一个在 A 中提前处理的时间 f[i]，在 B 中需要处理的时间是 g[i]，也就是总共的时间是花费了 f[i]+g[i]。而对产品使用不同的机器来处理，相当于 f 和 g 分别以一定的顺序拼接起来，最后需要使得 max(f[i]+g[i]) 最小。

而怎么拼接会使得最大值最小呢？很经典的做法就是 f 的升序和 g 的逆序拼接起来。

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
#define N (int)(1e6+5)
struct node {
    int x, id;
    node() { }
    node(int xx, int idd) {
        x = xx;
        id = idd;
    }
    bool operator < (const node &rhs) const {
        return x > rhs.x;
    }
};
priority_queue<node>q;
int f[N], g[N], n, m, l, w[N], d[N];
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> l >> n >> m;
    for (int i = 1; i <= n; i++) cin >> w[i];
    for (int i = 1; i <= m; i++) cin >> d[i];
    for (int i = 1; i <= n; i++) q.push(node(w[i], i));
    for (int i = 1; i <= l; i++) {
        auto x = q.top();
        q.pop();
        f[i] = x.x;
        q.push(node(x.x + w[x.id], x.id));
    }
    while (!q.empty()) q.pop();
    for (int i = 1; i <= m; i++) q.push(node(d[i], i));
    for (int i = 1; i <= l; i++) {
        auto x = q.top();
        q.pop();
        g[i] = x.x;
        q.push(node(x.x + d[x.id], x.id));
    }
    int ans = 0;
    for (int i = 1; i <= l; i++) ans = max(ans, f[i] + g[l - i + 1]);
    cout << ans;
    return 0;
}
```

另外的100%的做法：

对于分开考虑的两道工序，我们要让所有产品完成的最大值尽量的小，对于最大值最小的问题，可以采用二分的方式来得到。

对最后完成一个产品完成的时间进行二分答案，判断能否在规定的时间内做完 q 个产品。最后可以得到类似上一种做法的 f 序列和 g 序列。将 f 和 g 拼接在一起就可以得到答案了。

```cpp
#include <cstdio>
#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>
const int N = 1e5, T = 1e6;
int t, n, m;
int a[N], b[N];
long long c[T], d[T];
void work(int a[], int n, long long c[]) {
    int cnt = 0;
    long long l = 0, r = 1e15 / n;
    while (l < r) {
        long long m = (l + r) / 2;
        long long c = 0;
        for (int i = 0; i < n; ++i)
            c += m / a[i];
        if (c >= t) {
            r = m;
        } else {
            l = m + 1;
        }
    }
    for (int i = 0; i < n; ++i)
        for (long long j = a[i]; j < l; j += a[i])
            c[cnt++] = j;
    std::sort(c, c + cnt);
    while (cnt < t)
        c[cnt++] = l;
}
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(NULL);
    std::cin >> t >> n >> m;
    for (int i = 0; i < n; ++i)
        std::cin >> a[i];
    for (int i = 0; i < m; ++i)
        std::cin >> b[i];
    work(a, n, c);
    work(b, m, d);
    std::reverse(d, d + t);
    long long ans = 0;
    for (int i = 0; i < t; ++i)
        ans = std::max(ans, c[i] + d[i]);
    std::cout << ans << "\n";
    return 0;
}
```

T2 乐高积木

部分分

对于部分分，比如 $k \leq 3$ 时，我们可以直接手动枚举出对于每一个 k 总共有多少种可能性，如果发现 k = 1 和 k = 2 都只有 1 种可能的形状，k = 3 总共有 2 种形状，那么对于每一种形状，我们去判断其是否可以放入 $n \times m$ 的矩形中即可。

这里，如果有耐心，甚至可以枚举出 $k \leq 5$ 的情况。

对于 $k \geq n \times m$ 的数据，显然答案不是 0 就是 1，当 $k > n \times m$ 时，答案为 0，当 $k = n \times m$ 时，全部放入就是一种答案。

这样理论上可以拿到 $30+$ 分。

满分

本题的关键点在于判断每一个连通块是否重复以及如何枚举所有的连通块。

设将有 n 个小方块的连通块叫做 $n-$ 连通块。

因为要判断每一个 $n-$ 连通块 是否重复，那么我们需要将这些图形一一对应起来，我们可以想象这些小方块放入一个平面直角坐标系里面，然后对于每一个图形通过平移、旋转、翻转一一列举出来，并利用 set 来保证不重复计算。

我们先将每一个小方块（单元格）定义为结构体 node，由于不考虑格子的位置，故又可以将每一个 $n-$ 连通块 定义为 node 的集合，表示每一个 $n-$ 连通块 由一系列的单元格组成。

1. 对于平移操作

- 我们可以将 $n-$ 连通块 全部平移到坐标原点 $(0,0)$。具体操作：
  - 我们找出 $n-$ 连通块 中 x 和 y 坐标的最小值 minx、miny，那么它可以视为一个平移 $(minx, miny)$ 矢量，将 $n-$ 连通块 中的每一个单元格斗减去该矢量就可以移动到坐标原点了。
  - 我们将这一步操作定义成一个标准化函数 `normalize`

2. 对于旋转操作

- 我们可以定义一个 `rotate` 函数，表示将整个连通块围绕坐标原点顺时针旋转 90 度，实现只需要将每个格子都顺时针旋转 90 度即可。相应的几何变换为 $(x, y) \to (y, -x)$。

3. 对于翻转操作

- 由于既可以沿 x 轴翻转，也可以沿 y 轴翻转，但实际上沿 x 轴翻转后再绕坐标原点顺时针旋转 180 度即可得到沿 y 轴翻转的图案。因此这里我们定义一个 `flip` 函数，表示将一个连通块沿 x 轴翻转。相应的几何变换为 $(x, y) \to (x, -y)$。

那么，判断 $n-$ 连通块 的具体操为：

1. 首先将当前的连通块平移到坐标原点，每次都顺时针旋转 90 度，检查是否和当前的 $n-$ 连通块 集合中出现的有重复。如果均没有，将该连通块沿 x 轴翻转后，再依次顺时针旋转 90 度判断，如果均没有，就表示这是一种新的形态，加入到 $n-$ 连通块 所在的集合中即可。

2. 对于枚举 $n-$ 连通块，当 $n > 1$ 时，$n-$ 连通块 一定是在 $n-1-$ 连通块 的基础上生成的，即以每个 $n-1-$ 连通块 为基础，从某一个 $n-1-$ 连通块 的某个单元格开始，向上下左右 4 个方向扩展。如果可以扩展，且不出现重复，就找到了一个 $n-$ 连通块，加入到集合中来。最终完成n连通块的枚举。

如果对于每组数据都单独去搜索，因为 T 有 10^5，会超时。

因为 $k <= 10$，所以最多就 1000 种情况，设 $ans[i][j][k]$ 表示小方块为 i 个，底座为 $j \times k$ 情况下的答案，那么枚举每一种情况，预处理出答案， 这样对于每组数据都可以 $O(1)$ 回答了。

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

struct node {
    int x, y;
    bool operator < (const node & rhs) const {
        return x < rhs.x || (x == rhs.x && y < rhs.y);
    }
};
typedef set<node> lego; // lego 是一个单元格的集合
int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};
int ans[15][15][15];
set<lego> a[15]; // a 数组里面的每一个元素都是一个乐高集合 (i-连通块 集合)

// 找到乐高图中最小的 (x,y) 然后将整个图平移到坐标原点
inline lego normalize(const lego & p) {
    int minx = p.begin() -> x, miny = p.begin() -> y;
    for(auto item: p) {
        minx = min(minx, item.x);
        miny = min(miny, item.y);
    }
    lego p2;
    for(auto item: p) { // 将乐高p中的每个格子平移到坐标原点
        node tmp;
        tmp.x = item.x - minx, tmp.y = item.y - miny;
        p2.insert(tmp);
    }
    return p2;
}

// 将乐高图形顺时针旋转90度, 同时标准化(平移到坐标原点)
inline lego rotate(const lego & p) {
    lego p2;
    for(auto item: p) { // 将乐高p中每一个格子顺时针旋转 90 度
        node tmp;
        tmp.x = item.y, tmp.y = -item.x; // (x, y) -> (y, -x)
        p2.insert(tmp);
    }
    return normalize(p2);
}

// 将乐高图形按照x轴翻转(上下做镜面对称变换), 同时标准化(平移到坐标原点)
inline lego flip(const lego & p) {
```

```
    lego p2;
    for(auto item: p) {
        node tmp;
        tmp.x = item.x, tmp.y = -item.y; // (x, y) -> (x, -y)
        p2.insert(tmp);
    }
    return normalize(p2);
}

// 在乐高图形p中加入一个单元格c，并判断形成的连通块是不是新的连通块
// 如果是就加入乐高集合
void check(const lego & p0, const node & c) {
    lego p = p0;
    p.insert(c); // 放入一个新的单元格 c
    p = normalize(p); // 标准化处理(平移到坐标原点)
    int len = p.size(); // 乐高图形p有len个单元格，即 len-连通块
    for(int i = 0; i < 4; ++i) {
        if(a[len].find(p) != a[len].end()) return; // 集合中能找到，重复了
        p = rotate(p); // 旋转90度
    }
    p = flip(p); // 沿着x轴翻转（水平翻转）
    for(int i = 0; i < 4; ++i) { // 再旋转
        if(a[len].find(p) != a[len].end()) return;
        p = rotate(p);
    }
    a[len].insert(p); // 找到了一个不重复的 len-连通块保存
}

void cal() {
    lego s;
    s.insert((node){0, 0}); // 先放一个单元格
    a[1].insert(s); // 1连通块
    for(int i = 2; i <= 10; ++i) {
        for(auto p: a[i - 1]) { // 枚举每一个 i-1 连通块
            for(auto item: p) { // 枚举 i-1 连通块中的每一个单元格
                for(int j = 0; j < 4; ++j) {
                    int nx = item.x + dx[j], ny = item.y + dy[j];
                    node tmp = {nx, ny};
                    // i-1连通块中找不到 tmp，即当前扩展的格子不与i-1连通块任意一个格子重合
                    // 那么，至少在现在看来可能是有效的，但是还要进一步检验加入tmp形成的 n-连通块 是否重复
                    if(p.find(tmp) == p.end()) {
                        check(p, tmp);
                    }
                }
            }
        }
    }
    for(int k = 1; k <= 10; ++k) { // 枚举每一种情况，提前计算出答案
```

```cpp
        for(int n = 1; n <= k; ++n) {
            for(int m = 1; m <= k; ++m) {
                int cnt = 0;
                for(auto const & p: a[k]) { // 枚举每一个 k-连通块 p
                    int maxx = 0, maxy = 0;
                    for(auto item: p) { // 枚举 k-连通块 中的每一个单元格
                        maxx = max(maxx, item.x);
                        maxy = max(maxy, item.y);
                    }
                    // 能够放入 n * m 的大矩形中
                    if(min(maxx, maxy) <= min(n - 1, m - 1) && max(maxx, maxy) <= max(n - 1, m - 1)) {
                        ++cnt;
                    }
                }
                ans[k][n][m] = cnt;
            }
        }
    }
}

int main() {
    cal(); // 提前预处理出所有情况
    int t, k, n, m;
    cin>>t;
    while (t--) {
        cin>>k>>n>>m;
        cout<<ans[k][n][m]<<'\n';
    }
    return 0;
}
```

T3 XB的生日

20%的做法：

对于 $n \leq 5, T \leq 10$ 的部分，可以直接搜索，找出所有合法方案。

额外20%的做法（T<=500）：

对于 $T \leq 500$ 的部分，可以直接进行 dp ，按照时间段进行划分，并记录当前已有的原料。

设 $f[i][j][k]$ 代表到时间 i 截止，到了 j 号点，拥有 k 的原料的方案数（k为二进制表示）。

按照时间进行枚举，对于每个点 u ，考虑所有由他出发的边 (u,v,w)：

- 如果经过商店 $f[i][v][k|w]$ += $f[i-2][u][k]$ 。
- 如果不经过商店 $f[i][v][k] += f[i-1][u][k]$。

时间复杂度 $O(16nmT)$

额外20%的做法（n<=5）：

对于 $n \leq 5$ 但 T 很大的部分，考虑到状态数较小的时候，如果能够固定下转移的方程，可以使用矩阵乘法来优化计算。

可以构造一个大矩阵，每个点有 16 种状态，表示达到当前点已经包含哪些字符。

由于进入商店需要花费 2 的时间，对每个状态可以再新建一个节点，经过这个节点代表从某个状态出发将会进入一个商店。也即某个状态经过一条边时去商店，先去新建的节点，再去边的终点（把长度为 2 的路径变成两条长度为 1 的）。如果不去就直接连向边的终点。

由于只需要在 T 时间内回到点 1，所有要在矩阵中记录一个前缀和。(具体的前缀和处理参考下面的方法)

这样矩阵的大小将会是 $c = 5 \times 16 \times 2 + 1 = 161$，时间复杂度将会是 $O(c^3 * log_2 T)$。

额外20%的做法（BJMP）：

对于每条边都包含"BMJP"的部分，可以考虑所有的方案减去完全不去商店的方案。

对于每个点 u，考虑所有由他出发的边 (u,v)，$f[i][j]$ 表示时间 i 的时候，在点 j 的方案数。

所有方案的计算：类似上一部分的分裂点构造矩阵（把长度为2的路径变成两条长度为1的路径），使用矩阵加速计算。矩阵大小为 $c = 25 \times 2 + 1 = 51$。

- 经过商店 $f[i][v]$ += $f[i-1][u+n]$。
- 不经过商店 $f[i][v] += f[i-1][u]$。

由于只需要在 T 时间内回到点 1，所有要在矩阵中记录一个前缀和。

其中对于 t 时刻的矩阵，$s_{i,1}$ 记录的是 1-t 时刻所有从 i 到 1 就结束的方案数（t 的前缀和）。

$$\begin{bmatrix} 0 & 0 & \dots & 0 \\ s_{1,1} & f[i][1] & \dots & f[i][2n] \\ 0 & 0 & \dots & 0 \\ \dots & & & \\ 0 & 0 & \dots & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & f[i-1][1] & \dots & f[i-1][2n] \\ 0 & 0 & \dots & 0 \\ \dots & & & \\ 0 & 0 & \dots & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ s_{1,1} & a_{1,1} & 0 & \dots & a_{1,2n} \\ s_{2,1} & a_{2,1} & 0 & \dots & a_{2,2n} \\ \dots & & & & \\ s_{2n,1} & a_{2n,1} & 0 & \dots & a_{2n,2n} \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & \dots & 0 & \\ s_{1,1} & f_{T,1} & f_{T,2} & \dots & f_{T,2n} \\ 0 & 0 & \dots & 0 & \\ \dots & & & & \\ 0 & 0 & \dots & 0 & \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 & \\ 0 & f[0][1] & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \\ \dots & & & & \\ 0 & 0 & \dots & 0 & \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ s_{1,1} & a_{1,1} & 0 & \dots & a_{1,2n} \\ s_{2,1} & a_{2,1} & 0 & \dots & a_{2,2n} \\ \dots & & & & \\ s_{2n,1} & a_{2n,1} & 0 & \dots & a_{2n,2n} \end{bmatrix}^T$$

完全不经过商店：所有边都当做长度为1的路径。

- 不经过商店 $f[i][v] += f[i-1][u]$。

矩阵大小为 $c = 25 + 1 = 26$。

100%的做法：

考虑到总共的字符种类数比较少，参考上一部分的做法，可以考虑容斥来解决，考虑只经过特定的字符的方案。

所有的方案数先减去有 1 个字符一定不走的方案，再加上 2 种字符一定不走的情况，再减去有 3 个字符一定不走的方案，再加上 4 种字符一定不走的情况。

使用上一个部分分裂点构造矩阵的方法，需要注意的是某种字符不经过的时候，所有相关的边都不考虑即可构造出相应的矩阵。矩阵大小为 $c = 25 \times 2 + 1 = 51$ 。

时间复杂度：$O(2^4 c^3 log_2 T)$

```
#include <bits/stdc++.h>
using namespace std;
const int M = 55, N = 505, P = 5557;
int n, m, T, EA[N], EB[N], EC[N], mp[N];
```

```
int A[M][M], B[M][M];

void mult(int A[M][M], int B[M][M], int C[M][M]) {
    int T[M][M];
    for (int i = 0; i <= 2 * n; i++)
        for (int j = 0; j <= 2 * n; j++) {
            T[i][j] = 0;
            for (int k = 0; k <= 2 * n; k++) T[i][j] += A[i][k] * B[k][j];
        }
    for (int i = 0; i <= 2 * n; i++)
        for (int j = 0; j <= 2 * n; j++)
            C[i][j] = T[i][j] % P;
}

int solve(int has) { // B * A ^ n
    memset(A, 0, sizeof(A));
    memset(B, 0, sizeof(B));
    A[0][0] = 1;
    for (int i = 1; i <= n; i++) A[i][i + n] = 1; // 添加经过商店的点 i+n
    for (int i = 1; i <= m; i++) {
        int a = EA[i], b = EB[i];
        A[a][b]++; // 不经过商店的
        if ((EC[i] | has) == has) A[a + n][b]++; //经过商店的 a -> a+n -> b
    }
    for (int i = 1; i <= 2 * n; i++) A[i][0] = A[i][1]; // 初始 s_i0 = a_i1
    B[1][1] = 1;
    for (int i = 0; (1 << i) <= T; i++) {
        if (T & (1 << i)) mult(B, A, B);
        mult(A, A, A);
    }
    return B[1][0] % P;
}
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n >> m;
    mp['B'] = 1, mp['J'] = 2, mp['M'] = 4, mp['P'] = 8;
    for (int i = 1; i <= m; i++) {
        char str[9];
        cin >> EA[i] >> EB[i] >> str;
        for (int j = 0; j < strlen(str); j++) EC[i] |= mp[str[j]];
    }
    cin >> T;
    int ans = 0;
    for (int i = 0; i < 16; i++) {
        int f = 1;
        for (int j = 0; j < 4; j++) if ((i >> j) & 1) f = -f;
        ans = (ans + solve(i) * f + P) % P;
    }
    cout << ans << '\n';
    return 0;
}
```

T4 天分测试

性质：本质不同的起点和方向的选择只有 n * m * 8 个。

20%的做法：

同时枚举2个字符串的起点和方向，直接枚举K的长度，暴力判读是否一样。

时间复杂度：$O((8nm)^2 * k)$

40%的做法：

对每种起点和方向，记录对应的哈希值，记录有多少个哈希值与当前枚举起点和方向对应的字符串哈希值一样。

使用map来实现哈希的话，时间复杂度：$O(8nmk * log_2(nmk))$

70%的做法：

对于 n=m 的情况，由于所有方向的循环节都是n，这个时候只需要验证k=n的情况下相等，就可以得到所有k>n的情况了。

100%的做法：

直接计算哈希值，由于k的范围很大，只有 n*m 的范围是独特的，其他是由复制得到的，考虑使用倍增来加速求哈希的过程。

但是需要注意哈希的模数的选取，在模数取 998244353 或 $10^9 + 7$ 时都会被卡掉一个点。

可以选择使用：

1. 使用双哈希。
2. 使用一些不那么有名的模数，例如：998244853

时间复杂度：$O(8nm \log_2 k)$

```cpp
#include <bits/stdc++.h>
#define ll long long
using namespace std;
char grid[220][220];
const int dx[] = {0, 0, 1, 1, 1, -1, -1, -1};
const int dy[] = {1, -1, 1, 0, -1, 1, 0, -1};
const ll mod = 998244853;
const ll base = 47;
ll Hash[220][220][8][32];
ll Power[33];
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
```

```
    cout.tie(0);
    Power[0] = base;
    for (int i = 1; i < 33; i++) Power[i] = Power[i - 1] * Power[i - 1] % mod;
    int n, m;
    ll k;
    cin >> n >> m >> k;
    for (int i = 0; i < n; i++) cin >> grid[i];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            for (int d = 0; d < 8; d++) {
                Hash[i][j][d][0] = (grid[i][j] - 'a' + 1);
            }
    //预处理倍增哈希
    for (int l = 1; l < 30; l++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                for (int d = 0; d < 8; d++) {
                    int nx = ((i + dx[d] * (1 << (l - 1))) % n + n) % n;
                    int ny = ((j + dy[d] * (1 << (l - 1))) % m + m) % m;
                    Hash[i][j][d][l] = (Hash[i][j][d][l - 1] * Power[l - 1] +
Hash[nx][ny][d][l - 1]) % mod;
                }
    vector<ll> vec;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            for (int d = 0; d < 8; d++) {
                ll Hsh = 0;
                int x = i, y = j;
                for (int l = 29; l >= 0; l--)
                    if (k >> l & 1) {
                        int nx = ((x + dx[d] * (1 << (l))) % n + n) % n;
                        int ny = ((y + dy[d] * (1 << (l))) % m + m) % m;
                        Hsh = (Hsh * Power[l] + Hash[x][y][d][l]) % mod;
                        x = nx;
                        y = ny;
                    }
                vec.push_back(Hsh);
            }
    sort(vec.begin(), vec.end());
    int cnt = 0;
    ll sum = 0;
    for (int i = 0; i < (int)(vec.size()); i++) {
        if (i > 0 && vec[i] == vec[i - 1]) cnt++;
        else {
            sum += 1ll * cnt * cnt;
            cnt = 1;
        }
    }
    sum += 1ll * cnt * cnt;
    ll deno = 1ll * ((int)(vec.size())) * ((int)(vec.size()));
    ll g = __gcd(deno, sum);
    deno /= g;
    sum /= g;
    printf("%lld/%lld\n", sum, deno);
    return 0;
}
```

补充：

对于 n!=m 的情况，循环节为 lcm(n,m)，k的范围只需要考虑到 lcm(n,m) 的大小即可。

对于哈希的做法，可以使用unsigned long long进行自然溢出，如果你得了80或者90分，那么通常是哈希的底数或者模数找的不够大，可以尝试使用更好的模数或者双哈希。

附上另外100分做法：

```
#include <bits/stdc++.h>
#define LL long long
#define Z(a,b,c) a*n*m+b*m+c+1
//使用 Z 函数将 (n,m,d) 变为一维
using namespace std;
int n, m, k, g[25][320005], lg, x;
int dx[8] = {1, 1, 1, 0, 0, -1, -1, -1}, dy[8] = {-1, 0, 1, 1, -1, 1, 0, -1};
unsigned LL f[25][320005], ha[20], res;
LL A, t;
char s[210][210];
map<unsigned LL, int>mp;

LL gcd(LL x, LL y) {
    return !y ? x : gcd(y, x % y);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> n >> m >> k;
    k = min(k, n * m / (int)gcd(n, m));
    lg = log2(t = n * m * 8);
    for (int i = 0; i < n; i++) cin >> s[i];
    for (int d = 0; d < 8; d++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++) {
                f[0][Z(d, i, j)] = s[i][j] - 'a';
                g[0][Z(d, i, j)] = Z(d, (i + dx[d] + n) % n, (j + dy[d] + m) %
m);
            }
    ha[1] = 1e9 + 9;
    for (int i = 2; i <= lg; i++) ha[i] = ha[i - 1] * ha[i - 1];
    for (int i = 1; i <= lg; i++)
        for (int j = 1; j <= t; j++)
            f[i][j] = f[i - 1][j] * ha[i] + f[i - 1][g[i - 1][j]], g[i][j] = g[i
- 1][g[i - 1][j]];
    for (int i = 1; i <= t; i++) {
        res = 0, x = i;
        for (int j = 0; j <= lg; j++)
            if (k & (1 << j))
                res = res * ha[j + 1] + f[j][x], x = g[j][x];
        A += 2ll * mp[res] + 1, mp[res]++;
    }
    LL tt = gcd(A, t * t);
    printf("%lld/%lld", A / tt, t * t / tt);
    return 0;
}
```