Chapter 1

数学相关

1.1 Polya 原理、Burnside 引理

1.1.1 介绍

群：
给定一个集合 $G=\{a,b,c,...\}$ 和集合 G 上的二元运算 $\star$，并满足

- 封闭性：$\forall a,b\in G,\exists c\in G,a\star b=c.$
- 结合律：$\forall a,b,c\in G,(a\star b)\star c=a\star(b\star c).$
- 单位元：$\exists e\in G,\forall a\in G,a\star e=e\star a=a.$
- 逆元：$\forall a\in G,\exists b\in G,a\star b=b\star a=e,\ b=a^{-1}$

则称集合 G 在运算 $\star$ 之下是一个群，简称 G 是群。一般 $a\star b$ 简写为 ab。
置换：
n 个元素 $1,2,...,n$ 之间的置换 $\begin{pmatrix}1 & 2 & ... & n\\ a_1 & a_2 & ... & a_n\end{pmatrix}$, 表示 1 被 1 到 n 中的某个数 a_1 取代，2 被 1 到 n 中的某个数 a_2 取代，直到 n 被 1 到 n 中的某个数 a_n 取代，且 $a_1,a_2,...,a_n$ 互不相同
置换群：
置换群的元素是置换，运算是置换的连接。例如：
$\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}1&2&3&4\\4&3&2&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix}\begin{pmatrix}3&1&2&4\\2&4&3&1\end{pmatrix}=\begin{pmatrix}1&2&3&4\\2&4&3&1\end{pmatrix}$
Z_k(K 不动置换类)：
设 G 是 $1...n$ 的置换群。若 K 是 $1...n$ 中某个元素，G 中使 K 保持不变的置换的全体，记以 Z_k，叫做 G 中使 K 保持不动的置换类，简称 K 不动置换类。
E_k(等价类)：
设 G 是 $1...n$ 的置换群。若 K 是 $1...n$ 中某个元素，K 在 G 作用下的轨迹，记作 E_k。即 K 在 G 的作用下所能变化成的所有元素的集合。

$$|E_k|\cdot|Z_k|=|G|\quad k=1...n$$

$D(a_j)$ 表示在置换 a_j 下不变的元素的个数,s 表示置换种类数:

$$\sum_{i=1}^{n}|Z_i|=\sum_{i=1}^{s}D(a_i)$$

1.1.2 *Burnside* 引理

$L=\frac{1}{|G|}\sum_{i=1}^{n}|Z_i|=\frac{1}{|G|}\sum_{i=1}^{s}D(a_i)$, L 就是等价类数，也就是互异的组合状态的个数。
证明：不妨设 $N=1\ ......\ n$ 中共有 L 个等价类，$N=E^1+E^2+......+E^L$，则当 j 和 k 属于同一等价类时，有 $|Z_j|=|Z_k|$。所以

$$\sum_{i=1}^{n}|Z_i|=\sum_{i=1}^{L}\sum_{k\in E_i}|Z_k|=\sum_{i=1}^{L}|E_i|\cdot|Z_i|=L\cdot|G|$$

循环：

记 $(a_1a_2...a_n)$=$\begin{pmatrix} a_1 & a_2 & ... & a_{n-1} & a_n \\ a_2 & a_3 & ... & a_n & a_1 \end{pmatrix}$ 称为 n 阶循环。每个置换都可以写成若干互不相交的循环的乘积，两个循环 $(a_1a_2...a_n)$ 和 $(b_1b_2...b_n)$ 互不相交是指 $a_i \neq b_j, i, j = 1, 2, ..., n.$，

例如：$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix}$ =(1 3)(2 5)(4). 这样的表示是唯一的。置换的循环节数是上述表示中循环的个数。例如 (1 3)(2 5)(4) 的循环节数是 3。特别的：当 $n = 2$ 时，两阶循环 $(i\ j)$ 叫做 i 和 j 的对换。任何一个循环，都可以表达成若干换位之积。但是表达形式不尽统一，甚至连换位个数都不相同。

例如 (123) (12)(13) (13)(23) (21)(23) (12)(13)(31)(13)。尽管如此，有一个性质却是固有的，它不依换位的个数不同而异，那就是循环分解成换位的乘积时，换位个数奇偶性是不变的，或分解成奇数个换位之积或分解成偶数个换位之积。

若一个置换分解成奇数个换位之积，叫做奇置换；若分解成偶数个换位之积叫偶置换。单位置换为偶置换。

1.1.3 Polya 原理

设 G 是 p 个对象的一个置换群，用 m 种颜色涂染 p 个对象，$G = \{g_1, g_2, ..., g_s\}$,$g_i$ 是不同的置换选择，令 g_i 的循环节数为 $c(g_i)$ $(i \in [1, s])$, 则不同的染色方案数为:$L = \frac{1}{|G|}(m^{c(g_1)} + m^{c(g_2)} + ... + m^{c(g_s)})$

1.1.4 $n * n$ 的方阵，每个小格可涂 m 种颜色，求在旋转操作下本质不同的解的总数

我们可以在方阵中分出互不重叠的长为 $[\frac{n+1}{2}]$，宽为 $[\frac{n}{2}]$ 的四个矩阵。当 n 为偶数时，恰好分完；当 n 为奇数时，剩下中心的一个格子，它在所有的旋转下都不动，所以它涂任何颜色都对其它格子没有影响。令 m 种颜色为 $0 \sim m-1$，我们把矩阵中的每格的颜色所代表的数字顺次 (左上角从左到右，从上到下；右上角从上到下，从右到左；……) 排成 m 进制数，然后就可以表示为一个十进制数，其取值范围为 $0 \sim m^{\frac{n^2}{4}} - 1$。(因为 $\frac{n}{2} * \frac{n+1}{2} = \frac{n^2}{4}$) 这样，我们就把一个方阵简化为 4 个整数。我们只要找到每一个等价类中左上角的数最大的那个方案 (如果左上角相同，就顺时针方向顺次比较) 这样，在枚举的时候其它三个数一定不大于左上角的数，效率应该是最高的。进一步考虑，当左上角数为 i 时,$(i \in [0, R-1])$. 令 $R = m^{\frac{n^2}{4}}$ 可分为下列的 4 类：

1. 其它三个整数均小于 i，共 i^3 个。
2. 右上角为 i，其它两个整数均小于 i，共 i^2 个。
3. 右上角、右下角为 i，左下角不大于 i，共 $i+1$ 个。
4. 右下角为 i，其它两个整数均小于 i，且右上角的数不小于左下角的，共 $\frac{i\ *\ (i+1)}{2}$ 个。

因此:

$$\begin{aligned}
L &= \sum_{i=0}^{R-1}(i^3 + i^2 + i + 1 + \frac{1}{2}i(i+1)) = \sum_{i=0}^{R-1}(i^3 + \frac{3}{2}i^2 + \frac{3}{2}i + 1) \\
&= \sum_{i=1}^{R}((i-1)^3 + \frac{3}{2}(i-1)^2 + \frac{3}{2}(i-1) + 1) = \sum_{i=1}^{R}(i^3 - \frac{3}{2}i^2 + \frac{3}{2}i) \\
&= \frac{1}{4}R^2(R+1)^2 - \frac{3}{2} * \frac{1}{6}R(R+1)(2R+1) + \frac{3}{2} * \frac{1}{2}R(R+1) \\
&= \frac{1}{4}(R^4 + R^2 + 2R)
\end{aligned}$$

当 n 为奇数时，还要乘以一个 m.

采用 *Polya* 原理解决

确定置换群：

只有 4 个置换: 0° 90°, 180°, 270°, 所以置换群 $G = \{$ 0° 90°, 180°, 270°$\}$

计算循环节个数：

首先，给每个格子顺次编号 $1 \sim n^2$ ，再开一个二维数组记录置换后的状态。最后通过搜索计算每个置换下的循环节个数，效率为一次方级。

代入公式:

利用 *Plya* 定理得到最后结果。

$$L = \frac{1}{|\,G\,|}(m^{c(g_1)} + m^{c(g_2)} + ... + m^{c(g_s)})$$

当然也可以直接思考循环节：

- 当 n 为偶数，在转 $0°$ 时，循环节为 n^2 个，转 $180°$ 时，循环节为 $\frac{n^2}{2}$ 个，转 $90°$ 和 $270°$ 时，循环节为 $\frac{n^2}{4}$ 个
- 当 n 为奇数，在转 $0°$ 时，循环节为 n^2 个，转 $180°$ 时，循环节为 $\frac{n^2-1}{2}+1$ 个，转 $90°$ 和 $270°$ 时，循环节为 $\frac{n^2-1}{4}+1$ 个

综合考虑可得：$L = \frac{1}{4}(m^{n^2} + m^{\frac{n^2+3}{4}} + m^{\frac{n^2+1}{2}} + m^{\frac{n^2+3}{4}})$。可以发现这个式子，其实是和数学推导的式子完全吻合的。

```
typedef long long ll;
const int MAX_N = 10;
const ll mod = 1000000007;

int a[MAX_N][MAX_N], b[MAX_N][MAX_N];
int m, n; // m: 颜色数, n: 方阵大小
ll ans;

void Rotate() //逆时针旋转 90 度
{
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            a[n - j + 1][i] = b[i][j];
        }
    }
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            b[i][j] = a[i][j];
        }
    }
}

void CircleSection() //计算当前状态循环节数
{
    int num = 0; //记录循环节个数
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            if(a[i][j] > 0) { //搜索尚未被访问过的格子
                num++;
                int nexti, nextj, p;
                p = a[i][j];
                a[i][j] = 0;
                nexti = (p - 1) / n + 1;
                nextj = (p - 1) % n + 1; //得到这个循环的下一个格子
                while(a[nexti][nextj] > 0) {
                    p = a[nexti][nextj];
                    a[nexti][nextj] = 0; //已访问格子置零
                    nexti = (p - 1) / n + 1;
                    nextj = (p - 1) % n + 1;
                }
            }
        }
    }
    ll res = 1;
    for(int i = 1; i <= num; ++i ) {
        res = res * m % mod;
    }
    ans = (ans + res) % mod;
}
```

```

void init()
{
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= n; ++j) {
            b[i][j] = a[i][j] = (i - 1) * n + j;
        }
    }
}

int main()
{
    ll tt = quick_pow(4, mod - 2); //求除以 4 模 mod 的逆元
    while(~scanf("%d%d", &n, &m)){
        ans = 0;
        init(); //对方阵状态进行初始化
        CircleSection(); //旋转 0 度状态下的循环节个数
        Rotate();
        CircleSection();//逆时针 90 度
        Rotate();
        CircleSection(); //逆时针 180 度
        Rotate();
        CircleSection();//逆时针 270 度
        Rotate();
        ans = ans * tt % mod;
        printf("ans = %lld\n", ans); //根据 Polya 原理得到的结果

        ll R = quick_pow(m, n * n / 4);
        ll res = R * R % mod * R % mod * R % mod + R * R % mod + 2 * R % mod ;
        if(n & 1) res = res * m % mod;
        res = res * tt % mod;
        printf("res = %lld\n", res); //根据数学推导得到的结果
    }
    return 0;
}
```

1.1.5 各有 $a, b, c(a, b, c \geq 0, a + b + c \leq 40)$ 颗三种颜色，问这些珠子能串成的项链有多少种？考虑翻转和旋转。

[UVA 11255 Necklace]

令 $\sum_{i=1}^{3} color[i] = n$, 即珠子总数。

考虑旋转置换

我们考虑旋转 i 颗珠子的间距，则 $0, i, 2i...$ 构成一个循环，这个循环有 $\frac{n}{gcd(n,i)}$. 根据对成性，所有循环的长度均相同，因此一共有 $gcd(i, n)$ 个循环。循环与循环之间是等价类，所以在一个循环内的某颗珠子的颜色确定了，那么在其余循环内的同样地位位置的珠子颜色也就确定了。我们得到的答案是每个循环的方案数并且在旋转置换下，当循环节确定了，每个循环的方案数都是一样的。我们假设循环节的长度（也就是循环内元素的个数）为 $x \in [0, n)$，第 i 种颜色的珠子的个数为 $color[i]$ 个，如果 $color[i]\%x! = 0$, 那么第 i 种颜色的珠子在每个循环（等价类）内就不能均分，所以这种循环节就应该摒弃。当 $color[i]\%x == 0$ 时，令 $b[i] = \frac{color[i]}{x}$，则 $b[i]$ 就是每个循环（等价类）分得的这种颜色的珠子的个数。显然每个循环（等价类）内共有 $\sum_{i=1}^{3} b[i] = \frac{1}{x}\sum_{i=1}^{3} color[i]$ 颗珠子，记为 sum. 我们考虑单个循环（等价类）：有 sum 颗珠子，各个颜色分别有 $b[i]$ 颗，那么根据排列组合在循环节为 x 时，可得

$$Ans[x] = C_{sum}^{b[0]} * C_{sum-b[0]}^{b[1]} * C_{sum-b[0]-b[1]}^{b[2]}$$

考虑翻转置换

当 n 为奇数时，则对称轴上必有一点，对称轴有 n 条，每条对称轴形成 $\frac{n-1}{2}$ 个长度为 2 的循环和 1 个长度为 1 的循环。

当 n 为偶数时，有两种对称轴。穿过珠子的对称轴有 $\frac{n}{2}$ 条，各形成 $\frac{n}{2}-1$ 个长度为 2 的循环和两个长度为 1 的循环。不穿过珠子的对称轴也有 $\frac{n}{2}$ 条，各形成 $\frac{n}{2}$ 个长度 2 的循环。

可以发现不论 n 为奇偶，对称轴的总个数都为 n，同时旋转置换的个数也为 n, 所以根据 *Burnside* 引理最后需要除以 $2n$.

```
const int MAX_N = 42;

ll C[MAX_N][MAX_N];
int color[5], b[5], n;

void init() //组合数打表
{
    C[0][0] = 1;
    for(int i = 1; i < MAX_N; ++i) {
        C[i][0] = C[i][i] = 1;
        for(int j = 1; j < i; ++j) {
            C[i][j] = C[i - 1][j] + C[i - 1][j - 1];
        }
    }
}

int gcd(int x, int y)
{
    return y == 0 ? x : gcd(y, x % y);
}

ll CircleSection(int x)
{
    int sum = 0;
    for(int i = 1; i <= 3; ++i) {
        if(b[i] % x) return 0;
        b[i] /= x;
        sum += b[i];
    }
    ll res = 1;
    for(int i = 1; i <= 3; ++i) {
        res *= C[sum][b[i]];
        sum -= b[i];
    }
    return res;
}

ll Rotate() //旋转置换
{
    ll res = 0;
    for(int i = 0; i < n; ++i) {
        int d = gcd(i, n);
        memcpy(b, color, sizeof(color));
        res += CircleSection(n / d); // n/d 是循环元素个数
    }
    return res;
}

ll Flip() //翻转置换
{
    ll res = 0;
```

```
    if(n & 1) {
        for(int i = 1; i <= 3; ++i) {
            memcpy(b, color, sizeof(color));
            b[i]--;
            if(b[i] < 0) continue;
            res += CircleSection(2) * n; // n 条对称轴
        }
    }else {
        //穿过珠子
        for(int i = 1; i <= 3; ++i) {
            for(int j = 1; j <= 3; ++j) {
                memcpy(b, color, sizeof(color));
                b[i]--, b[j]--;
                if(b[i] < 0 || b[j] < 0) continue;
                res += CircleSection(2) * (n / 2);
            }
        }
        //不穿过珠子
        memcpy(b, color, sizeof(color));
        res += CircleSection(2) * (n / 2);
    }
    return res;
}

int main()
{
    init();
    int T;
    scanf("%d", &T);
    while(T--) {
        n = 0;
        for(int i = 1; i <= 3; ++i) {
            scanf("%d", color + i);
            n += color[i];
        }
        ll ans = 0;
        ans += Rotate();
        ans += Flip();
        printf("%lld\n", ans / (2 * n));
    }
    return 0;
}
```

1.1.6 给出 12 根等长的火柴棒，每根火柴棒的颜色属于 $1-6$ 中的一种，问能拼成多少种不同的正方体？(考虑旋转)

[UVA 10601 Cubes] 首先正方体的旋转置换有 24 种。下面将每个循环内元素的个数称为循环的长度。

注意是棱边的置换循环，而不是面的置换循环.

1. 静止。只有一种置换。有 12 个循环，每个循环的长度为 1。
2. 以相对面的中心为轴旋转。可以旋转的角度是 $90°, 180°, 270°$。选择轴有 3 种，每种轴下有 3 种旋转，所以有 3*3=9 种置换。
3. 旋转 $90°, 270°$，都是三个循环，每个循环的长度为 4.
4. 旋转 $180°$，有 6 个循环，每个循环的长度为 2

5. 以对边中点为轴，只可以旋转 180°，选择轴有 6 种，所以有 6*1=6 种置换。在这种旋转下有 5 个长度为 2 的循环和 2 个长度为 1 的循环

6. 以对顶点为为轴，可以旋转 120° 240°。选择轴有 4 种，所以有 4*2=8 种置换。

7. 旋转 $120°, 240°$ 都是有 4 个长度为 3 的循环。

找到了在每种置换下的循环个数和每个循环的长度就可以参考前面【例 2】处理旋转置换的方法解决。
需要特别关注的是，以对边中点为轴，只可以旋转 180°, 有 5 个长度为 2 的循环和 2 个长度为 1 的循环。
我们可以先枚举两个长度为 1 的循环选择的颜色，然后对于 5 个长度为 2 的循环就和上面的一样了。最后别忘了根据 *Burnside* 引理需要除以总的置换数 24。

```
ll work(int k)
{ //每 k 条边必须相同，分成 12/k 组以对边中点为轴旋转( 180° 是分成 5 组)
    memcpy (b, a, sizeof(a));
    int sum = 0;
    for (int i = 1; i <= 6; ++i) {
        if(b[i] % k) return 0;
        b[i] /= k;
        sum += b[i];
    }
    ll res = 1;
    for (int i = 1; i <= 6; ++i) {
        res *= C[sum][b[i]];
        sum -= b[i];
    }
    return res;
}

ll solve()
{
    ll res = 0;
    //静止
    res += work(1);
    //以相对面中心为轴
    res += (ll)3 * 2 * work(4); //旋转 90° 和270°
    res += (ll)3 * work(2); //旋转180°
    // 以对顶点为轴可以旋转, 120° 或 240°
    res += (ll)4 * 2 * work(3);
    // 以对边种点为轴，只能旋转 180°
    for(int i = 1; i <= 6; ++i) {
        for(int j = 1; j <= 6; ++j) {
            if(a[i] == 0 || a[j] == 0) continue;
            a[i]--; a[j]--; // 将 a[i] 和 a[j] 设为选择的两条对边的颜色
            res += (ll)6 * work(2);
        //剩下的是 5 个循环长度为 2 的循环, 6 代表对边选择情况
            a[i]++; a[j]++;
        }
    }
    return res / 24;
}
```

1.1.7 [POJ 2888 Magic Bracelet]

有一串 n 个珠子的项链，用 m 种颜色来染，有 k 个限制条件:$a[i]$ 和 $b[i]$ 不能相邻。问本质不同的项链有多少种？(考虑旋转，答案对 9973 取模，且 $gcd(n, 9973) = 1$)。数据范围:$n \le 10^9, 1 \le m \le 10, 0 \le k \le \frac{m*(m-1)}{2}$

假设有 k 个循环，用 $link[i][j]$ 表示第 i 种颜色能否和第 j 种颜色相邻: 当 $link[i][j] = 1$，表示 i 和 j 不能相邻，否则可以相邻。无向边:$link[i][j] = link[j][i]$ 。

用 $dp[k][i][j]$ 表示经过 k 个循环从第 i 种颜色转移到第 j 种颜色的方案数：

$$dp[k][i][j] = \sum_{p \le m} (1 - link[p][j]) * dp[k-1][i][p]$$

初始化：$dp[1][i][j] = 1 - link[i][j]$
初始矩阵:$A[i][j] = 1 - link[i][j]$ 来表示颜色间的连通性
由上面的递推式，$A^k[i][j]$ 代表的是: 从第 i 种颜色经过 k 个循环后变为第 j 种颜色的方法数。考虑循环，需要知道从 i 出发回到 i 的方案数即：$A^k[i][i]$。$solve(k)$ 表示循环个数为 k 时的方案数:

$$solve(k) = \sum_{i \le m} A^k[i][i]$$

离散数学里有：如果用 0,1 矩阵 A 来表示无向图的连通情况的话，A^k 代表的就是一个点经过 k 条路后能到达的地方的方法数。

假设对于每个循环的步长为 i，也就是 $0, i, 2i, ...$ 构成一个循环。这个循环的周期为 $\frac{i*n}{gcd(i,n)}$，所以这个循环有 $\frac{n}{gcd(i,n)}$ 个元素, 共有 $gcd(i,n)$ 个循环。所以枚举的循环个数一定是 n 的因子, 即：$k \mid n$。
满足循环个数为 k 的置换的旋转步长 i 满足 $gcd(i,n) = k$，此种置换的个数也就是 $gcd(\frac{i}{k}, \frac{n}{k}) = 1$ 的 i 的个数，即:$\phi(\frac{n}{k})$.

综上对于每个满足 $n\%k = 0$ 的 k 可以得到的方案数是

$$solve(k) * \phi(\frac{n}{k})$$

枚举每个 k 然后求和最后还要除以 n(因为总的置换数为 n), 又因为有模数且模数为素数，那么就相当于乘以 n^{mod-2}(费马小定理)。

$$Ans = \{\sum_{i|n} solve(i) * \phi(\frac{n}{i})\} * n^{p-2} \ \%p$$

```
const ll mod = 9973;

int link[15][15];

struct Matrix{
    int row, col;
    ll data[15][15];

    Matrix operator * (const Matrix& rhs) const { // 矩阵相乘条件:col = rhs.row
        Matrix res;
        res.row = row, res.col = rhs.col;
        for(int i = 1; i <= res.row; ++i) {
            for(int j = 1; j <= res.col; ++j) {
                res.data[i][j] = 0;
                for(int k = 1; k <= row; ++k) {
                    res.data[i][j] += data[i][k] * rhs.data[k][j];
                }
                res.data[i][j] %= mod;
            }
        }
        return res;
    }

    Matrix operator ^ (const int m) const {   // 矩阵快速幂
        Matrix res, tmp;
        res.row = row, res.col = col; //row == col
        memset(res.data, 0, sizeof(res.data));

        tmp.row = row, tmp.col = col;
        memcpy(tmp.data, data, sizeof(data));
        for(int i = 1; i <= res.row; ++i) { res.data[i][i] = 1; }
```

```
        int mm = m;
        while(mm) {
            if(mm & 1) res = res * tmp;
            tmp = tmp * tmp;
            mm >>= 1;
        }
        return res;
    }
};

inline ll solve(int len, int m)
{
    Matrix tmp;
    tmp.row = tmp.col = m;
    for(int i = 1; i <= m; ++i) {
        for (int j = 1; j <= m; ++j) {
            tmp.data[i][j] = 1 - link[i][j];
        }
    }
    tmp = tmp ^ len;

    ll ans = 0;
    for (int i = 1; i <= m; ++i) {
        ans = (ans + tmp.data[i][i]) % mod;
    }
    return ans;
}

int main()
{
    int T, n, m, t;
    scanf("%d", &T);
    while (T--) {
        memset(link, 0, sizeof(link));
        scanf("%d%d%d", &n, &m, &t);
        for(int i = 0; i < t; ++i) {
            int former, later;
            scanf("%d%d", &former, &later);
            link[former][later] = link[later][former] = 1;
        }

        ll ans = 0;
        for(int i = 1; i * i <= n; ++i) {
            if(n % i) continue;
            ans = (ans + phi(n / i) * solve(i, m)) % mod;
            if(n / i == i) continue;
            ans = (ans + phi(i) * solve(n / i, m)) % mod;
        }
        printf("%lld\n", ans * quick_pow(n % mod, mod - 2, mod) % mod);
    }
    return 0;
}
```

如果仅仅限制相邻的珠子颜色不能一样，总共有 m 种颜色。对于步长为 k 的循环，定义 $dp[i][0]$ 表示经历 i 个循环珠子颜色和循环起始珠子颜色一致的方案数，定义 $dp[i][1]$ 表示颜色不一样 (初始化 $dp[1][0] = 1, dp[1][1] = 0$)，那么可以得到转移方程:

$$dp[i][0] = dp[i-1][1]$$
$$dp[i][1] = dp[i-1][1] * (m-2) + dp[i][0] * (m-1)$$

如果 m 很大的话，就用矩阵快速幂进行加速。定义矩阵：$A = \begin{pmatrix} 0 & 1 \\ m-1 & m-2 \end{pmatrix}$,$B = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$,$C = A^k * B$. 那么步长为 k 的循环总的方案数（考虑起始珠子有 m 种颜色可选）是:

$$\phi(\frac{n}{k}) * m * C[1][1]$$

条件是 $n\%k = 0$, 最后还要乘上 n 模 mod 的逆元。

1.2 容斥原理

1.2.1 求区间 $[A, B]$ 中和 n 互素的数个数？$1 \leq A \leq B \leq 10^{15}, 1 \leq n \leq 10^9$

[HDU 4135]

我们将 n 进行素因子分解为 $p_1, p_2, p_3, \cdots, p_k$，先求解 $[1, r]$ 中和 n 不互素的数字个数。我们只需要考虑最大公约数是素因子的倍数的情况。选择素因子 num 个相乘，得到 mul，计算 $[1, r]$ 中和 n 不互素的数字个数是根据 num 的奇偶性：奇加偶减，$\frac{r}{mul}$。

有位运算和 dfs 两种写法。

```
vector<ll> fac;

void GetFactor(int n)
{
    fac.clear();
    for(ll i = 2; i * i <= n; ++i) {
        if(n % i == 0) {
            fac.push_back(i);
            while(n % i == 0) n /= i;
        }
    }
    if(n > 1) fac.push_back(n);
}
/*
ll solve(ll A, int n)
{
    ll ans = 0;
    int total = fac.size();
    for(ll i = 1; i < (1 << total); ++i) { // 用二进制位表示该位上对应编号的素因子是否选择
        int bits = 0;
        ll res = 1;
        for(int j = 0; j < total; ++j) {
            if(i & (1 << j)) { // 如果选择了第 j 个素因子
                bits++;
                res *= fac[j];
            }
        }
        if(bits & 1) ans += A / res; // 选择的素因子个数为奇数个
        else ans -= A / res;
    }
    return A - ans;
}
*/
ll ans;

void dfs(int cur, int num, ll mul, ll A)
{
    if(cur == fac.size()) {
        if(num & 1) ans -= A / mul;
        // A / mul 是最大公约数是 mul 的数字个数， ans 求的是互素数个数
        else ans += A / mul;
        return;
    }
    dfs(cur + 1, num, mul, A);
    dfs(cur + 1, num + 1, mul * fac[cur], A);
}

int main()
{
    int T, n, cases = 0;
```

```
    ll A, B;
    scanf("%d", &T);
    while(T--) {
        scanf("%lld%lld%d", &A, &B, &n);
        GetFactor(n);
        ans = 0;
        dfs(0, 0, 1, B);
        ll res = ans;
        ans = 0;
        dfs(0, 0, 1, A - 1);
        printf("Case #%d: %lld\n", ++cases, res - ans);
        //printf("Case #%d: %lld\n", ++cases, solve(B, n) - solve(A - 1, n));
    }
    return 0;
}
```

1.2.2 给一个 n，在 $p \in [1, n]$ 范围满足 $m^k = p(m \geq 1, k > 1$ 的数字 p 的个数。$1 \leq n \leq 10^{18}$

[HDU 2204] 我们可以枚举幂次 k，考虑到 $2^{60} > 10^{18}$，最多只需要枚举到 60 幂次。
同时对于一个数 p 的幂次 k 是个合数，那么 k 一定可以表示成 $k = r * k'$, 其中 k' 是素数的形式，那么：

$$p = m^k = m^{r*k'} = (m^r)^{k'}$$

所以我们只需要枚举素幂次 k 即可。
同时如果 $p^k \leq n$，那么对于任意的 $p' < p$，也一定满足 $p'^k \leq n$。所以对于每个 k 我们令 $p^k = n$，即 $p = n^{\frac{1}{n}}$，求出最大的 p，同时也就是满足 $p^k \leq n$ 的所有 p 的个数。但是这样子会有重复。例如：$k = 2$ 时，2^{2^3} 和 $k = 3$ 时,2^{3^2} 就重复计数了。这时候需要用容斥原理：加上奇数个素幂次相乘的个数，减去偶数个素幂次相乘的个数。
又因为 $2 * 3 * 5 < 60, 2 * 3 * 5 * 7 > 60$，那么最多只要考虑三个素幂次相乘情况。
时间复杂度：$O(3 * 2^{17})$(60 以内共 17 个素数)

```
const ll prime[20] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59};
const int len = 17;
const double eps = 1e-8;

ll ans;
double n;

void dfs(int cur, int num, int total, ll k)
{
    if(k > 60) return; // 素因子连乘最多不能超过 60 次幂，因为2 ^ 60 > 10 ^ 18
    if(num == total) {
        ll p = (ll)(pow(n, 1.0 / (0.0 + k)) + eps) - 1; // 先把 1 去掉, 精度误差eps
        ans += p;
        return ;
    }
    if(cur == len) return ;
    dfs(cur + 1, num, total, k); //第 i 个素数不选
    dfs(cur + 1, num + 1, total, k * prime[cur]); //第 i 个素数选择
}

int main()
{
    while(~scanf("%lf", &n)) {
        ll res = 0;
        for(int i = 1; i <= 3; ++i) {
            ans = 0;
            dfs(0, 0, i, 1);
//从下标 0 开始，当前选择素数个数为 0 需要选择素数个数, i 个选择素数乘积为,1
            if(i & 1) res += ans;
            else res -= ans;
```

```
        }
        res += 1; // 在1 dfs 时都没有统计
        printf("%lld\n", res);
    }
    return 0;
}
```

1.3 博弈论

1.3.1 威佐夫博弈 ($Wythoff\quad Game$)

有两堆石子各有 a,b 个，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。判断先手是胜者还是负者。

我们用 $(a_k,b_k),a_k \le b_k,k=0,1,2,...,n$ 表示两堆物品的数量并称其为局势，如果甲面对 (0 0)，那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是：$(0,0)$,$(1,2)$,$(3,5)$$(4,7)$,$(6,10)$,$(8,13)$,$(9,15)$,$(11,18)$,$(12,20)$。可以看出,$a_0=b_0=0$，$a_k$ 是未在前面出现过的最小自然数，而 $b_k=a_k+k$，奇异局势有如下三条性质：

1. 任何自然数都包含在一个且仅有一个奇异局势中。由于 a_k 是未在前面出现过的最小自然数，所以有 $a_k > a_{k-1}$，而 $b_k = a_k+k > a_{k-1}+k-1 = b_{k-1} > a_{k-1}$。所以性质 1 成立。
2. 任意操作都可将奇异局势变为非奇异局势。事实上，若只改变奇异局势 (a_k,b_k) 的某一个分量，那么另一个分量不可能在其他奇异局势中，所以必然是非奇异局势。如果使 (a_k,b_k) 的两个分量同时减少，则由于其差不变，且不可能是其他奇异局势的差，因此也是非奇异局势。
3. 采用适当的方法，可以将非奇异局势变为奇异局势。

从如上性质可知，两个人如果都采用正确操作，那么<u>面对非奇异局势，先拿者必胜；反之，则后拿者取胜</u>。
任给一个局势 (a,b)，判断它是不是奇异局势，有如下公式：

$$a_k = \lfloor k \times \frac{1+\sqrt{5}}{2} \rfloor, b_k = a_k + k \quad (k=0,1,2,...,n)$$

由于 $\frac{1+\sqrt{5}}{2} = \frac{2}{\sqrt{5}-1}$，可以先求出 $j = \lfloor a \times \frac{\sqrt{5}-1}{2} \rfloor$，则 $a_j = \lfloor j \times \frac{\sqrt{5}+1}{2} \rfloor$，

- 若 $a = \lfloor j \times \frac{\sqrt{5}+1}{2} \rfloor = a_j$ 那么判断 b 是否等于 $b_j = a_j + j = a + j$
- 若 a 不等于 a_j, 那么判断是否满足 $a = a_{j+1} = \lfloor (j+1) \times \frac{\sqrt{5}+1}{2} \rfloor$,$b = b_{j+1} = a_{j+1} + j + 1$
- 若都不是，那么就不是奇异局势。

```
double p = (sqrt(5.0) + 1.0) / 2.0;
if(a > b) swap(a, b);
int k = (int)(a / p);
if((a == (int)(k * p) && b == a + k) ||
    (a == (int)((k + 1) * p) && b == a + k + 1)) {
    printf("0\n"); // 先手输
} else {
    printf("1\n"); // 先手胜
}
```