第 4 章 数学基础

4.1 数论

4.1.1 数论基础

4.1.1.1 取模运算

定义：给定一个正整数 p，任意一个整数 n，一定存在等式：$n = kp + r$， 其中 k、r 是整数，且 $0 < r < p$，则称 k 为 n 除以 p 的商，r 为 n 除以 p 的余数。

对于正整数 p 和整数 a, b，定义如下取模运算：$a \,\%\, p (a \bmod p)$，表示 a 除以 p 的余数。

模 p 加法：其结果是 $a + b$ 算术和除以 p 的余数

模 p 减法：其结果是 $a - b$ 算术差除以 p 的余数

模 p 乘法：其结果是 $a * b$ 算术乘法除以 p 的余数

模 p 除法：除法的取模需要求逆元（见 4.1.7 乘法逆元）

$n \,\%\, p$ 得到结果的正负由被除数 n 决定,与 p 无关。例如：$7\%4 = 3, -7\%4 = -3, 7\%-4 = 3, -7\%-4 = -3$。

运算规则：

1. 模运算与加减乘幂运算：

- $(a + b) \,\%\, p = (a \,\%\, p + b \,\%\, p) \,\%\, p$
- $(a - b) \,\%\, p = (a \,\%\, p - b \,\%\, p + p) \,\%\, p$，加 p 是为了防止出现负数。
- $(a * b) \,\%\, p = (a \,\%\, p * b \,\%\, p) \,\%\, p$
- $a^b \,\%\, p = ((a \,\%\, p)^b) \,\%\, p$

2. 结合律：

- $((a + b) \,\%\, p + c) \,\%\, p = (a + (b + c) \,\%\, p) \,\%\, p$
- $((a * b) \,\%\, p * c) \%\, p = (a * (b * c) \,\%\, p) \,\%\, p$

3. 交换律：

- $(a + b) \,\%\, p = (b + a) \,\%\, p$
- $(a * b) \,\%\, p = (b * a) \,\%\, p$

4. 分配律：

- $(a + b) \,\%\, p = (a \,\%\, p + b \,\%\, p) \,\%\, p$
- $((a + b) \%\, p * c) \,\%\, p = ((a * c) \,\%\, p + (b * c) \,\%\, p) \,\%\, p$

取模运算的一个常用的重要性质为：$n\%i \Leftrightarrow n - i * \left\lfloor \frac{n}{i} \right\rfloor$

4.1.1.2 整除和同余

1. 整除

定义：如果 $\exists k \in Z$，使得 $b = a * k$；或者说 $b\%a == 0$。那么称 a 整除 b (或者说 b 被 a 整除)，记作 $a|b$。$a \mid b$ 等价于$b/a = k\ (k \in \mathrm{Z})$

整除的性质如下：

① 若 $a \mid b$ 且 $b \mid c$，那么有 $a \mid c$。

② 若 $b \mid a$ 且 $c \mid a$，且 b 和 c 互质，则 $(b * c) \mid a$。

③ 若 $a \mid b$ 且 $a \mid c$，则 $\forall x, y \in Z, a \mid (bx + cy)$。

④ 若 $a \mid b$ 且 $c \mid d$，那么 $ac \mid bd$。

⑤ 若 $p \mid (a - b)$，则 $a \equiv b \pmod{p}$

2. 同余

定义：设 p 是一个正整数，如果两个整数 a 和 b 满足 $a - b$ 能够被 p 整除，即 $(a - b)/p$ 得到一个整数，或者说 $p \mid (a - b)$。那么就称整数 a 与 b 对模 p 同余，记作 $a \equiv b \pmod{p}$。

设 a, b 是两个整数，若两数分别对正整数 p 取模，余数相同。那么就称整数 a 与 b 对模 p 同余，记作 $a \equiv b \pmod{p}$。

同余的性质如下：

①自反性：$a \equiv a \pmod{p}$

②对称性：若 $a \equiv b \pmod{p}$，则 $b \equiv a \pmod{p}$

③传递性：若 $a \equiv b \pmod{p}$，$b \equiv c \pmod{p}$，则 $a \equiv c \pmod{p}$。

④同加性：若 $a \equiv b \pmod{p}$，则 $a + c \equiv b + c \pmod{p}$。

⑤可加性：若 $a \equiv b \pmod{p}$，$c \equiv d \pmod{p}$，则 $a + c \equiv b + d \pmod{p}$

⑥同乘性：若 $a \equiv b \pmod{p}$，则 $a * c \equiv b * c \pmod{p}$

⑦可乘性：若 $a \equiv b \pmod{p}$，$c \equiv d \pmod{p}$，则 $a * c \equiv b * d \pmod{p}$

⑧同幂性：若 $a \equiv b \pmod{p}$，$a^n \equiv b^n \pmod{p}$

⑨同除性：若 $a * c \equiv b * c \pmod{p}$，则 $a \equiv b \pmod{p/\gcd(p, c)}$。特殊地，若 $\gcd(p, c) = 1$，则 $a \equiv b \pmod{p}$。

4.1.1.3 素数与合数

设整数 $p \neq 0, \pm 1$。如果 p 除了平凡约数外没有其他约数，那么称 p 为素数（不可约数）。若整数 $a \neq 0, \pm 1$ 且 a 不是素数，则称 a 为合数。

p 和 $-p$ 总是同为素数或者同为合数。如果没有特别说明，素数总是指正的素数。

若整数的因数是素数，则该素数称为该整数的素因数（素约数）。

素数分布定理：小于或等于 x 的素数的个数，用 $\pi(x)$ 表示。随着 x 的增大，有这样的近似结果：$\pi(x) \sim \frac{x}{\ln x}$。从不大于 n 的自然数随机选一个，它是素数的概率大约是 $\frac{1}{\ln n}$。

4.1.1.4 算数基本定理

算数基本定理（唯一分解定理）：

任何一个大于 1 的自然数 n，那么 n 可以唯一分解成有限个不相同的质数的乘积 $n = {p_1}^{a_1} * {p_2}^{a_2} *,\ldots,* {p_n}^{a_n}$，这里 p_i 均为质数，其中指数 a_i 是正整数。

4.1.2 素数

4.1.2.1 素数判定

判断一个数是不是素数的朴素算法从素数的性质入手：一个数 n 的因数只有 1 和它本身，因为一个数的因数关于 $\sqrt{n}$ 对称分布，那么只需要从 2 枚举到 $\sqrt{n}$，如果可以其中存在一个数可以整除 n 那么它不是素数，否则它是素数。

```
1. bool checkPrime(int n) {
2.     for (int i = 2; i * i <= n; i++) {
3.         if (n % i == 0) return false;
4.     }
5.     return true;
6. }
```

4.1.2.2 素数筛选

埃拉托斯特尼筛法，简称埃氏筛，是一种由希腊数学家埃拉托斯特尼所提出的一种简单检定素数的算法。要得到自然数 n 以内的全部素数，必须把不大于根号 n 的所有素数的倍数剔除，剩下的就是素数。即从 2 开始，每第一次出现且没有被标记过的数一定是素数，然后在 n 的范围内将它的所有倍数标记为合数。筛选出 n 以内的素数并将它们存在数组的算法代码如下：

```
1. const int maxn = 1e6 + 10;
2. int prime[maxn];
3. bool is_prime[maxn];
```

```
int p;

int Eratosthenes(int n) {
    memset(is_prime, 1, sizeof(is_prime));
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            prime[p++] = i;
            if (1LL * i * i <= n)
                for (int j = i * i; j <= n; j += i)
                    // 因为从 2 到 i - 1 的倍数我们之前筛过了，这里直接从 i 的倍数开始
                    is_prime[j] = false;    // i 的倍数标记为合数
        }
    }
    return p;
}
```

关于上述算法中，为什么当 $i*i \leq n$ 时才进行筛选，考虑一个大于 $\sqrt{n}$ 的合数，它一定不存在两个大于 $\sqrt{n}$ 的素因子，根据唯一分解定理它一定存在小于 $\sqrt{n}$ 的素因子，而满足这样条件的素因子的所有倍数都被标记过，那么大于 $\sqrt{n}$ 的数就无需继续判断。如果取出所有的素数，只需要一个判断的标记数组，那么上述埃氏筛代码还可以进一步简化。

埃氏筛的时间复杂度为 $O(n\text{log}\text{log}n)$。

根据埃氏筛，如何求出 n 以内所有数的最大质因子？显然，在埃氏筛的过程中对素数是从小到大枚举判定的，对所有素数的倍数，更新其最大质因子即可。

```
const int maxn = 1e6 + 10;
int max_prime[maxn];

void getMaxPrime() {
    max_prime[1] = 1;
    for (int i = 2; i < maxn; i++) {
        if (!max_prime[i]) {
            for (int j = i; j < maxn; j += i) max_prime[j] = i;
        }
    }
}
```

任意一个正整数 k，若 $k > 2$，则 k 可以表示成若干个质数相乘的形式。埃氏筛法中，在枚举 k 的每一个质因子时，都计算了一次 k，从而造成了冗余。因此在改进算法中，只利用 k 的最小质因子去计算一次 k。这种筛法又被称为欧拉筛或者线性筛。

与埃氏筛法不同的是，对于外层枚举 i，无论 i 是质数还是合数，我们都会用 i 的倍数去筛。但在枚举的时候，我们只枚举 i 的质数倍。此外，在从小到大依次枚举质数 p 来计算 i 的倍数时，还需要检查 i 是否能够整除 p。若 i 能够整除 p，则停止枚举。若 i 能整除

p，因为我们是从小到大枚举素数的，那么又可以说 i 已经被 p 筛过了，所以 i 乘其他质数的结果也一定是 p 的倍数，后面的数已经通过 p 筛出而不需再筛。综上，线性筛法可以保证每个合数只会被枚举到一次，时间复杂度为 $O(n)$。

欧拉筛的具体证明需要学习欧拉函数，见 4.1.8.2 欧拉函数。

```
int cnt, prime[maxn];
bool vis[maxn];

void euler(int n) {
    for (int i = 2; i <= n; ++i) {
        if (!vis[i]) prime[cnt++] = i;
        for (int j = 0; j < cnt && 1ll * i * prime[j] <= n; ++j) {
            vis[i * prime[j]] = 1;
            if (i % prime[j] == 0) break;
        }
    }
}
```

正因为线性筛是依据最小质因子来筛的算法，那么可以通过线性筛对范围内的数打一个最小质因子的表。

4.1.2.3 反素数

对于任何正整数 n，其约数个数记为 $f(n)$。如果某个正整数 n 满足：对任何正整数 $i(0<i<n)$，都有 $f(i)<f(n)$，那么称 n 为反素数。

性质：

1. 一个反素数的所有质因子必然是从 2 开始的连续若干质数，因为反素数是保证约数个数为 x 的这个数 n 尽量小

2. 设反素数 $n=2^{t_1}\times 3^{t_2}\times 5^{t_3}\times\ldots$，那么必有 $t_1\geq t_2\geq t_3\geq\cdots\geq t_n$。

反素数相关问题只需要搜索即可。我们可以把当前走到每一个素数前面的时候列举成一棵树的根节点，然后一层层的去找，出现以下条件停止：

1) 当前走到的数字已经大于我们想要的数字了

2) 当前枚举的因子已经用不到了（和 1 重复了）

3) 当前因子大于我们想要的因子了

4) 当前因子正好是我们想要的因子（此时判断是否需要更新最小 ans）

然后 dfs 里面不断一层一层枚举次数继续往下迭代可以，常见的反素数问题：

1. 给定一个数 n，求出一个最小的非负整数 x，使得它的约数个数为 n。

首先筛选出素数，然后搜索。注意三次剪枝，第二次剪枝写成除法是为了防止溢出。

```
int prime[] = {2,  3,  5,  7,  11, 13, 17, 19, 23, 29, 31, 37, 41,
               43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
```

```
int n;
unsigned long long ans;

void dfs(int deep, int limit, unsigned long long cur, int num) {
    if (num > n) return;
    if (num == n && ans > cur) {
        ans = cur;
        return;
    }
    for (int i = 1; i <= limit; i++) {
        if (ans / prime[deep] < cur || num * (i + 1) > n)
            break;   // 剪枝：不超过当前答案最小值且因数个数不超过 n
        cur *= prime[deep];
        if (n % (num * (i + 1)) == 0)   // 剪枝：只有当前因数整除 n 才能继续搜
            dfs(deep + 1, i, cur, num * (i + 1));
    }
}
```

2. 求 $[1,n]$ 中因数最多的数，有多个解输出最小的解。

显然因数最多的数一定是反素数。

```
ll n, ans;
int cnt;
int prime[] = {2,  3,  5,  7,  11, 13, 17, 19, 23, 29, 31, 37, 41,
               43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

void dfs(int deep, int num, int limit, ll cur) {
    if (cur > n || deep >= 16) return;
    if (num > cnt) {
        cnt = num;
        ans = cur;
    }
    if (num == cnt && ans > cur) ans = cur;
    for (int i = 1; i <= limit; i++) {
        if (n / prime[deep] < cur) break;
        cur *= prime[deep];
        dfs(deep + 1, num * (i + 1), i, cur);
    }
}
```

3. 求 n 范围内的反素数表。

类似埃氏筛，枚举所有的因子，更新其所有的倍数，最后得到的即区间内因数个数为 $d[i]$ 的最小的 i。时间复杂度 $O(n\log\log n)$。

```
int Map[maxn], d[maxn];

```

```
void init() {
    for (int i = 1; i < maxn; i++) {
        for (int j = i; j < maxn; j += i) d[j]++;
        if (!Map[d[i]]) Map[d[i]] = i;
    }
}
```

4.1.2.4 例题

素数密度（洛谷 P1835）

给定区间 $[L,R],1 \le L \le R < 2^{31}, R-L \le 10^6$ 请计算区间中素数的个数。

【输入格式】

两个正整数 L,R。

【输入格式】

一行，一个整数，表示区间中素数的个数。

【分析】

因为一个非素数是被它的最小素数因子筛掉，$\sqrt{2^{31}-1}$ 内的数要么是素数，要么是能被 $\sqrt{2^{31}-1}$ 内的素数整除的合数，也就是说，$[L,R]$ 区间的所有非素数的素数因子都在 $\sqrt{2^{31}-1}$ 内，预先将 $\sqrt{2^{31}-1}$ 内的所有素数找出来，然后用这些素数去筛掉指定区间的所有非素数。

求出某个区间的素数且这些素数可能很大，需要进行下标偏移。

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

using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 1e6 + 10;
int up = 0x7fffffff;

vector<int> prime, ans;
bool is_prime[maxn], vis[maxn];

void euler() {
    int m = sqrt(up + 0.5);
    prime.clear();
    memset(is_prime, 1, sizeof(is_prime));
```

```
16.     is_prime[0] = is_prime[1] = 0;
17.     for (int i = 2; i <= m; i++) {
18.         if (is_prime[i]) prime.push_back(i);
19.         for (int j = 0; j < prime.size() && 1LL * i * prime[j] <= m; j++) {
20.             is_prime[i * prime[j]] = 0;
21.             if (i % prime[j] == 0) break;
22.         }
23.     }
24. }
25.
26. void solve(ll l, ll r) {
27.     memset(vis, 1, sizeof vis);
28.     if (l == 1) l++;
29.     for (int i = 0; i < prime.size() && 2LL * prime[i] <= r; i++) {
30.         if (prime[i] < l) {
31.             ll x = l / prime[i], L;
32.             if (x * prime[i] < l)
33.                 L = (x + 1) * prime[i];
34.             else
35.                 L = x * prime[i];
36.             for (ll j = L; j <= r; j += prime[i]) vis[j - l] = 0;
37.         } else {
38.             for (ll j = 2LL * prime[i]; j <= r; j += prime[i]) vis[j - l] = 0;
39.         }
40.     }
41.     ans.clear();
42.     for (int i = 0; i <= r - l; i++)
43.         if (vis[i]) ans.push_back(i + l);
44.     cout << ans.size() << endl;
45. }
46.
47. int main() {
48.     // freopen("out.txt", "r", stdin);
49.     // freopen("ans.txt", "w", stdout);
50.     ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
```

```
    int l, r;
    euler();
    cin >> l >> r;
    solve(l, r);
    return 0;
}

```

漂亮数（牛客 NC224933）

小红定义一个数满足以下条件为“漂亮数”：

1. 该数不是素数。

2. 该数可以分解为 2 个素数的乘积

【输入格式】

第一行输入一个正整数 $t(1 \le t \le 10^5)$ ，代表有t次询问

两个正整数 $l,r(1 \le l \le r \le 10^8)$ ，用空格隔开。

【输入格式】

共输出 t 行，每行为一个整数，代表区间$[l,r]$中漂亮数的数量。

【分析】

素数筛中若$prime[j]$本身就是一个素数，如果i也是素数，$prime[j] * i$就是漂亮数。同一个漂亮数，有没有可能是另外两个素数的乘积呢？ 质因数分解，既然两个质数的乘积就是这个数，那就不会有别的质数的乘积是这个数。

前缀和预处理一下，得到每个区间内漂亮数的数量。

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

using namespace std;

const int N = 1e8 + 10;

int cnt;
bool st[N];
bool vis[N];
int sum[N];
int primes[N];

void Init() {
    for (int i = 2; i <= N; i++) {
```

```
15.         if (!st[i]) primes[cnt++] = i;
16.         for (int j = 0; primes[j] <= N / i; j++) {
17.             st[primes[j] * i] = true;
18.             if (!st[i]) vis[primes[j] * i] = true;
19.             if (i % primes[j] == 0) break;
20.         }
21.     }
22.
23.     for (int i = 1; i <= N; i++) {
24.         sum[i] = sum[i - 1];
25.         if (vis[i]) sum[i]++;
26.     }
27. }
28.
29. int main() {
30.     Init();
31.     int T;
32.     scanf("%d", &T);
33.     while (T--) {
34.         int l, r;
35.         scanf("%d%d", &l, &r);
36.         printf("%d\n", sum[r] - sum[l - 1]);
37.     }
38.     return 0;
39. }
40.
```


The Number of Pairs（Codeforces 1499 D）

给定c, d, x，求出满足$c * \text{lcm}(a, b) - d * \text{gcd}(a, b) = x$的$(a, b)$的对数。

【输入格式】

第一行输入一个正整数 $t(1 \le t \le 10^4)$，代表有 t 个测试用例。

第二行输入两个正整数$c, d, x(1 \le c, d, x \le 10^7)$，用空格隔开。

【输入格式】

对于每个测试用例输出一行，代表满足上式 (a, b)的对数。

【分析】

首先是推导式子，这种式子的下手点一般就是根据都是正整数，那么转化过程中涉及

到除法的式子一定可以整除。设$\gcd(a,b)=g$，那么：

$$c*\frac{a*b}{g}-d*g=x$$

$$c*\frac{a*b}{g}=x+d*g$$

$$c=\frac{d*g+x}{\frac{a*b}{g}}$$

最终化为了：$c=\frac{d+\frac{x}{g}}{\frac{a}{g}*\frac{b}{g}}$，根据上面所说，$c$是整数那么等号右边的中间结果都是整数，因此$g|x$；然后又可以观察到分母是两个互质的数的乘积，实际上答案已经浮出水面了，就是枚举x的所有因数g，设求出$\frac{d+\frac{x}{g}}{c}$能被分成m对互质的数的乘积，实际上m就是该数的质因子的种类数，根据二项式定理得出贡献的对数为2^m。

在埃氏筛的第二层循环可以对每个数累加质因子的种类数！这样预处理的时间复杂度为$O(2e^7*\log\log 2e^7)$，至于因数分解直接$O(\sqrt{x})$即可。

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

using namespace std;
typedef long long ll;
const int maxn = 2e7 + 10;

int num[maxn];
bitset<maxn> vis;

void init() {
    for (int i = 2; i < maxn; i++) {
        if (!vis[i]) {
            num[i] = 1;
            for (int j = i + i; j < maxn; j += i) {
                vis[j] = 1;
                num[j]++;
            }
        }
    }
}

```

```cpp
vector<int> fac;

int main() {
    ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T;
    int c, d, x;
    init();
    cin >> T;
    while (T--) {
        cin >> c >> d >> x;
        fac.clear();
        for (int i = 1; i * i <= x; i++) {
            if (x % i == 0) {
                int j = x / i;
                if (i == j)
                    fac.push_back(i);
                else
                    fac.push_back(i), fac.push_back(j);
            }
        }
        ll ans = 0;
        for (auto g : fac) {
            int sum = d + x / g;
            if (sum % c) continue;
            ans += 1LL << num[sum / c];
        }
        cout << ans << "\n";
    }
    return 0;
}

```

4.1.3 约数

4.1.3.1 约数

1. 一些常见概念：

（1）约数（因数）：若 $a \mid b$，则称 b 是 a 的倍数，a 是 b 的约数。

（2）0 是所有非 0 整数的倍数。对于整数 $b \neq 0$，b 的约数只有有限个。

（3）平凡约数（平凡因数）：对于整数 $b \neq 0$，± 1、$\pm b$ 是 b 的平凡约数。当 $b = \pm 1$ 时，b 只有两个平凡约数。

（4）对于整数 $b \neq 0$，b 的其他约数称为真约数（真因数、非平凡约数、非平凡因数）。

2. 约数的性质：

（1）设整数 $b \neq 0$。当 d 遍历 b 的全体约数的时候，$\frac{b}{d}$ 也遍历 b 的全体约数。

（2）设整数 $b > 0$，则当 d 遍历 b 的全体正约数的时候，$\frac{b}{d}$ 也遍历 b 的全体正约数。

4.1.3.2 因数分解

一个数 n 的所有因数均关于 $\sqrt{n}$ 的对称分布，故只需遍历到 $\sqrt{n}$ 即可，然后只要遇到能整除的 i，那么 n/i 肯定也是 n 的因数。注意当因数刚好等于 $\sqrt{n}$ 时 $i = n/i$，所以加个特判。时间复杂度 $O(\sqrt{n})$。

```
vector<int> factor;

void getFactor(int n) {
    factor.clear();
    for (int i = 1; 1ll * i * i <= n; i++)
        if (n % i == 0) {
            if (i * i == n)
                factor.push_back(i);
            else {
                factor.push_back(i);
                factor.push_back(n / i);
            }
        }
}
```

4.1.3.3 最大公约数

最大公约数即为 Greatest Common Divisor，常缩写为 gcd。一组整数的公约数，是指同时是这组数中每一个数的约数的数。1 是任意一组整数的公约数。一组整数的最大公约数，是指所有公约数里面最大的一个。那么如何求最大公约数呢？我们先考虑两个数的情况。

如果我们已知两个数 a 和 b，如何求出二者的最大公约数呢？常用的是欧几里得算法，即辗转相除法。

令 $r_0 = a, r_1 = b$，不妨设 $a > b$。反复运用带余除法式，可以求出二者的最大公约数：

$$r_0 = q_1 r_1 + r_2, 0 < r_2 < r_1$$

$$r_1 = q_2 r_2 + r_3, 0 < r_3 < r_2$$

$$\cdots\cdots$$

$$r_{n-2} = q_{n-1} r_{n-1} + r_n, 0 < r_n < r_{n-1}$$

$$r_{n-1} = q_n r_n + r_{n+1}, \qquad r_{n+1} = 0$$

由于 $r_{n+1} < r_n < r_{n-1} < \cdots < r_2 < r_1 = b$，所以经过有限的步骤，必然存在 n，使得 $r_{n+1} = 0$。

我们发现如果 b 是 a 的约数，那么 b 就是二者的最大公约数。下面讨论不能整除的情况，即 $a = b * q + r$，其中 $r < b$。我们通过证明可以得到 $\gcd(a,b) = \gcd(b, a \% b)$，过程如下：

1. 设 $a = bk + c$，显然有 $c = a \% b$。设 $d \mid a, d \mid b$，则 $c = a - bk, \frac{c}{d} = \frac{a}{d} - \frac{b}{d}k$。由右边的式子可知 $\frac{c}{d}$ 为整数，即 $d \mid c$，所以对于 a, b 的公约数，它也会是 $b, a \% b$ 的公约数。
2. 反过来也需要证明：设 $d \mid b, d \mid (a \% b)$，我们还是可以像之前一样得到 $\frac{a\%b}{d} = \frac{a}{d} - \frac{b}{d}k, \frac{a\%b}{d} + \frac{b}{d}k = \frac{a}{d}$。因为左边式子显然为整数，所以 $\frac{a}{d}$ 也为整数，即 $d \mid a$，所以 $b, a\%b$ 的公约数也是 a, b 的公约数。
3. 既然两式公约数都是相同的，那么最大公约数也会相同。

联系上述带余式的辗转相除法，可以得到 $(a,b) = (r_0, r_1) = (r_1, r_2) = \cdots = (r_{n-1}, r_n) = (r_n, r_{n+1}) = (r_n, 0) = r_n$。

所以得到递归式 $\gcd(a,b) = \gcd(b, a \% b)$，这里两个数的大小是不会增大的，那么我们也就得到了关于两个数的最大公约数的一个递归求法。

```
1. int gcd(int a, int b) {
2.     return b == 0 ? a : gcd(b, a % b);
3. }
```

下面证明，欧几里得算法的时间复杂度为 $O(\log n)$。

证明：当我们求 $\gcd(a,b)$ 的时候，会遇到两种情况：

- $a < b$，这时候 $\gcd(a,b) = \gcd(b,a)$；
- $a > b$，这时候 $\gcd(a,b) = \gcd(b, a \% b)$，而对 a 取模会让 a 至少折半。这意味着这一过程最多发生 $O(\log n)$ 次。

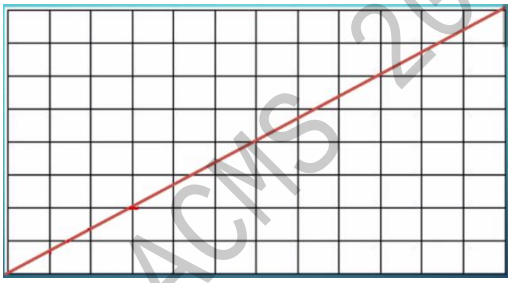
第一种情况发生后一定会发生第二种情况，因此第一种情况的发生次数一定不多于第二种情况的发生次数。从而我们最多递归 $O(\log n)$ 次就可以得出结果。

最大公因数的常见性质如下：

- $\gcd(a,b) = c \rightarrow \gcd(a/c, b/c) = 1$
- $\gcd(a*b,c), a \perp c \rightarrow \gcd(a*b,c) = \gcd(b,c)$
- $\gcd(a,b,c) = \gcd(\gcd(a,b),c)$
- 两个数 a,b 互质的充要条件为 $\gcd(a,b) = 1$，也记作 $(a,b) = 1$

最大公因数的一些应用如下：

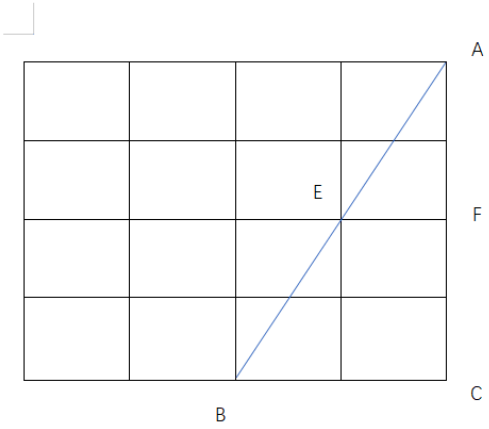
1. 如下图$n \times m$的方格棋盘，从左下角到右上角穿过一条直线，求直线和多少个方格内部有交点。



观察每一个点，不难发现每当和一条纵线相交后，下一次一定个横线相交，不考虑特殊点的情况下是 $n+m$。但是像图中加粗的点那样，就重复计算了一次，因此需要找出这种特殊点的个数。从左下角开始找第一个特殊点，其左下角为 2×3 的矩形；然后找第二个特殊点，左下角为 4×6 的矩形；第三个为 6×9 的矩形.....我们发现一个共同点，就是这些矩形都和 $n \times m$ 的矩形相似，所以特殊点的个数也就是 $\gcd(n,m)$，故答案就是 $n + m - \gcd(n,m)$。

2. 有一个 $n \times n$ 的方阵，给出方阵中的两个坐标 $(x_1,y_1),(x_2,y_2)$，如何判断两点的连线不会和其他点相交。

如图所示，显然如果两个点之间会有交点，那么△ABC 和△AEF 会构成相似三角形，和第一个例子类似，那么就出现 $gcd(|BC|,|AC|) > 1$。



因此令$a = \text{abs}(x_1 - x_2), b = \text{abs}(y_1 - y_2)$，当且仅当 $\gcd(a, b) = 1$ 时两点的连线才不会和其他点相交。

4.1.3.3 最小公倍数

一组整数的公倍数，是指同时是这组数中每一个数的倍数的数。0 是任意一组整数的公倍数。一组整数的最小公倍数，是指所有正的公倍数里面，最小的一个数。两个数 a, b 的最小公倍数记作 $\text{lcm}(a, b)$

如何求出最小公倍数，要从 gcd/lcm 在唯一分解定理下的意义：

假设 a 经过质因数分解后为 ${p_1}^{k_1} * {p_2}^{k_2}, \ldots * {p_n}^{k_n}$，$b$ 经过质因数分解为${p_1}^{t_1} * {p_2}^{t_2}, \ldots * {p_n}^{t_n}$。那么 $\gcd(a, b) = {p_1}^{\min(k_1, t_1)} * {p_2}^{\min(k_2, t_2)}, \ldots * {p_n}^{\min(k_n, t_n)}$。$\text{lcm}(a, b) = {p_1}^{\max(k_1, t_1)} * {p_2}^{\max(k_2, t_2)}, \ldots * {p_n}^{\max(k_n, t_n)}$。

由于 $k_i + t_i = \max(k_i, t_i) + \min(k_i, t_i)$。所以得到结论是 $gcd(a, b) \times \text{lcm}(a, b) = a \times b$。要求两个数的最小公倍数，先求出最大公约数即可。

```
1. int lcm(int a, int b) {
2.     return a / gcd(a, b) * b;
3. }
```

求多个数的最小公倍数，和求多个数的最大公因数同理。

4.1.3.4 一些定理

1. 因数个数定理：

若 n 可以唯一分解成有限个质数的乘积 $n = {p_1}^{a_1} * {p_2}^{a_2} *, \ldots, * {p_n}^{a_n}$，则 n 的因数个数为 $(a_1 + 1) * (a_2 + 1) * (a_3 + 1) * \ldots * (a_n + 1)$

2. 因数和定理：

对于 $n = {p_1}^{a_1} * {p_2}^{a_2} *, \ldots, * {p_n}^{a_n}$，其所有的因数无非是选择其中的若干个素数及幂次相乘得到，根据乘法原理，所有因数的和为：

$$(p_1^0+p_1^1+p_1^2+\ldots+p_1^{a_1})*(p_2^0+p_2^1+\ldots+p_2^{a_2})\ldots(p_n^0+p_n^1+p_n^2+\ldots+p_n^{a_n})$$

4.1.3.5 例题

Orac and LCM（Codeforces 1350 C）

给出一个序列 a，求出$gcd(\{lcm(a_i,a_j)|i<j\})$。

【输入格式】

第一行输入一个正整数 $n(2\le n\le 10^5)$ 。

第二行输入序列的n个正整数$a_1,a_2,\ldots,a_n(1\le a_i\le 2\times 10^5)$。

【输入格式】

输出一个正整数表示计算上式的答案。

【分析】

做法一：

实际上不难发现序列中a_1对应的lcm对为$n-1$对，a_2对应的$n-2$对...以此类推，关键是如何化简如下的表达式：

$$gcd\{lcm(a_1,a_2)，lcm(a_1,a_3)，\ldots，lcm(a_1,a_n)\}$$

根据 gcd 和 lcm 在质因数分解下的意义，那么现在我们再看上式，对于所有数的一个公共的质因子p，假设幂次分别为$k_1,k_2,\ldots k_n$，上式的意义为：

$$min\{max(k_1,k_2)，max(k_1,k_3)，\ldots，max(k_1,k_n)\}$$

然后该式等价于：

$$max\{k_1，min\{k_2,k_3,\ldots,k_n)\}\}$$

即化为表达式：

$$lcm\{k_1，gcd\{k_2,k_3,\ldots,k_n\}\}$$

因此我们得到等式：

$$gcd\{lcm(a_1,a_2)，lcm(a_1,a_3)，\ldots，lcm(a_1,a_n)\}=lcm\{k_1，gcd\{k_2,k_3,\ldots,k_n\}\}$$

最后对于本题，我们只需预处理后缀的gcd即可

做法二：

根据上述在质因数分解意义下的 gcd 和 lcm，实际上就是需要求出n个数中同时出现最少$n-1$次的质因子中幂次最大和次大的那个，如下代码给出了第二种做法。

```
1. #include <bits/stdc++.h>
2.
3. using namespace std;
4. typedef long long ll;
5. const int maxn = 2e5 + 10;
6.
```

```
unordered_map<int, int> fi, se;
unordered_set<int> st;
int minp[maxn], prime[maxn], vis[maxn];
int cnt;

void euler() {
    for (int i = 1; i < maxn; i++) minp[i] = i;
    for (int i = 2; i < maxn; i++) {
        if (minp[i] == i) {
            prime[cnt++] = i;
        }
        for (int j = 0; j < cnt && i * prime[j] < maxn; j++) {
            minp[i * prime[j]] = prime[j];
            if (i % prime[j] == 0) break;
        }
    }
}

void divide(int n) {
    map<int, int> mp;
    int fac, p, num;
    while (n > 1) {
        fac = minp[n];
        n /= fac;
        mp[fac]++;
    }
    for (auto i : mp) {
        p = i.first, num = i.second;
        vis[p]++, st.insert(p);
        if (!fi.count(p)) {
            fi[p] = num;
            continue;
        }
        if (!se.count(p)) {
            if (num < fi[p]) {
```

```
42.                 se[p] = fi[p];
43.                 fi[p] = num;
44.             } else
45.                 se[p] = num;
46.             continue;
47.         }
48.         if (num <= fi[p]) {
49.             se[p] = fi[p];
50.             fi[p] = num;
51.         } else if (num < se[p]) {
52.             se[p] = num;
53.         }
54.     }
55. }
56.
57. ll qkp(ll x, ll n) {
58.     ll ans = 1;
59.     while (n) {
60.         if (n & 1) ans = ans * x;
61.         x *= x;
62.         n >>= 1;
63.     }
64.     return ans;
65. }
66.
67. int main() {
68.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
69.     int n;
70.     cin >> n;
71.     euler();
72.     for (int i = 1, x; i <= n; i++) {
73.         cin >> x;
74.         divide(x);
75.     }
76.     ll ans = 1;
```

```
    for (auto p : st) {
        if (vis[p] >= n - 1) {
            if (vis[p] == n - 1)
                ans *= qkp(p, fi[p]);
            else
                ans *= qkp(p, se[p]);
        }
    }
    cout << ans << "\n";
    return 0;
}

```

Magical GCD（UVA 1642）

给定一个长度为 $n(1 \le n \le 10^5)$的序列，每个数$a_i(1 \le a_i \le 10^{12})$，找一个连续子序列使得子序列的公约数与长度的乘积最大，并输出这个最大值。

【输入格式】

第一行输入一个正整数 T，代表测试用例的数量。

对于每个测试用例，第一行输入 n 个正整数表示序列的长度。

接下来一行输入n个数的序列$a_1, a_2, \ldots, a_n$。

【输入格式】

共T行，每行表示一个测试用例的答案。

【分析】

给定序列a，连续子段的gcd有$log(max\{a_i\})$种可能。证明：考虑 gcd 的性质，设$x = p_1^{a_1} * p_2^{a_2} \ldots, y = p_1^{b_1} * p_2^{b_2} \ldots$，那么$gcd(x,y) = p_1^{min(a_1,b_1)} * p_2^{min(a_2,b_2)} \ldots$，此时考虑再加入一个数$z$求$gcd$，如果要发生变化$p_1, \ldots, p_n$对应的$min$至少有一个需要减少，然后因为最小的质数为2，那么最多减少log次就变成了1，得证。

考虑枚举右界j，我们只需每次和之前出现的gcd再求出新的gcd并更新答案，因为gcd是单调不增的，而下标从小到大是单调递增的，因此如果出现相同的gcd只需保留之前的下标才能使答案尽可能的大。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;

```

```
6. ll a[maxn];
7. ll mg[105], pos[105];
8.
9. ll gcd(ll a, ll b) {
10.     return !b ? a : gcd(b, a % b);
11. }
12.
13. int main() {
14.     int t, n, cnt;
15.     scanf("%d", &t);
16.     while (t--) {
17.         scanf("%d", &n);
18.         ll ans = 0;
19.         int cnt = 0;
20.         for (int i = 1; i <= n; i++) {
21.             scanf("%lld", &a[i]);
22.             ans = max(ans, a[i]);              // 特判一个数是否为最优解
23.             for (int j = 1; j <= cnt; j++) {   // 枚举之前的 gcd
24.                 mg[j] = gcd(mg[j], a[i]);
25.                 if (mg[j] == mg[j - 1]) {   // 如果出现相同则整体左移
26.                     for (int k = j; k < cnt; k++) mg[k] = mg[k + 1], pos[k] =
pos[k + 1];
27.                     cnt--, j--;
28.                 } else
29.                     ans = max(ans, (i - pos[j] + 1) * mg[j]);   // 否则更新答案
30.             }
31.             mg[++cnt] = a[i], pos[cnt] = i;   // 将这个数插入
32.         }
33.         printf("%lld\n", ans);
34.     }
35.     return 0;
36. }
37.
```

4.1.4 质因数分解

通过 4.1.1.4 我们了解了算数基本定理，那么通过该节我们将学习质因数分解的两个算法。质因数分解的 Pollard-Rho 算法将在第八章的数学提高篇学习。

首先考虑朴素的算法，从 2 开始，如果含有该质因子则一直除到不含为止，这样相当于含 2 因子的合数都不会再被整除。每次可以保证整除的数一定是这个数的质因数时间复杂度 $O(\sqrt{n})$。

```
int fac[maxn], num[maxn];
int cnt;

void solve(int n) {
    int m = sqrt(n + 0.5);
    memset(num, 0, sizeof num);
    for (int i = 2; i <= m && n != 1; i++) {
        if (n % i == 0) {
            fac[cnt] = i;
            while (n % i == 0) {
                num[cnt]++;
                n /= i;
            }
            cnt++;
        }
    }
    if (n > 1) {   // 最后特判大于根号 n 的素数
        fac[cnt] = n;
        num[cnt++]++;
    }
}
```

根据线性筛的最小质因子思想有一个线性对 $[1,n]$ 内的数快速质因数分解的算法。首先线性筛预处理出所有质因子以及每个数的最小质因子，然后在循环中每次只需除掉最小的质因子，这样不断除直到 n 变成 1 就能得到可重复且乱序的质因子数组，但是相同的数总是连续的，那么我们排序后线性去重即可。时间复杂度 $O(n)$。

```
vector<int> prime;
int minp[maxn];

void euler() {
    for (int i = 1; i < maxn; i++) minp[i] = i;
    for (int i = 2; i < maxn; i++) {
        if (minp[i] == i) prime.push_back(i);
        for (int j = 0; j < prime.size() && 1LL * i * prime[j] < maxn; j++) {
            minp[i * prime[j]] = prime[j];
```

```
            if (i % prime[j] == 0) break;
        }
    }
}

void divide(int n) {
    if (n == 1) return;
    int p[1005], num[1005], cnt = 0, k = 1;
    while (n > 1) {
        p[++cnt] = minp[n];
        n /= minp[n];
    }
    sort(p + 1, p + cnt + 1);
    num[1] = 1;
    for (int i = 2; i <= cnt; i++) {
        if (p[i] == p[i - 1])
            num[k]++;
        else {
            num[++k] = 1;
            p[k] = p[i];
        }
    }
    for (int i = 1; i <= k; i++) {
        printf("%d^%d ", p[i], num[i]);
    }
}
```


阶乘分解（AcWing 197）

给定整数 N，试把阶乘 $N!$ 分解质因数，按照算术基本定理的形式输出分解结果中的 p_i和c_i即可。

【输入格式】

输入一个正整数 $N(1 \le N \le 10^6)$

【输入格式】

$N!$质因数分解的结果。共若干行，每行两个正整数 p_i和c_i。

【分析】

若把 $[1,N]$ 每个数分别分解质因数，再把结果合并，时间复杂度过高，为 $O(N\sqrt{N})$。显然，$N!$的每个质因子都不会超过N，我们可以先筛选出 $[1,N]$ 的每个质数 p，然后考虑阶乘 $N!$ 中一共包含多少个质因子 p。

$N!$ 中质因子 p 的个数就等于 $[1,N]$ 每个数包含质因子 p 的个数之和。在 $[1,N]$中，p 的倍数，即至少包含 1 个质因子 p 的显然有 $\lfloor \frac{N}{p} \rfloor$ 个。而 p^2 的倍数，即至少包含 2 个质因子

p 的有$\lfloor\frac{N}{p^2}\rfloor$个。不过其中的 1 个质因子已经在 $\lfloor\frac{N}{p}\rfloor$ 里统计过，所以只需要再统计第 2 个质因子，即累加上 $\lfloor\frac{N}{p^2}\rfloor$，而不是 $2*\lfloor\frac{N}{p^2}\rfloor$。

综上所述，$N!$ 中质因子 p 的个数为：

$$\lfloor\frac{N}{p}\rfloor+\lfloor\frac{N}{p^2}\rfloor+\lfloor\frac{N}{p^3}\rfloor+\ldots+\lfloor\frac{N}{p^{\lfloor log_p^n\rfloor}}\rfloor=\sum_{p^k\le N}\lfloor\frac{N}{p^k}\rfloor$$

对于每个 p，只需要 $O(logN)$ 的时间计算上述和式。故对整个 $N!$ 分解质因数的时间复杂度为 $O(NlogN)$ 。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 10;
long long p[N], vis[N], n, m;

long long init(long long n)   // 素数筛
{
    for (long long i = 2; i <= n; i++) {
        if (!vis[i]) p[++p[0]] = i;
        for (long long j = 2; j <= n / i; j++) vis[i * j] = 1;
    }
}
long long power(long long a, long long b)   // 快速幂
{
    long long ans = 1;
    while (b) {
        if (b & 1) ans = ans * a;
        a *= a;
        b >>= 1;
    }
    return ans;
}
int main() {
    ios::sync_with_stdio(false);
    cin >> n;
    init(n);
    for (long long i = 2; i <= n; i++)
```

```
        if (!vis[i]) {
            long long ans = 0;
            cout << i << " ";
            for (long long k = 1; power(i, k) <= n; k++) {
                ans = ans + n / power(i, k);
            }
            cout << ans << endl;
        }
    return 0;
}

```

Prime Game（2018 ICPC 南京区域赛 J）

给出一个n个数的序列，定义$mul(l,r)=\prod_{i=l}^{r}a_i$，$fac(l,r)$为$mul(l,r)$内不同的质因子个数，求$\sum_{i=1}^{n}\sum_{j=i}^{n}fac(i,j)$。

【输入格式】

第一行输入一个正整数 $n(1\le n\le 10^6)$代表序列的长度；

第二行输入序列的n个正整数$a_i(1\le a_i\le 10^6)$。

【输入格式】

输出上述方程的结果。

【分析】

对于这样需要操作每个区间的问题，常见的思路是固定左端点或者固定右端点，然后不断移动另一端观察区间需要维护问题的变化有无规律，一般来说常见的是有单调性。但是对于本题来说，这样的思路是行不通的。

那么我们考虑另外的思路，首先不难想到对于一段长度为n的序列，其所有的子区间的个数为$\frac{n(n+1)}{2}$。然后需要考虑最重要的一点是，每个质因数对一个区间的贡献要么为1要么为0，我们首先假设所有的质因数对所有区间都有贡献，显然多加了一部分。就是从前向后考虑每种质因数的相邻位置，二者中间的序列是不含有该质因子的，于是这部分是多算的。那么我们只需要遍历每种质因数然后减去多的贡献。

因为在数据范围内每种数的质因数种类数极少，然后我们枚举出现的所有质因数，计算答案即可。

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

using namespace std;
typedef long long ll;
```

```cpp
const int maxn = 1e6 + 10;

int prime[maxn], minp[maxn], all[maxn];
vector<int> fac[maxn], pos[maxn];
bitset<maxn> vis;
int cnt, tot;

void euler() {
    for (int i = 1; i <= maxn; i++) minp[i] = i;
    for (int i = 2; i < maxn; i++) {
        if (minp[i] == i) {
            prime[++cnt] = i;
            pos[i].push_back(0);
        }
        for (int j = 1; j <= cnt && 1LL * i * prime[j] < maxn; j++) {
            minp[i * prime[j]] = prime[j];
            if (i % prime[j] == 0) break;
        }
    }
}

void divide(int idx, int n) {
    tot = 0;
    while (n != 1) {
        all[tot++] = minp[n];
        n /= minp[n];
    }
    sort(all, all + tot);
    int m = unique(all, all + tot) - all;
    for (int i = 0; i < m; i++) {
        pos[all[i]].push_back(idx);
        vis[all[i]] = 1;
    }
}

```

```cpp
inline ll cal(ll x) {
    return x * (x + 1) / 2;
}

bool ok = 0;

ll solve(int p) {
    ll ans = 0;
    for (int i = 1, l, r; i < pos[p].size(); i++) {
        l = pos[p][i - 1] + 1, r = pos[p][i] - 1;
        if (l <= r) ans += cal(r - l + 1);
    }
    return ans;
}

int main() {
    ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int n;
    euler();
    vis.reset();
    cin >> n;
    for (int i = 1, x; i <= n; i++) {
        cin >> x;
        divide(i, x);
    }
    int num = vis.count();
    ll ans = cal(n) * num;
    for (int i = 1; i <= cnt; i++) {
        if (vis[prime[i]]) {
            pos[prime[i]].push_back(n + 1);
            ll res = solve(prime[i]);
            ans -= res;
        }
    }
    cout << ans << "\n";
```

```
    return 0;
}

```

4.1.5 快速幂

快速幂，二进制取幂（也称平方法），是一个在 $O(\log n)$ 的时间内计算 a^n 的常用算法，而暴力的计算需要 $O(n)$ 的时间。

计算 a 的 n 次方表示将 n 个 a 乘在一起。然而当 a,n 太大的时侯，这种方法就不太适用了。不过我们知道：$a^{b+c} = a^b * a^c, a^{2b} = a^b * a^b = (a^b)^2$。二进制取幂的想法是，我们将取幂的任务按照指数的二进制表示来分割成更小的任务。

首先我们将 n 表示为 2 进制，举一个例子：$3^{13} = 3^{(1101)_2} = 3^8 * 3^4 * 3^1$。因为 n 有 $\lfloor \log_2 n \rfloor + 1$ 个二进制位，因此当我们知道了 $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log_2 n \rfloor}}$ 后，我们只用计算 $O(\log n)$ 次乘法就可以计算出 a^n。于是我们只需要知道一个快速的方法来计算上述 3 的 2^k 次幂的序列。这个问题很简单，因为序列中（除第一个）任意一个元素就是其前一个元素的平方。举一个例子：$3^1, 3^2 = (3^1)^2, 3^4 = (3^2)^2 \ldots$于是为了计算 $3^{13} = 3^8 * 3^4 * 3^1$，我们只需要将对应二进制位为 1 的整系数幂乘起来就行了。

因为幂函数的曲线是爆炸增长的过程，因此快速幂常常需要取模运算，或者称之为快速幂取模。

```
typedef long long ll;

ll qkp(ll x, ll n, ll p) {
      ll ans = 1;
    x %= p;
      while(n) {
                if (n & 1) ans = ans * x % p;
                x = x * x % p;
        n >>= 1;
      }
      return ans;
}
```

64 位整数乘法（AcWing 90）

求 a乘 b对 p取模的值。

【输入格式】

第一行输入正整数 a。

第二行输入正整数 b。

第三行输入正整数 p。

$1 \le a, b, p \le 10^6$

【输入格式】

输出一个正整数表示 $a * b \% p$的值。

【分析】

如果直接计算$a * b$这会超过 long long 的最大范围。所以快速幂的乘法要使用慢速乘，并且每次计算后取模就可以了。当然也可以使用下面的黑科技快速乘。

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

typedef long long ll;

inline ll mul(ll a, ll b, ll p) {
    if (p <= 1000000000)
        return a * b % p;
    else if (p <= 1000000000000LL)
        return (((a * (b >> 20) % p) << 20) + (a * (b & ((1 << 20) - 1)))) % p;
    else {
        ll d = (ll)floor(a * (long double)b / p + 0.5);
        ll ret = (a * b - d * p) % p;
        if (ret < 0) ret += p;
        return ret;
    }
}

int main() {
    ll x, y, p;
    cin >> x >> y >> p;
    cout << mul(x, y, p) << endl;
    return 0;
}

```


但是 $O(\log_2 n)$ 的“慢速乘”还是太慢了，这在很多对常数要求比较高的算法比如 Miller_Rabin 和 Pollard-Rho 中，就显得不够用了。所以我们要介绍一种可以处理模数在 long long 范围内、不需要使用黑科技 __int128 的、复杂度为 $O(1)$ 的“快速乘”。

```
long long mul(long long a, long long b, long long m) {
```

```
    unsigned long long c =
        (unsigned long long)a * b - (unsigned long long)((long double)a / m * b +
0.5L) * m;
    if (c < m) return c;
    return c + m;
}
```

4.1.6 扩展欧几里得

4.1.6.1 裴蜀定理

裴蜀定理，又称贝祖定理。是一个关于最大公约数的定理。定理内容是：设 a,b 是不全为零的整数，则存在整数 x, y, 使得 $ax + by = \gcd(a, b)$。

裴蜀定理的两个重要推论为：

1. a 和 b 互质的充要条件是存在整数 x, y，有 $ax + by = 1$。
2. $ax + by = c$ 有解的充要条件为 $\gcd(a, b) | c$。

裴蜀定理可以扩展到 n 个数的情况，即设 $a_1, a_2, \dots, a_n$是不全为 0 的整数，则存在整数 $x_1, x_2, \dots, x_n$使得 $a_1x_1 + a_2x_2 + \cdots + a_nx_n = \gcd(a_1, a_2, \dots, a_n)$。

裴蜀定理的证明要从辗转相除法的带余式方程组中入手。设$d = \gcd(a, b)$，那么原式 $ax + by = \mathrm{d}$ 方程两边同除以 d，得到 $a'x + b'y = 1$，只需要去证明这个式子有解，令 $r_0 = a, r_1 = b$，那么有：

$$r_0 = x_1r_1 + r_2, 0 < r_2 < r_1$$
$$r_1 = x_2r_2 + r_3, 0 < r_3 < r_2$$
$$\dots\dots$$
$$r_{n-3} = x_{n-2}r_{n-2} + r_{n-1}, 0 < r_{n-1} < r_{n-2}$$
$$r_{n-2} = x_{n-1}r_{n-1} + r_n, 0 < r_n < r_{n-1}$$
$$r_{n-1} = x_nr_n + r_{n+1}, \qquad r_{n+1} = 0$$

令辗转相除法在除到互质时结束，则有 $r_n = 1$，代入上一个式子可以得到 $r_{n-2} = x_{n-1}r_{n-1} + 1$，即 $1 = r_{n-2} - x_{n-1}r_{n-1}$。将倒数第三个式子 $r_{n-1} = r_{n-3} - x_{n-2}r_{n-2}$ 代入其中，得到：$1 = (1 + x_{n-1}x_{n-2})r_{n-2} - x_{n-1}r_{n-3}$。

用同样的方法逐层消去 $r_{n-2}, r_{n-3}, \dots, r_2$，可以证明出存在$x, y$，使得 $a'x + b'y = 1$，于是原式得证。

4.1.6.2 扩展欧几里得

扩展欧几里得算法又称 exgcd，就是利用欧几里得算法，求出贝祖定理 $ax + by =$

$\gcd(a,b)$的整数解。寻找一组解的过程如下：

由$gcd(a,b)=gcd(b,a\%b)$，可得$gcd(b,a\%b)=bx+(a\%b)y$；

令$a_1=b,b_1=a\%b$，那么：

$$a_1x_1+b_1y_1=gcd(a_1,b_1)=gcd(b_1,a_1\%b_1)=b_1x_2+(a_1\%b_1)y_2;$$

再令 $a_2=b_1,b_2=a_1\%b_1$，那么：

$$a_2x_2+b_2y_2=gcd(a_2,b_2)=gcd(b_2,a_2\%b_2)=b_2x_3+(a_2\%b_2)y_3;$$

同理以此这样向下推导……最后可化简为 $a_nx_n+0*y_n=gcd(a_n,0)=a_n$，此时得到最后一组解为 $x_n=1,y_n=0$。

因为$a\%b=a-\left\lfloor\frac{a}{b}\right\rfloor*b$，又因为第一次我们化简的等式中有:$a_1x_1+b_1y_1=b_1x_2+(a_1\%b_1)y_2$，所以：

$$a_1x_1+b_1y_1=b_1x_2+(a_1-\left\lfloor\frac{a_1}{b_1}\right\rfloor*b_1)y_2=a_1y_2+b_1(x_2-\left\lfloor\frac{a_1}{b_1}\right\rfloor*y_2)$$

所以 $x_1=y_2,y_1=x_2-a_1/b_1*y_2$，根据此推导式可以从最后一组解回溯，求得$x_1,y_1$。

下面的代码中是利用 C++引用优化的版本。我们都是由下一层回溯得到上一层的，当由第 i 层返回到第 $i-1$ 层时，我们把 y_i 传给 x_{i-1}，这样由上面公式来看是正确结果；而 y_{i-1} 怎么求呢？ $y_{i-1}=x_i-(a_{i-1}/b_{i-1})*y_i$。而 y_i 已经赋值给了 x_{i-1}，x_i 已经赋值给了 y_{i-1}，因此 $y_{i-1}-=(a_{i-1}/b_{i-1})*x_{i-1}$ 就可以得到 y_{i-1}，再一层层向上传递即可。

```
1. ll exgcd(ll a, ll b, ll &x, ll &y) {
2.     if (!b) {
3.         x = 1, y = 0;
4.         return a;
5.     }
6.     ll gcd = exgcd(b, a % b, y, x);
7.     y -= (a / b) * x;
8.     return gcd;
9. }
```


扩展欧几里得的常见应用有：求解二元一次不定方程，求乘法逆元，求解线性同余方程等。其中乘法逆元将在 4.1.7 章节介绍。

1. 求解二元一次不定方程

已知 $ax+by=c$，设 $r=\gcd(a,b)$，则：

若 $c\%r\neq 0$，该方程组无整数解

若 $c\%r=0$，我们可以先求出 $ax+by=r$ 对应的一组整数解 (x_1,y_1)，然后再乘以 c 并除以 r 即可得到 $ax+by=c$ 的最小整数解 $x_0=x_1*c/r,y_0=y_1*c/r$，那么其通解为 $x=x_0+k*(b/r),y=y_0-k*(a/r)$，$k$ 取任意整数。

上述最小整数解的含义是 x,y 其中一个为最小正整数或者二者均是最小正整数；若 x 当前的系数为负数，那么假设求解 $a(-x)+by=c$，将得到的整数解取负即可。

2. 求解线性同余方程

已知线性同余方程 $ax \equiv b \pmod p$,则该同余方程可以化为求解二元一次不定方程 $ax + py = b$。设 $r = \gcd(a,p)$，显然当且仅当 b%r==0 时方程组有解。

我们先求 $ax + py = r$ 的解 x_1，接着可以得出 $ax \equiv b \pmod p$ 的一组解为 $x_0 = x_1 * b/r$，设解之间的最小间隔为 $s = p/r$，且可以证明该方程组一共有 r 个解。

因此 $ax \equiv b \pmod p$ 的最小整数解为 $(x_0\%s + s)\%s, ax \equiv b \pmod p$ 的通解为 $x = x_0 + k * \left(\frac{p}{r}\right), k \in [0, r-1]$。

```
1. int gcd(int a, int b) {
2.     return b == 0 ? a : gcd(b, a % b);
3. }
4.
5. int exgcd(int a, int b, int &x, int &y) {
6.     if (!b) {
7.         x = 1, y = 0;
8.         return a;
9.     }
10.    int gcd = exgcd(b, a % b, y, x);
11.    y -= (a / b) * x;
12.    return gcd;
13. }
14.
15. int solve(int a, int b, int p) {
16.    if (b % gcd(a, p)) return -1;
17.    int x, y, r, x0;
18.    r = exgcd(a, p, x, y);
19.    int s = p / r;
20.    x = x * b / r;
21.    x0 = (x % s + s) % s;    // 最小整数解
22.    return x0;
23. }
24.
```


【模板】二元一次不定方程 (exgcd)（洛谷 P5656）

给定不定方程ax + by = c。

若该方程无整数解，输出 -1。

若该方程有整数解，且有正整数解，则输出其正整数解的数量，所有正整数解中 x 的最小值，所有正整数解中 y 的最小值，所有正整数解中 x 的最大值，以及所有正整数解中 y 的最大值。

若方程有整数解，但没有正整数解，你需要输出所有整数解中 x 的最小正整数值， y 的最小正整数值。

正整数解即为 x, y 均为正整数的解，0 不是正整数。

整数解即为 x,y 均为整数的解。

x 的最小正整数值即所有 x 为正整数的整数解中 x 的最小值，y 同理。

【输入格式】

第一行输入正整数 $T(1 \le T \le 2 \times 10^5)$ 代表数据组数。

接下来 T 行，每行三个由空格隔开的正整数$a,b,c(1 \le a,b,c \le 10^9)$。

【输入格式】

输出共 T 行。

若该行对应的询问无整数解，一个数字 -1。

若该行对应的询问有整数解但无正整数解，包含 2 个由空格隔开的数字，依次代表整数解中，x 的最小正整数值，y 的最小正整数值。

否则包含 5 个由空格隔开的数字，依次代表正整数解的数量，正整数解中，x 的最小值，y 的最小值，x 的最大值，y 的最大值。

【分析】

本题需要知道exgcd求出的最小整数解会有哪几种情况，因为a,b,c均为正整数：

- $x > 0$，$y \le 0$
- $x \le 0$，$y > 0$
- $x > 0$，$y > 0$

先给出扩欧求二元一次不定方程的解释：已知$ax + by = c$，设$r = gcd(a,b)$，则

- 若$c\%r! = 0$，该方程组无整数解
- 若$c\%r = 0$，我们可以先求出$ax + by = r$对应的一组整数解(x_1,y_1)，然后再乘以c并除以r即可得到$ax + by = c$的最小整数解$x_0 = x_1 * c/r, y_0 = y_1 * c/r$，那么其通解为$x = x_0 + k * (b/r), y = y_0 - k * (a/r)$，k取任意整数；

上述最小整数解的含义是x,y其中一个为最小正整数或者二者均是最小正整数。因此实际上通解的x_0,y_0必须是一个增大一个减小，一起变化的，因此不难想到分论讨论寻找范围

对于上述第一种情况，要知道是否有正整数解，就是看x最多减去多少次仍大于 0，并且计算y最少加上多少次能大于 0，对于得到这两个次数，分$dx \ge dy$和dx < dy两种情况讨论，前者肯定有多个正整数解，后者不存在正整数解，然后就是细节计算了。第二种情况同理。至于第三种情况，是一定至少存在一组正整数解的，那么仍然按照上面的思想计算即可。

```
1. #include <bits/stdc++.h>
2.
3. using namespace std;
4. typedef long long ll;
5. const int maxn = 2e5 + 10;
6.
```

```
ll gcd(ll a, ll b) {
    return !b ? a : gcd(b, a % b);
}

void exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return;
    }
    exgcd(b, a % b, y, x);
    y -= (a / b) * x;
}

int main() {
    ll a, b, c, x1, y1, mi_x, mi_y, ma_x, ma_y, cnt;
    int t;
    scanf("%d", &t);
    while (t--) {
        scanf("%lld%lld%lld", &a, &b, &c);
        ll g = gcd(a, b);
        if (c % g != 0) {
            puts("-1");
        } else {
            exgcd(a, b, x1, y1);
            ll x = x1 * c / g, y = y1 * c / g;
            a /= g, b /= g;
            if (x > 0 && y <= 0) {
                ll dx = x / b - (x % b == 0), dy = abs(y) / a + 1;
                if (dx >= dy) {
                    cnt = dx - dy + 1;
                    mi_x = x - dx * b, mi_y = y + dy * a;
                    ma_x = x - dy * b, ma_y = y + dx * a;
                    printf("%lld %lld %lld %lld %lld\n", cnt, mi_x, mi_y, ma_x,
ma_y);
                } else {
```

```
41.                     mi_x = x - dx * b, mi_y = y + dy * a;
42.                     printf("%lld %lld\n", mi_x, mi_y);
43.                 }
44.             } else if (x <= 0 && y > 0) {
45.                 ll dx = abs(x) / b + 1, dy = y / a - (y % a == 0);
46.                 if (dy >= dx) {
47.                     cnt = dy - dx + 1;
48.                     mi_x = x + dx * b, mi_y = y - dy * a;
49.                     ma_x = x + dy * b, ma_y = y - dx * a;
50.                     printf("%lld %lld %lld %lld %lld\n", cnt, mi_x, mi_y, ma_x,
ma_y);
51.                 } else {
52.                     mi_x = x + dx * b, mi_y = y - dy * a;
53.                     printf("%lld %lld\n", mi_x, mi_y);
54.                 }
55.             } else {
56.                 ll dx = x / b - (x % b == 0), dy = y / a - (y % a == 0);
57.                 cnt = dx + dy;
58.                 mi_x = x - dx * b, mi_y = y - dy * a;
59.                 ma_x = x + dy * b, ma_y = y + dx * a;
60.                 printf("%lld %lld %lld %lld %lld\n", cnt, mi_x, mi_y, ma_x,
ma_y);
61.             }
62.         }
63.     }
64.
65.     return 0;
66. }
67.
```


4.1.7 乘法逆元

在取模运算的除法取模那里，我们说取模时除法并不能直接对除数取模。若所有整数 $\mathbb{Z}$ 对数 n 取模，那么构成一个模 n 的剩余类环。如果环中所有数对模 n 都存在逆元，那么构成一个数域。以前我们知道除以一个整数等于乘以它的倒数，这个倒数在模意义下被叫

做乘法逆元（以下简称为逆元）。剩余类环中的数和逆元相乘得到的是模意义下的单位元。

若 $ax \equiv 1 \pmod{p}$，且 $gcd(a,p) = 1$，那么我们就能定义：x 为 a 模 p 意义下的逆元，记为 a^{-1} 或 $\text{inv}(a)$。

4.1.7.1 扩展欧几里得求逆元

此方法只要满足 a 和 p 互质即可使用。

根据逆元的定义可得 $ax \equiv 1 \pmod{p}$，我们可以转化为求解 $ax + py = 1$，再根据扩展欧几里得算法即可求解 x。

由于最后的求得的 x 可能是负数，因此我们再进行处理：$(x\%p + p)\%p$。

```
1. ll exgcd(ll a, ll b, ll &x, ll &y) {
2.     if (!b) {
3.         x = 1, y = 0;
4.         return a;
5.     }
6.     ll gcd = exgcd(b, a % b, y, x);
7.     y -= (a / b) * x;
8.     return gcd;
9. }
10.
11. ll inv(ll a, ll p) {
12.     ll x, y;
13.     exgcd(a, p, x, y);
14.     return (x % p + p) % p;
15. }
```


4.1.7.2 快速幂求逆元

费马小定理：若 p 为素数，a 为正整数，且 $(a,p) = 1$。则有 $a^{p-1} \equiv 1 \pmod{p}$。

由 $ax \equiv 1 \pmod{p}$，再根据费马小定理得 $ax \equiv a^p - 1 \pmod{p}$，所以 $x \equiv a^{p-2} \pmod{p}$。这样我们直接使用快速幂取模就可以求出逆元。

```
1. ll qkp(ll x, ll n, ll p) {
2.       ll ans = 1;
3.     x %= p;
4.     while (n) {
5.         if (n & 1) ans = ans * x % p;
6.         x = x * x % p;
7.         n >>= 1;
8.     }
9.       return ans;
```

```
}

ll inv(ll a, ll p) {
    return qkp(a, p - 2, p);
}
```

4.1.7.3 特殊情况求逆元

若求逆元 $a^{-1}(mod\ p)$ 且 $a,p \neq 1$，那么费马小定理和扩欧均不再适用，此时需要引入以下定理：

$$\frac{a}{b}(\mathrm{mod}\ p) = a(\mathrm{mod}\ \ b * p)/b$$

证明：设 $\frac{a}{b} = kp + x(x < p)$，那么有 $a = k * b * p + bx$。即 $a(\mathrm{mod}\ b * p) = bx$。所以 $a(\mathrm{mod}\ b * p)/b = x$，原式得证。

4.1.7.4 线性求逆元

1. 求 $1 \sim n$ 在模 p 下的所有乘法逆元（$n < p$ 且 p 为质数）

   如果一个个求，显然时间复杂度有点高，因此我们试着找下递推的规律：

   令 $x = p/i\ , y = p\%i$, 则有：$x * i + y = p$;

   然后 $x * i + y \equiv 0\ (\mathrm{mod}\ p)$，$y \equiv -x * i\ (\mathrm{mod}\ p)$;

   同余式两边同乘以 $\mathrm{inv}(y) * \mathrm{inv}(i)$，得 $[y * \mathrm{inv}(y)] * \mathrm{inv}(i) \equiv -x * y * [i * \mathrm{inv}(i)]\ (\mathrm{mod}\ p)$;

   一个数和它的逆元相乘为单位元 1，则 $\mathrm{inv}[i] \equiv -x * \mathrm{inv}[y]\ (\mathrm{mod}\ p)$;

   最后将 $x = p/i, y = p\%i$ 带入就可以得到: $\mathrm{inv}[i] \equiv (p - p/i) * \mathrm{inv}[p\%i]\ (\mathrm{mod}\ p)$。

   上式中 $p - p/i$ 是因为负数在取模意义下相当于加上 p 再取模。

```
ll inv[maxn];

void solve(int n, ll p) {
    inv[1] = 1;
    for(int i = 2; i <= n; i++) {
        inv[i] = (p - p / i) * inv[p % i] % p;
    }
}
```

2. 求阶乘$1! \sim n!$ 的在模 p 下的所有逆元。

   逆元就可以看做是在模意义下的倒数，那么 $\mathrm{inv}[n!] = \mathrm{inv}[(n+1)!] * (n+1)$，故我们先由快速幂取模求出 $\mathrm{inv}[n!]$ 再从后向前递推求出所有的逆元。

```cpp
ll fac[maxn], inv[maxn];

ll qkp(ll x, ll n, ll p) {
    ll ans = 1;
    x %= p;
    while (n) {
        if (n & 1) ans = ans * x % p;
        x = x * x % p;
        n >>= 1;
    }
    return ans;
}

// 求 1!-n!的所有逆元
void solve(int n, ll p) {
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = fac[i - 1] * i % p;
    }
    inv[n] = qkp(fac[n], p - 2, p);
    for (int i = n - 1; i >= 0; i--) {
        inv[i] = inv[i + 1] * (i + 1) % p;
    }
}
```


4.1.8 数论函数

4.1.8.1 积性函数

定义：积性函数指对于所有互质的整数 a 和 b 有性质 $f(ab)=f(a)f(b), a \perp b$ 的数论函数。若积性函数 $f(x)$ 满足任意的整数 a,b 有 $f(ab)=f(a)f(b)$，又称为完全积性函数

性质：

对正整数 n 质因数分解得到 $n=p_1^{a_1}*p_2^{a_2}*\ldots*p_n^{a_n}$，积性函数满足 $f(n)=\prod f\left(p_i^{k_i}\right)=f\left(p_1^{a_1}\right)f\left(p_2^{a_2}\right)\ldots f\left(p_n^{a_n}\right)$，完全积性函数满足 $f(n)=\prod f(p_i)^{k_i}=f(p_1)^{a_1}f(p_2)^{a_2}\ldots f(p_n)^{a_n}$。

若 $f(x),g(x)$ 均为积性函数，那么以下函数也为积性函数：

$$h(x)=f(x^p)$$

$$h(x)=f^p(x)$$

$$h(x)=f(x)g(x)$$

$$h(x)=\sum_{d|x} f(x)g\left(\frac{x}{d}\right)$$

常见积性函数：

- $\varphi(n)$：欧拉函数，小于 n 且与 n 互质的数的个数；
- $gcd(x,n)$：欧几里得函数，当 n 固定时才为积性函数；
- $\mu(n)$：莫比乌斯函数，关于非平方数的质因子数目；
- $\varepsilon(n)=[n=1]$：单位函数（完全积性）；
- $id_k(n)=n^k$：幂函数（完全积性）；
  - 当 $n=1$ 时为恒等函数 $id(n)$；
- $\sigma_k(n)=\sum_{d|n} d^k$：除数函数；
  - 当 $k=1$ 时 $\sigma(n)=\sum_{d|n} d$，即约数和函数；
  - 当 $k=0$ 时 $\sigma_0(n)=\sum_{d|n} 1$，即约数个数函数；
- $1(n)$：常数函数（完全积性）；
- $d(n)=\sum_{d|n} 1$：约数个数函数，等价于 $\sigma_0(n)$

4.1.8.2 欧拉函数

欧拉函数就是小于 n 且与 n 互质的整数个数。

设 n 的唯一分解式为 $n=p_1^{a_1}p_2^{a_2}\dots p_n^{a_n}$

$$\varphi(n)=n\frac{(p_1-1)}{p_1}\frac{(p_2-1)}{p_2}\dots\frac{(p_n-1)}{p_n}=n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\dots\left(1-\frac{1}{p_n}\right)$$

联系到积性函数，不难发现欧拉函数是积性函数。

那么如何求欧拉函数？

1. **求单个数的欧拉函数**

```
ll euler_phi(ll n) {
    int m = sqrt(n + 0.5);
    ll ans = n;
    for (int i = 2; i <= m; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}
```

2. **埃氏筛法求欧拉函数**

根据欧拉函数的定义，利用埃氏筛，每求得一个素数就更新其倍数所有的 phi 值。

```
int phi[maxn];

```

```
void phi_table(int n) {
    memset(phi, 0, sizeof phi);
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
        if (!phi[i]) {
            for (int j = i; j <= n; j += i) {
                if (!phi[j]) phi[j] = j;
                phi[j] = phi[j] / i * (i - 1);
            }
        }
}
```

3. **欧拉筛法求欧拉函数**

首先要了解欧拉函数的以下性质：

对于素数 p：

- 若 $n\%p == 0$，那么 $\varphi(p * n) = p * \varphi(n)$；
- 若 $n\%p! = 0$，那么 $\varphi(p * n) = \varphi(p) * \varphi(n) = (p-1) * \varphi(n)$；
- 若 n 为某一素数的幂次，那么 $\varphi(p^a) = (p-1) * p^{a-1}$；

欧拉筛求欧拉函数的证明：注意到线性筛的时候，每一个合数都是被最小的质因子筛掉。比如设 p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，那么线性筛的时候 n 通过 $n' \times p_1$ 筛掉。观察线性筛的过程，对 $n'\ mod\ p_1$ 分情况讨论：

- 若 $n'\ mod\ p_1 = 0$，那么 n' 包含了 n 的所有质因子，$\varphi(n) = n \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = p_1 \times n' \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = p_1 \times \varphi(n')$；
- 若$n'\ mod\ p_1 \neq 0$，这时 n' 和 p_1 是互质的，根据欧拉函数的性质，$\varphi(n) = \varphi(p_1) \times \varphi(n') = (p_1 - 1) \times \varphi(n')$。

```
vector<int> prime;
bitset<maxn> vis;
int phi[maxn];

void euler() {
    phi[1] = 1;
    for (int i = 2; i < maxn; i++) {
        if (!vis[i]) {
            phi[i] = i - 1;
            prime.push_back(i);
        }
        for (int j = 0; j < prime.size() && i * prime[j] < maxn; j++) {
            vis[i * prime[j]] = 1;
            if (i % prime[j]) {
```

```
                phi[i * prime[j]] = phi[i] * phi[prime[j]];
            } else {
                phi[i * prime[j]] = phi[i] * prime[j];
                break;
            }
        }
    }
}
```

欧拉函数的一些性质：

1. 当 m,n 互质时，有 $\varphi(m*n)=\varphi(m)*\varphi(n)$。
2. 对于 $n>2$，$\varphi(n)$ 一定为偶数。
3. 对于素数 p：
   a) 若 $n\%p==0$，那么 $\varphi(pn)=p*\varphi(n)$。
   b) 若 $n\%p\neq 0$，那么 $\varphi(pn)=\varphi(p)*\varphi(n)=(p-1)*\varphi(n)$。
4. 对于互质的 x 与 p，有 $x^{\varphi(p)}\equiv 1(mod\ p)$，因此 x 的逆元为 $x^{\varphi(p)}$，即欧拉定理。特别地，当 p 为质数时，$\varphi(p)=p-1$，此时逆元为 x^{p-1}，即费马小定理 $x^{p-1}\equiv 1(mod\ \ p)$。
5. 当 n 为奇数时，$\varphi(2*n)=\varphi(n)$。
   证明：n 为奇数那么 $gcd(2,n)=1$，所以 $\varphi(2*n)=\varphi(2)*\varphi(n)=\varphi(n)$。
6. 设 p 为素数，$\varphi(p^a)=p^a-p^{a-1}$。
   证明：根据容斥定理，使用 p^a 减去与它不互质的整数个数即可，小于 p^a 且不和 p^a 互质的数的个数就是范围内 p 的倍数的个数，有 $[\frac{p^a}{p}]=p^{a-1}$ 个，那么定理得证。
   根据这个定理，欧拉函数也能通过质因数分解求解。
7. $[m+1,2m],[2m+1,3m],\ldots,[(n-1)m+1,nm]$ 这些区间内和 m 互质的个数均为 $\varphi(m)$ 个。
   证明根据此定理：若 $a\perp b$，那么 $(a+b*k)\perp b$。
8. 由上述性质六可得：若 n 是 m 的倍数，那么 $[1,n]$ 中和 m 互质的数的个数为 $\frac{n}{m}\varphi(m)$。
9. 设 P_k 为第 k 个质数，对于 $P_i\leq x<P_{i+1}$，恒有 $phi[x]\leq phi[P_i]$。
10. 对于任意正整数 n，其所有的因子 d 的欧拉函数之和恰好为 n，即 $\sum_{d|n}\varphi(d)=n$。
11. 小于 n 且与 n 互质的数之和，即 $\sum_{i=1}^{n} i*[gcd(i,n)=1]=\begin{cases}1 & n\leq 2\\ \frac{\varphi(n)\times n}{2} & n>2\end{cases}$
    证明：
    当 $n\leq 2$ 时显然为 1。
    当 $n>2$ 时，$\varphi(n)$ 总为偶数，因为和 n 互质的数总是成对出现，假设 n 和 i 互质，那么 $n-i$ 和 i 也互质（根据更相减损的性质），他们的和为 n，并且有

$\varphi(n)$ 对，那么答案就是 $\frac{\varphi(n)\times n}{2}$。

嵌套欧拉函数：

称形如 $\varphi\left(\varphi\left(\varphi(\cdots(n))\right)\right)$ 的式子为嵌套欧拉函数。定理：若 $\varphi\left(\varphi\left(\varphi(\cdots(n))\right)\right)=1$，那么嵌套的层数为 $O(logn)$。

证明：当 n 为偶数时，$\varphi(n)\le\frac{n}{2}$；当 n 为奇数时，$\varphi(n)$ 为偶数。

[SDOI2008] 沙拉公主的困惑（洛谷 P2155）

大富翁国因为通货膨胀，以及假钞泛滥，政府决定推出一项新的政策：现有钞票编号范围为 1 到 N 的阶乘，但是，政府只发行编号与 M! 互质的钞票。房地产第一大户沙拉公主决定预测一下大富翁国现在所有真钞票的数量。现在，请你帮助沙拉公主解决这个问题，由于数量可能非常大，你只需计算出答案对 R 取模后的结果即可。

【输入格式】

第一行输入两个正整数 $T,R(1\le T\le 10^4,2\le R\le 10^9+10)$ 代表数据组数和模数，其中保证 R 为质数。

接下来 T 行，每行两个正整数$N,M(1\le M\le N\le 10^7)$。

【输入格式】

输出共 T 行，每行输出$[1,N!]$中与 M!互质的数的数量对 R 取模后的结果。

【分析】

首先根据欧拉函数的两个性质：

$[m+1,2m],[2m+1,3m],\ldots,[(n-1)m+1,nm]$这些区间内和$m$互质的个数均为$\varphi(m)$个。证明过程根据此定理：若$a\perp b$，那么$(a+b*k)\perp b$

由上述性质得：若n是m的倍数，那么$[1,n]$中和m互质的数的个数为$\frac{n}{m}\varphi(m)$

那么本题就转化为了求$\frac{N!}{M!}\varphi(M!)=N!\prod_{i=1}^{t}[(p_i-1)/p]$，其中$p_i$为$M!$中出现的所有质数，那么问题转化为预处理1e7以内的阶乘取模和质数，然后预处理1e7内的所有的前缀乘积$\prod_{i=1}^{t}[(p_i-1)/p]$

但是此题给出的取模数的范围是$[2,1e9+10]$的质数，那么需要知道在模R意义下求逆元前提是要两个数互质，但是如果恰好是$\frac{1}{R}(mod\ R)$，此时是没有意义的。注意到此时肯定有$N\ge M\ge R$，那么$N!$一定含有质因子R，那么可以将分母唯一的质因子R约掉，具体就是在预处理阶乘和逆元乘积时考虑到等于R的情况

但是这样交上去并不能通过所有测试用例，因为当$N\ge R\ \&\&\ M<R$时，因为无法消去分母的R的因子，因此按预处理得出的结果是不对的，因为答案一定是R的倍数，那么直接特判输出零即可。

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

```
2.
3. using namespace std;
4. typedef long long ll;
5. const int maxn = 1e7 + 10;
6.
7. vector<int> prime;
8. ll fac[maxn], inv[maxn], ans[maxn], Mod;
9. bitset<maxn> vis;
10.
11. void euler() {
12.     vis.reset();
13.     for (int i = 2; i < maxn; i++) {
14.         if (!vis[i]) prime.push_back(i);
15.         for (int j = 0; j < prime.size() && i * prime[j] < maxn; j++) {
16.             vis[i * prime[j]] = 1;
17.             if (i % prime[j] == 0) break;
18.         }
19.     }
20. }
21.
22. void init(ll p) {
23.     fac[0] = 1;
24.     for (int i = 1; i < maxn; i++) {
25.         if (i != p)
26.             fac[i] = (fac[i - 1] * i) % p;
27.         else
28.             fac[i] = fac[i - 1];
29.     }
30.     inv[1] = 1;
31.     for (int i = 2; i < maxn; i++) {
32.         inv[i] = (p - p / i) * inv[p % i] % p;
33.     }
34.     ans[1] = 1;
35.     for (int i = 2; i < maxn; i++) {
36.         if (vis[i])
```

```
37.             ans[i] = ans[i - 1];
38.         else {
39.             if (i != p)
40.                 ans[i] = (i - 1) * inv[i] % p * ans[i - 1] % p;
41.             else
42.                 ans[i] = (i - 1) * ans[i - 1] % p;
43.         }
44.     }
45. }
46.
47. int main() {
48.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
49.     int t, n, m;
50.     euler();
51.     cin >> t >> Mod;
52.     init(Mod);
53.     while (t--) {
54.         cin >> n >> m;
55.         if (n >= Mod && m < Mod) {
56.             cout << "0\n";
57.             continue;
58.         }
59.         cout << fac[n] * ans[m] % Mod << "\n";
60.     }
61.
62.     return 0;
63. }
64.
```


4.1.8.3 整除分块

1. 问题引入

求 $\sum_{i=1}^{k}\lfloor\frac{n}{i}\rfloor, n \leq 10^{14}$。

等价定义 1：设 $d(x)$ 为 x 的因子个数，则 $\sum_{i=1}^{n}\lfloor\frac{n}{i}\rfloor$ 等价于 $\sum_{i=1}^{n} d\,(i)$。

简单证明：$\sum_{i=1}^{n} d(i) = \sum_{i=1}^{n} \sum_{d|i} 1 = \sum_{d=1}^{n} \lfloor \frac{n}{d} \rfloor$，右边等式是因为当 i 取遍 $[1,n]$ 时 d 也取遍 $[1,n]$，改为枚举 d，那么问题变为对于每个 d 在 $[1,n]$ 有多少个它的倍数，即 $\lfloor \frac{n}{d} \rfloor$ 个，等式成立。

等价定义 2：设 x 的质因数分解式为 $x = p_1^{a_1} * p_2^{a_2} * \ldots * p_n^{a_n}$，令 $g(x) = (a_1 + 1) * (a_2 + 1) \ldots (a_n + 1)$，那么也等价于 $\sum_{i=1}^{n} g(i)$

2. 分析

显然枚举会超时，如果打表的话会发现有很多 $\lfloor \frac{n}{i} \rfloor$ 是一样的，那么需要分块计算，即将相等的一部分一次求出。而且打表的输出可以发现都是相同的 $\lfloor \frac{n}{i} \rfloor$ 会连续出现

设 $\lfloor \frac{n}{j} \rfloor$ 和 $\lfloor \frac{n}{i} \rfloor$ 相等，则 j 的最大值为 $\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \rfloor$。根据这个结论，只需要设置两个变量 l,r，每次令 $r = \lfloor \frac{n}{\lfloor \frac{n}{l} \rfloor} \rfloor$，一次计算出相同部分。

因为 n 以内的因数最多有 $2\sqrt{n}$ 种，那么时间复杂度为 $O(\sqrt{n})$，代码如下。

```
ll cal(ll n, ll k) {
    ll ans = 0;
    for (ll l = 1, r; l <= k; l = r + 1) {
        r = n / (n / l);
        if (r > k) r = k;
        ans += (n / l) * (r - l + 1);
    }
    return ans;
}
```

该公式也能应用到二维及以上的数论分块，例如求 $\sum_{i=1}^{min(n,m)} \lfloor \frac{n}{i} \rfloor \lfloor \frac{m}{i} \rfloor$。分析方式同理，代码如下：

```
ll cal(ll n, ll m) {
    ll ans = 0;
    ll up = min(n, m);
    for (ll l = 1, r; l <= up; l = r + 1) {
        r = min(n / (n / l), m / (m / l));
        if (r > up) r = up;
        //...
    }
    return ans;
}
```

3. 应用

数论分块可以快速计算一些含有除法向下取整的和式（即形如 $\sum_{i=1}^{n} f(i)g(\lfloor \frac{n}{i} \rfloor)$ 的和

式）。当可以在 $O(1)$ 内计算 $f(r)-f(l)$ 或已经预处理出 f 的前缀和时，数论分块就可以在 $O(\sqrt{n})$ 的时间内计算上述和式的值。

[CQOI2007]余数求和（洛谷 P2261）

给出正整数n,k，计算$\sum_{i=1}^{n} k \bmod i$。

【输入格式】

第一行输入两个正整数 $n,k(1 \le n,k \le 10^9)$。

【输入格式】

输出一行一个整数表示答案。

【分析】

首先知道$\sum_{i=1}^{n} k \bmod i$这个式子是不能暴力的。根据取模的定义，我们可以化简：

$$\sum_{i=1}^{n} k \bmod i = \sum_{i=1}^{n} k - i * \lfloor \frac{k}{i} \rfloor = \sum_{i=1}^{n} k - \sum_{i=1}^{n} i * \lfloor \frac{k}{i} \rfloor$$

看到$\sum_{i=1}^{n} \lfloor \frac{k}{i} \rfloor$很容易想到整除分块，但我们单纯的求出来的只是每一块的和，实际上每一块都是连续的，相当于是$i*(l+(l+1)+\ldots+r)$，然后用等差数列的公式求和即可：$(k/l)*(r-l+1)*(l+r)/2$

然后就是整除分块中需要注意的边界问题，因为n是有可能大于k的，因此如果到后面$l>k$时$k/l=0$，必须结束枚举，那么就是直接令$r=n$。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
const int maxn = 2e5 + 10;

ull n, k;

ull cal() {
    ull ans = 0;
    for (ull l = 1, r; l <= n; l = r + 1) {
        if (k / l)
            r = min(k / (k / l), n);
        else
            r = n;
        ans += (k / l) * (r - l + 1) * (l + r) / 2;
    }
```

```
18.     return ans;
19. }
20.
21. int main() {
22.     ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
23.     cin >> n >> k;
24.     cout << n * k - cal() << endl;
25.     return 0;
26. }
27.
```

4.1.9 欧拉定理及扩展

欧拉定理

欧拉定理内容：当 $gcd(a,n)=1$ 时，有 $a^{\varphi(n)} \equiv 1 \pmod{n}$

推论：当 $gcd(a,n)=1$ 时，有 $a^b \equiv a^{b\%\varphi(n)} \pmod{n}$

推论证明：$b = \varphi(n) * \lfloor \frac{b}{\varphi(n)} \rfloor + b\%\varphi(n)$

费马小定理

若 p 为素数，a 为正整数，且 a,p 互质。则有 $a^{p-1} \equiv 1 \pmod{p}$。

可以发现，费马小定理就是欧拉定理中 n 为素数的特殊情况。

扩展欧拉定理

当 a,n 均为正整数时（此时没有互质的条件）：

$$a^b \equiv \begin{cases} a^b & ,b < \varphi(n) \\ a^{b\%\varphi(n)+\varphi(n)} & ,b \geq \varphi(n) \end{cases} \pmod{n}$$

证明：

当 $b \leq \varphi(n)$ 时，显然成立。仅需要证明 $b \geq \varphi(n)$ 的情况：

1. 当 a,n 互质时，$a^b \equiv a^{b \bmod \varphi(n)} * 1 \equiv a^{b \bmod \varphi(n)} a^{\varphi(n)} = a^{b \bmod \varphi(n)+\varphi(n)}$
2. 当 a,n 不互质时，把 n 通过唯一分解定理拆分为 $n = p_1^{q_1} p_2^{q_2} \ldots p_n^{q_n}$，因为假设 n_1, n_2 互质，那么由 $x \equiv y \pmod{n_1}, x \equiv y \pmod{n_2}$ 可以得到 $x \equiv y \pmod{n_1 n_2}$，所以可以进行合并。下面分类讨论 $p_i^{q_i}$：
   1) 若 $\gcd(a, p_i^{q_i}) = 1$，因为欧拉函数是积性函数 $\varphi(p_i^{q_i}) | \varphi(n)$，再根据费马小定理和第一点互质的情况，可得：$a^b \equiv a^{\varphi(n)*\lfloor \frac{b}{\varphi(n)} \rfloor + b\%\varphi(n)} \equiv a^{b\%\varphi(n)} \equiv a^{b\%\varphi(n)+\varphi(n)} \pmod{p_i^{q_i}}$
   2) 若 $\gcd(a, p_i^{q_i}) \neq 1$，则 $p | a$，即 a 是 p 的倍数，设 $a = kp$；注意到 $b \geq$

$\varphi(n) \geq \varphi(p_i^{q_i})$，所以 $p_i^{q_i}$ 既是 a^b 的因数，也是 $a^{b\%\varphi(n)+\varphi(n)}$ 的因数，即 $a^b \equiv a^{b\%\varphi(n)+\varphi(n)} \equiv 0 \pmod{p_i^{q_i}}$

证毕。

扩展欧拉定理常用于欧拉降幂，例如：给定 n 个数 a_i，求 $\left(a_1^{a_2^{\cdots^{a_n}}}\right)\%p$

思路：考虑比较简单的情况：$a^{b^c}\%p$。根据欧拉降幂的公式，$a^{b^c}\%p = \left(a^{(b^c)\%\varphi(p)+[b^c>\varphi(p)]*\varphi(p)}\right)\%p$，然后根据嵌套欧拉函数的性质，对 p 求嵌套欧拉函数最多经过 $O(logp)$ 层到 1，这时更上层的指数都是 0。如何判断 $b^c > \varphi(p)$?，因为 b^c 可能很大，这时需要对取模做一个处理，在快速幂运算时如果大于模数，取模后再加上 p，这样能够保证一旦大于模数最终的结果一定可以多加一个 $\varphi(p)$。

```
ll a[maxn];
map<ll, ll> mp;

ll Mod(ll x, ll p) {
    return x > p ? x % p + p : x;
}

ll getphi(ll p) {
    if (mp.count(p)) return mp[p];
    ll n = p, ans = p;
    int m = sqrt(p + 0.5);
    for (int i = 2; i <= m; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return mp[p] = ans;
}

ll mul(ll a, ll b, ll p) {    // 如果模数过大则采用慢速乘
    ll ans = 0;
    while (b) {
        if (b & 1) ans = Mod(ans + a, p);
        a = Mod(a + a, p);
        b >>= 1;
    }
    return ans;
}

```

```
ll qkp(ll x, ll n, ll p) {
    ll ans = 1;
    while (n) {
        if (n & 1) ans = Mod(ans * x, p);
        x = Mod(x * x, p);
        n >>= 1;
    }
    return ans;
}

ll cal(int i, ll p) {   // cal(1, p)
    if (i == n || p == 1) return Mod(a[1], p);
    return qkp(a[i], cal(i + 1, getphi(p)), p);
}
```

Calculation（HDU 2837）

给出公式$f(n)=(n\%10)^{f(n/10)}, f(0)=0$，且$0^k=0$。给定$n,m$计算$f(n)\%m$。

【输入格式】

第一行输入一个正整数 T。

第二行输入两个正整数 $n,m(2\le n,m\le 10^9)$。

【输入格式】

输出 T 行，每行一个整数表示$f(n)\%m$的答案。

【分析】

使用欧拉降幂的公式计算即可，注意快速幂取模的时候若中间变量小于模数则不管，当大于等于模数时进行取模.，这样返回进行下一层运算时相当于用了欧拉降幂公式。

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

using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;

inline ll mul(ll a, ll b, ll p) {
    if (p <= 1000000000)
        return a * b % p;
    else if (p <= 1000000000000LL)
        return (((a * (b >> 20) % p) << 20) + (a * (b & ((1 << 20) - 1)))) % p;
    else {
        ll d = (ll)floor(a * (long double)b / p + 0.5);
        ll ret = (a * b - d * p) % p;
```

```
15.         if (ret < 0) ret += p;
16.         return ret;
17.     }
18. }
19.
20. ll qkp(ll x, ll n, ll p) {
21.     if (x == 0) return n == 0;
22.     ll ans = 1;
23.     x %= p;
24.     while (n) {
25.         if (n & 1) ans = mul(ans, x, p);
26.         x = mul(x, x, p);
27.         n >>= 1;
28.     }
29.     return ans == 0 ? p : ans % p;
30. }
31.
32. ll euler_phi(ll n) {
33.     int m = sqrt(n + 0.5);
34.     ll ans = n;
35.     for (int i = 2; i <= m; i++) {
36.         if (n % i == 0) {
37.             ans = ans / i * (i - 1);
38.             while (n % i == 0) n /= i;
39.         }
40.     }
41.     if (n > 1) ans = ans / n * (n - 1);
42.     return ans;
43. }
44.
45. ll f(ll n, ll m) {
46.     if (n < 10) return n;
47.     return qkp(n % 10, f(n / 10, euler_phi(m)), m);
48. }
49.
```

```
int main() {
    ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int t, n, m;
    cin >> t;
    while (t--) {
        cin >> n >> m;
        cout << f(n, m) % m << "\n";
    }
    return 0;
}

```

4.2 线性代数

矩阵快速幂、行列式、高斯消元、矩阵求逆、常系数线性递推

4.2.1 行列式

1. 行列式的性质

（1）行列式和它的转置行列式值相等，即：

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{vmatrix}$$

（2）互换行列式的任意两行（两列），行列式的值将改变正负号：

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = - \begin{vmatrix} a_{21} & a_{22} & \dots & a_{2n} \\ a_{11} & a_{12} & \dots & a_{1n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

（3）行列式某行或者某列的公因子可以提到行列式记号外面：

$$\begin{vmatrix} b_1 a_{11} & b_1 a_{12} & \dots & b_1 a_{1n} \\ b_2 a_{21} & b_2 a_{22} & \dots & b_2 a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ b_n a_{n1} & b_n a_{n2} & \dots & b_n a_{nn} \end{vmatrix} = \prod_{i=1}^{n} b_i \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

（4）行列式具有分行（列）相加性：

$$\begin{vmatrix} b_1+c_1 & b_2+c_2 & \dots & b_n+c_n \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \begin{vmatrix} b_1 & b_2 & \dots & b_n \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} + \begin{vmatrix} c_1 & c_2 & \dots & c_n \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

（5）将行列式某一行（列）的各元素同乘以一个数k后加到另外一行（列）其值不变。

这一点同矩阵初等变换

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21}+ka_{11} & a_{22}+ka_{12} & \dots & a_{2n}+ka_{1n} \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

（6）拉普拉斯定理：行列式等于它的任一行（列）的各元素与其对应的代数余子式的乘积之和：

$$|A| = \sum_{i=1}^{n} (-1)^{1+i} a_{1i} C_{1i}$$

（7）分块行列式的值等于其主对角线上两个子块行列式的值的乘积（右上角的块必须为全零）：

$$\begin{vmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ b_{11} & b_{12} & c_{11} & c_{12} \\ b_{21} & b_{22} & c_{21} & c_{22} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{vmatrix}$$

2. 行列式化简常用技巧

（1）上三角行列式、下三角行列式以及主对角线行列式的值都是主对角线上元素乘积。

$$\begin{vmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ . & . & \dots & 0 \\ . & . & \dots & 0 \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} = \prod_{i=1}^{n} a_{ii}$$

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ . & . & \dots & a_{3n} \\ . & . & \dots & . \\ 0 & 0 & \dots & a_{nn} \end{vmatrix} = \prod_{i=1}^{n} a_{ii}$$

$$\begin{vmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ . & . & \dots & 0 \\ . & . & \dots & . \\ 0 & 0 & \dots & a_{nn} \end{vmatrix} = \prod_{i=1}^{n} a_{ii}$$

（2）副对角线行列式的值：

$$\begin{vmatrix} 0 & 0 & \dots & a_{11} \\ 0 & 0 & \dots & 0 \\ . & . & \dots & . \\ 0 & a_{n-1,n-1} & \dots & . \\ a_{nn} & 0 & \dots & 0 \end{vmatrix} = (-1)^{\frac{n(n-1)}{2}} * \prod_{i=1}^{n} a_{ii}$$

（3）若行列式有两行（列）元素对应相等，那么此行列式的值为零；若行列式有一行

（列）元素全为零，那么此行列式的值为零；若行列式有两行（列）元素成比例，那么此行列式的值为零。

3. 行列式的多种计算方法

（1）定义法

计算较为复杂一般只用于低阶行列式。

（2）三角化法

将行列式化为上三角（下三角）行列式。

（3）箭形行列式

形如$\begin{vmatrix} a & a & \dots & a \\ a & b & \dots & 0 \\ . & . & \dots & . \\ a & 0 & \dots & 0 \\ a & 0 & \dots & x \end{vmatrix}$的行列式，只需按主对角线上的值$t$使得第一列减去后面每列的$\frac{1}{t}$。

（4）降阶法

根据拉普拉斯展开定理，对行列式进行适当的展开。

（5）升阶法

将原行列式增加一行一列，而保持原行列式的值不变或与原行列式有某种巧妙的关系。

例：

$$\begin{vmatrix} x & a & \dots & a \\ a & x & \dots & a \\ . & . & \dots & . \\ a & a & \dots & a \\ a & a & \dots & x \end{vmatrix} = \begin{vmatrix} 1 & a & a & \dots & a \\ 0 & x & a & \dots & a \\ 0 & a & x & \dots & a \\ 0 & . & . & \dots & . \\ 0 & a & a & \dots & a \\ 0 & a & a & \dots & x \end{vmatrix} = \begin{vmatrix} 1 & a & a & \dots & a \\ -1 & x-a & 0 & \dots & 0 \\ -1 & 0 & x-a & \dots & 0 \\ -1 & . & . & \dots & . \\ -1 & 0 & 0 & \dots & 0 \\ -1 & 0 & 0 & \dots & x-a \end{vmatrix}$$

然后$C_1 + \frac{1}{x-a}C_2, \dots, C_1 + \frac{1}{x-a}C_n$，最后化简为：$\begin{vmatrix} 1+\frac{na}{x-a} & a & a & \dots & a \\ 0 & x-a & 0 & \dots & 0 \\ 0 & 0 & x-a & \dots & 0 \\ 0 & . & . & \dots & . \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & x-a \end{vmatrix}$

（6）递推法

找出行列式和其相应的 $n-1, n-2, \dots$ 阶行列式之间的递推关系。

（7）数学归纳法

由行列式的特殊形式，计算低阶行列式的公式猜想推广到高阶行列式。

（8）拆分法

根据行列式的某些位置由若干数相加，那么根据行列式的分行（列）相加性拆分成多个行列式求解。

（9）分解乘积法

根据行列式的特点利用行列式的乘法公式把所给行列式分解成两个易求解的行列式之积，通过对这两个行列式的计算从而得到其值。

$$\begin{vmatrix} a_1+b_1 & a_1+b_2 & \dots & a_1+b_n \\ a_2+b_1 & a_2+b_2 & \dots & a_2+b_n \\ . & . & \dots & . \\ . & . & \dots & . \\ a_n+b_1 & a_n+b_2 & \dots & a_n+b_n \end{vmatrix} = \begin{vmatrix} a_1 & 1 & 0 & \dots & 0 \\ a_2 & 1 & 0 & \dots & 0 \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ a_n & 1 & 0 & \dots & 0 \end{vmatrix} * \begin{vmatrix} b_1 & b_2 & b_3 & \dots & b_n \\ 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & \dots & 0 \\ . & . & . & \dots & . \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix}$$

$$= \begin{cases} a_1+b_1 & n=1 \\ (a_1-a_2)(b_2-b_1) & n=2 \\ 0 & n \geq 3 \end{cases}$$

4.2.2 矩阵及初等变换

1. 矩阵表示

```
struct Matrix {
    int n, m;
    int matrix[505][505];

    Matrix() {}

    Matrix(int x, int y) {
        n = x, m = y;
        memset(matrix, 0, sizeof(matrix));
    }
};
```

2. 矩阵初等变换

（1）交换矩阵的两行或两列（对调 i,j，两行记为 $r_i \Leftrightarrow r_j$）

（2）一个非零数 k 乘矩阵的某一行（列）所有元素（第 i 行乘以 k 记为 $r_i * k$ ）

（3）把矩阵的某一行（列）所有元素乘以一个数 k 后加到另一行（列）对应的元素（第 j 行乘以 k 加到第 i 行记为 $r_i + k * r_j$ ）

3. 矩阵加法

```
Matrix add(Matrix a, Matrix b) {
    Matrix ans;
    for (int i = 1; i <= a.n; i++)
        for (int j = 1; j <= a.m; j++) ans.matrix[i][j] = (a.matrix[i][j] +
b.matrix[i][j]) % Mod;
    return ans;
}
```

4. 矩阵乘法

条件：仅当第一个矩阵的列数和第二个矩阵的行数相同才可以相乘；

口诀：前行乘后列；

例如：$A \times B = C, a_{i,j} \in A, b_{i,j} \in B, c_{i,j} \in C$，则 $c_{i,j} = a_{i,1} * b_{1,j} + a_{i,2} * b_{2,j} + \dots + a_{i,n} * b_{n,j}$

```
Matrix mul(Matrix a, Matrix b) {
    Matrix ans(a.n, b.m);
    for (int i = 1; i <= ans.n; i++) {
        for (int j = 1; j <= ans.m; j++)
            for (int k = 1; k <= a.m; k++) {
                ans.matrix[i][j] += a.matrix[i][k] * b.matrix[k][j] % Mod;
                ans.matrix[i][j] %= Mod;
            }
    }
    return ans;
}

Matrix qkp(Matrix mx, ll n) {
    Matrix ans(mx.n, mx.m);
    for (int i = 1; i <= mx.n; i++) ans.matrix[i][i] = 1;
    while (n) {
        if (n & 1) ans = mul(ans, mx);
        mx = mul(mx, mx);
        n >>= 1;
    }
    return ans;
}
```


4.2.3 矩阵快速幂

考虑矩阵乘法中一种特殊的情形，F 是 $1*n$ 矩阵，A 是 $n*n$ 矩阵,则 F'=F*A 也是$1*n$ 矩阵。F 和 F' 可看作一维数组，省略它们的行下标 1。按照矩阵乘法的定义，$\forall j \in [1,n]$，有$F'_j = \sum_{k=1}^{n} F_k * A_{k,j}$。它的含义是，经过一次与 A 的矩阵乘法运算后，F 数组中的第 k 个值会以 $A_{k,j}$ 为系数累加到 F' 数组的第 j 个值上，等价于在一个向量的 k,j 两个状态之间发生了递推。

问题引入：斐波那契数列形式如下：

$$F(n)=\begin{cases}1 & n=1\\ 1 & n=2\\ F(n-1)+F(n-2) & n\geq 3\end{cases}$$

试求出斐波那契数列的第 $n(1 \leq n \leq 10^9)$ 项，结果对 10^9+7 取模。

对于斐波那契数列的递推公式：①$F(n)=F(n-1)+F(n-2)$，我们可以得到 ②$F(n-1)=F(n-1)+0*F(n-2)$。

根据线性代数知识，联立 ① 和 ②，可以求得：

$$\begin{bmatrix}F(n)\\F(n-1)\end{bmatrix}=\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}F(n-1)\\F(n-2)\end{bmatrix}$$

其中的常数矩阵称为递推式的特征矩阵。

实际上上述的推导特征矩阵的过程不是通用的。对于一个更一般的递推式$F(n) = \mathrm{a} * F(n-1) + \mathrm{b} * F(n-2)$，假设 $X_{n-1} = \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$，显然有 $X_n = \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$，现在已知的是我们可以通过构造一个二阶方阵 m 使得 $X_n = m * X_{n-1}$，首先假设方阵的所有项都是未知数，即依次将等号右边的第二个式子代入，可以得到：$m = \begin{bmatrix} p & q \\ r & s \end{bmatrix}$，那么有 $\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} p & q \\ r & s \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$，根据矩阵乘法的规律，我们不难一一求出$p, q, r, s$，最后即可得到特征方阵。通过递推规律，我们可以得到 X_n 和前两项的规律：

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} a & b \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} F(2) \\ F(1) \end{bmatrix}$$

这样我们可以通过构造矩阵，然后使用矩阵乘法形式的快速幂来求解斐波那契数列的第 n 项了。

```
1. typedef long long ll;
2. const int Mod = 1e9 + 7;
3.
4. struct Matrix {
5.     ll matrix[10][10];
6.     int n, m;
7.
8.     Matrix() {}
9.
10.    Matrix(int x, int y) {
11.        n = x, m = y;
12.        memset(matrix, 0, sizeof(matrix));
13.    }
14. };
15.
16. Matrix mul(Matrix a, Matrix b) {
17.    Matrix ans(a.n, b.m);
18.    for (int i = 1; i <= ans.n; i++)
19.        for (int j = 1; j <= ans.m; j++)
20.            for (int k = 1; k <= a.m; k++) {
21.                ans.matrix[i][j] += a.matrix[i][k] * b.matrix[k][j] % Mod;
22.                ans.matrix[i][j] %= Mod;
23.            }
24.    return ans;
25. }
26.
27. Matrix qkp(Matrix mx, ll n) {
28.    Matrix ans(mx.n, mx.m);
29.    for (int i = 1; i <= mx.n; i++) ans.matrix[i][i] = 1;
```

```
    while (n) {
        if (n & 1) ans = mul(ans, mx);
        mx = mul(mx, mx);
        n >>= 1;
    }
    return ans;
}

ll solve(ll x) {
    if (x == 1 || x == 2) return 1;
    Matrix a(2, 2);
    a.matrix[1][1] = 1, a.matrix[1][2] = 1;
    a.matrix[2][1] = 1, a.matrix[2][2] = 0;
    Matrix b(2, 1);
    b.matrix[1][1] = 1, b.matrix[2][1] = 1;
    Matrix ans = mul(qkp(a, x - 2), b);
    return ans.matrix[1][1];
}
```

上述问题使我们对“矩阵乘法加速递推”有了初步的认识。一般来说，如果一类问题具有以下特点:

1. 可以抽象出一个长度为 n 的一维向量，该向量在每个单位时间发生一次变化。

2. 变化的形式是一个线性递推(只有若干“加法”或“乘一个系数”的运算)。

3. 该递推式在每个时间可能作用于不同的数据上，但本身保持不变。

4. 向量变化时间（即递推的轮数）很长，但向量长度 n 不大。

那么可以考虑采用矩阵乘法进行优化。我们把这个长度为 n 的一维向量称为“状态矩阵”，把用于与“状态矩阵”相乘的固定不变的矩阵 A 称为“转移矩阵”。若状态矩阵中的第 x 个数对下一单位时间状态矩阵中的第 y 个数产生影响，则把转移矩阵的第 x 行第 y 列赋值为适当的系数。

矩阵乘法加速递推的关键在于定义出“状态矩阵”，并根据递推式构造出正确的“转移矩阵”，之后就可以利用快速幂和矩阵乘法完成程序实现了。时间复杂度为 $O(n^3 \log T)$，其中 T 为递推总轮数，n 为矩阵的行数（列数）。

Recursive sequence（HDU 5950）

给出一个如下的递推式：

$$f(n) = f(n-1) + 2f(n-2) + n^4$$

给出前两项求第n项对 2147493647 取模的结果。

【输入格式】

第一行输入一个正整数 T代表测试用例的数量。。

接下来T行，每行输入三个整数 $n, a, b (n, a, b \leq 2^{31})$。

【输入格式】

输出T行，每行一个整数表示答案。

【分析】

对于n^4，实际上和$f(n)$并不是递推关系，一开始想的是将n^4和前面部分拆开来求，但是这样是错误的，不能拆开，因此考虑以下思路：

$$n^4 = [(n-1)+1]^4 = C_4^0(n-1)^4 + C_4^1(n-1)^3 + C_4^2(n-1)^2 + C_4^1(n-1) + 1$$

那么有：

$$f(n) = f(n-1) + 2f(n-2) + C_4^0(n-1)^4 + C_4^1(n-1)^3 + C_4^2(n-1)^2 + C_4^1(n-1) + 1$$

对于这个东西，我们首先设$X_{n-1} = \begin{bmatrix} f(n-1) \\ f(n-2) \\ (n-1)^4 \\ (n-1)^3 \\ (n-1)^2 \\ (n-1)^1 \\ 1 \end{bmatrix}$

那么不难得出$X_n = \begin{bmatrix} f(n) \\ f(n-1) \\ n^4 \\ n^3 \\ n^2 \\ n^1 \\ 1 \end{bmatrix}$

考虑构造出一个7×7的方阵m使得$X_n = m * X_{n-1}$，我们可以首先假设所有项都为未知数，然后根据矩阵乘法的规律，可以求出每一项，最后可以得到：

$$m = \begin{bmatrix} 1 & 2 & 1 & 4 & 6 & 4 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

那么最后的递推矩阵式为：

$$\begin{bmatrix} f(n) \\ f(n-1) \\ n^4 \\ n^3 \\ n^2 \\ n^1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 & 4 & 6 & 4 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 6 & 4 & 1 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{n-2} \begin{bmatrix} f(2) \\ f(1) \\ 2^4 \\ 2^3 \\ 2^2 \\ 2^1 \\ 1 \end{bmatrix}$$

```
1. #include <bits/stdc++.h>
2.
3. using namespace std;
4. typedef long long ll;
5. const ll Mod = 2147493647;
6. const int maxn = 2e5 + 10;
```

```
7.
8.  const int arr1[10][10] = {{1, 2, 1, 4, 6, 4, 1},
9.                            {1, 0, 0, 0, 0, 0, 0},
10.                           {0, 0, 1, 4, 6, 4, 1},
11.                           {0, 0, 0, 1, 3, 3, 1},
12.                           {0, 0, 0, 0, 1, 2, 1},
13.                           {0, 0, 0, 0, 0, 1, 1},
14.                           {0, 0, 0, 0, 0, 0, 1}};
15. int arr2[] = {0, 0, (1 << 4), (1 << 3), (1 << 2), (1 << 1), 1};
16.
17. struct Matrix {
18.     ll matrix[10][10];
19.     int n, m;
20.
21.     Matrix() {}
22.
23.     Matrix(int x, int y) {
24.         n = x, m = y;
25.         memset(matrix, 0, sizeof(matrix));
26.     }
27. };
28.
29. Matrix mul(Matrix a, Matrix b) {
30.     Matrix ans(a.n, b.m);
31.     for (int i = 1; i <= ans.n; i++) {
32.         for (int j = 1; j <= ans.m; j++) {
33.             for (int k = 1; k <= a.m; k++) {
34.                 ans.matrix[i][j] += a.matrix[i][k] * b.matrix[k][j] % Mod;
35.                 ans.matrix[i][j] %= Mod;
36.             }
37.         }
38.     }
39.     return ans;
40. }
41.
```

```
42. Matrix qkp(Matrix mx, ll n) {
43.     Matrix ans(mx.n, mx.m);
44.     for (int i = 1; i <= mx.n; i++) ans.matrix[i][i] = 1;
45.     while (n) {
46.         if (n & 1) ans = mul(ans, mx);
47.         mx = mul(mx, mx);
48.         n >>= 1;
49.     }
50.     return ans;
51. }
52.
53. ll solve(ll n) {
54.     if (n == 1) return arr2[1];
55.     if (n == 2) return arr2[0];
56.     Matrix mx(7, 7);
57.     for (int i = 1; i <= 7; i++) {
58.         for (int j = 1; j <= 7; j++) {
59.             mx.matrix[i][j] = arr1[i - 1][j - 1];
60.         }
61.     }
62.     Matrix dw(7, 1);
63.     for (int i = 1; i <= 7; i++) {
64.         dw.matrix[i][1] = arr2[i - 1];
65.     }
66.     Matrix ans = mul(qkp(mx, n - 2), dw);
67.     return ans.matrix[1][1];
68. }
69.
70. int main() {
71.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
72.     int n, T;
73.     cin >> T;
74.     while (T--) {
75.         cin >> n >> arr2[1] >> arr2[0];
76.         cout << solve(n) << endl;
```

```
    }
    return 0;
}

```

石头游戏（AcWing 206）

石头游戏在一个 n行 m列的网格上进行，每个格子对应一种操作序列，操作序列至多有 10 种，分别用 0~9 这 10 个数字指明。

操作序列是一个长度不超过 6 且循环执行、每秒执行一个字符的字符串。

每秒钟，所有格子同时执行各自操作序列里的下一个字符。

序列中的每个字符是以下格式之一：

1. 数字 0~9：表示拿 0~9 个石头到该格子。
2. NWSE：表示把这个格子内所有的石头推到相邻的格子，N 表示上方，W 表示左方，S 表示下方，E 表示右方。
3. D：表示拿走这个格子的所有石头。

给定每种操作序列对应的字符串，以及网格中每个格子对应的操作序列，求石头游戏进行了 t秒之后，石头最多的格子里有多少个石头。

在游戏开始时，网格是空的。

【输入格式】

第一行 4 个整数 n, m, t, act。

接下来 n 行，每行 m 个字符，表示每个格子对应的操作序列。

最后 act 行，每行一个字符串，表示从 0 开始的每个操作序列。

$1 \le m, n \le 8, 1 \le t \le 10^8, 1 \le act \le 10$

【输入格式】

一个整数：游戏进行了 t秒之后，所有方格中石头最多的格子有多少个石头。

【分析】

设$\forall i \in [1, n], j \in [1, m], num(i, j) = (i - 1) * m + j$。

把网格看作长度为$n * m$的一维向量,定义 1 行$n * m + 1$列的“状态矩阵”F,下标为$0 \sim n * m$，其中$F[num(i, j)]$记录格子(i, j)中石头的个数。

特别地，令$F[0]$始终等于 1。

F 会随着时间的增长而不断变化。设F_k表示k秒之后的“状态矩阵”。在游戏开始时，根据题意和 F 的定义，有$F_0 = [1\ 0\ 0 \dots 0]$。

注意到操作序列的长度不超过 6，而 1~6 的最小公倍数是 60,所以每经过 60 秒，所有操作序列都会重新处于最开始的字符处。我们可以统计出第k秒$(1 \le k \le 60)$各个格子执行了什么字符，第$k + 60$秒执行的字符与第k秒一定是相同的。

对于 1~60 之间的每个k，各个格子在第k秒执行的操作字符可以构成一个“转移矩阵”

A_k，矩阵行、列下标都是$0\sim n*m$。构造方法如下:

1. 若网格(i,j)第k秒的操作字符为“N”，且$i>1$，则令$A_k[num(i,j),num(i-1,j)]=1$，表示把石头推到上边的格子。字符“W”“S”“E”类似。

2. 若网格(i,j)第k秒的操作字符是一个数字x，则令$A_k[0,num(i,j)]=x$，$A_k[num(i,j),num(i,j)]=1$，表示格子里本来就有的石头不动，再拿x块石头。

3. 令$A_k[0,0]=1$。

4. A_k的其他部分均赋值为 0。

上述构造实际上把“状态矩阵”的第 0 列作为“石头来源”。$A_k[0,0]=1$保证了$F[0]$始终等于 1。$A_k[0,num(i,j)]=x$相当于从“石头来源”获取x个石头，放到格子(i,j)中。

使用矩阵乘法加速递推，遇到常数项时，经常需要在“状态矩阵”中添加一个额外的位置，始终存储常数 1，并乘上“转移矩阵”中适当的系数，累加到“状态矩阵”的其他位置

设$A=\prod_{i=1}^{60}A_i$，$t=q*60+r(0\le r<60)$。根据上面的讨论：$F_t=F_0*A^q*\prod_{i=1}^{60}A_i$。使用矩阵乘法和快速幂计算该式，$F_t$ ↵第$1\sim n*m$列中的最大值即为所求。

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

using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}

struct Matrix {
    int n, m;
    ll a[65][65];

    Matrix() {}

    Matrix(int x, int y) {
```

```
22.         n = x, m = y;
23.         memset(a, 0, sizeof a);
24.     }
25.
26.     Matrix operator*(const Matrix &p) const {
27.         Matrix ret(n, p.m);
28.         for (int i = 0; i <= n; i++)
29.             for (int j = 0; j <= p.m; j++)
30.                 for (int k = 0; k <= m; k++) ret.a[i][j] += a[i][k] * p.a[k][j];
31.         return ret;
32.     }
33. } t[65];
34.
35. Matrix qkp(Matrix mx, int q) {
36.     Matrix ans(mx.n, mx.m);
37.     for (int i = 0; i <= ans.n; i++) ans.a[i][i] = 1;
38.     while (q) {
39.         if (q & 1) ans = ans * mx;
40.         mx = mx * mx;
41.         q >>= 1;
42.     }
43.     return ans;
44. }
45.
46. string s[15];
47. int b[10][10];
48. int n, m, num, cnt, tot;
49.
50. inline int ID(int p, int q) {
51.     return n * (p - 1) + q;
52. }
53.
54. int main() {
55.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
56.     cin >> n >> m >> num >> cnt;
```

```
    for (int i = 1; i <= n; i++) {
        cin >> s[0];
        for (int j = 1; j <= s[0].size(); j++) {
            b[i][j] = s[0][j - 1] - '0';
        }
    }
    for (int i = 0; i < cnt; i++) cin >> s[i];
    tot = 1;
    for (int i = 0; i < cnt; i++) tot = lcm(tot, (int)s[i].size());
    for (int i = 0; i < cnt; i++) {
        int x = tot / (int)s[i].size();
        string res = "";
        for (int j = 1; j <= x; j++) res += s[i];
        s[i] = res;
    }
    Matrix G(n * m, n * m);
    for (int i = 0; i <= n * m; i++) G.a[i][i] = 1;
    for (int k = 1; k <= tot; k++) {    // 求出每秒的转移矩阵
        t[k].n = t[k].m = n * m, t[k].a[0][0] = 1;
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++) {
                char ch = s[b[i][j]][k - 1];
                if (ch >= '0' && ch <= '9')
                    t[k].a[0][ID(i, j)] = ch - '0', t[k].a[ID(i, j)][ID(i, j)] =
1;
                else if (ch == 'N' && i > 1)
                    t[k].a[ID(i, j)][ID(i - 1, j)] = 1;
                else if (ch == 'S' && i < n)
                    t[k].a[ID(i, j)][ID(i + 1, j)] = 1;
                else if (ch == 'W' && j > 1)
                    t[k].a[ID(i, j)][ID(i, j - 1)] = 1;
                else if (ch == 'E' && j < m)
                    t[k].a[ID(i, j)][ID(i, j + 1)] = 1;
            }
        G = G * t[k];
```

```
    }
    int x = num / tot, y = num % tot;
    Matrix F(0, n * m);
    F.a[0][0] = 1;
    G = qkp(G, x);
    F = F * G;
    for (int k = 1; k <= y; k++) F = F * t[k];
    ll ans = 0;
    for (int i = 1; i <= n * m; i++) ans = max(ans, F.a[0][i]);
    cout << ans << "\n";
    return 0;
}
```

4.2.4 高斯消元

高斯消元法（Gauss-Jordan elimination）是求解线性方程组的经典算法，它在当代数学中有着重要的地位和价值，是线性代数课程教学的重要组成部分。高斯消元法除了用于线性方程组求解外，还可以用于行列式计算、求矩阵的逆，以及其他计算机和工程方面。

4.2.4.1 线性方程组的求解

首先简单复习一下矩阵初等变换：

1) 交换矩阵的两行或两列（对调 i,j，两行记为 $r_i \Leftrightarrow r_j$）
2) 以一个非零数 k 乘矩阵的某一行（列）所有元素（第 i 行乘以 k 记为 $r_i * k$ ）
3) 把矩阵的某一行（列）所有元素乘以一个数 k 后加到另一行（列）对应的元素（第 j 行乘以 k 加到第 i 行记为 $r_i + k * r_j$ ）

对于下列 n 元一次线性方程组：

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \cdots\cdots & \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

我们用“系数矩阵*未知数矩阵=常数矩阵”来表示上述线性方程组：

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

然后我们称矩阵 (A,b) 为该方程组的增广矩阵：

$$(A,b)=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{pmatrix}$$

消元法解线性方程组过程实际上是对方程组未知数的系数和常数项作有限制的运算，未知变量并未参与运算。因此，将解线性方程组的消元运算移植到矩阵变换中就是对线性方程组的增广矩阵进行三种初等变换并化简为行阶梯矩阵或者行最简形矩阵。

在线性代数的学习中，我们是通过矩阵的秩来判断线性方程组是否有解，但是如果写成程序，就需要分以下几种情况讨论解：

① 无解：当方程中出现 $(0,0,\dots,0,a)$ 的形式，且 $a!=0$ 时；或者某一列全为 0，说明是无解的。

② 唯一解：如果行阶梯矩阵形成了严格的上（下）三角阵，或者可以化为 n 阶行最简矩阵，那么就有唯一解。

③ 无穷解：存在无穷解即代表至少有一行元素全为 0（包括第 $n+1$ 行），那么如果有 k 行就代表有 k 个自由变元。

4.2.4.2 高斯消元法

高斯消元法就是利用矩阵初等变换，将增广矩阵变换为行阶梯矩阵，进而通过逐层回带求出解。总的来说主要进行以下步骤（以下三角行阶梯矩阵为例）：

消元第一步：每次考虑 $a[i][i]$，然后从第 i 列第 i 行以下中找出该列最大的元素，并将那一行和该行互换位置。之所以找最大的，是因为下面要用该数做除法，而除数越小误差越大。

消元第二步：如果上面找到的最大值仍为 0 证明无解，直接返回。

消元第三步：第 i 列从第 $j=i+1$ 行开始，先除以 $a[i][i]$ 并保存结果，接着该行都减去该行乘以上述结果，以达到将第 j 行第 i 列消为 0 的目的。重复上述三步 n 次即可。

消元第四步：由于我们上述化的是下三角型行阶梯矩阵，那么最后一行只有一个变量，倒数第二行有两个变量...以此类推，那么之间回带求出结果保存在 $a[i][n+1]$ 位置上。

```
1. const double eps = 1e-8;
2. double a[1005][1005];
3. int n;
4.
5. bool gauss() {
6.     for (int i = 1; i < n; i++) {
7.         int temp = i;
8.         for (int j = i + 1; j <= n; j++)
9.             if (fabs(a[j][i]) > fabs(a[temp][i])) temp = j;
```

```
        // 如果当前列的最大值仍然为 0，无解
        if (fabs(a[temp][i]) < eps) return false;
        // 与最大主元所在行交换
        if (temp != i) {
            for (int j = i; j <= n + 1; j++) swap(a[i][j], a[temp][j]);
        }
        double p;
        for (int j = i + 1; j <= n; j++) {   // 消元
            p = a[j][i] / a[i][i];
            for (int k = i; k <= n + 1; k++) a[j][k] -= a[i][k] * p;
        }
    }
    // 回代求解
    for (int i = n; i >= 1; i--) {
        for (int j = i + 1; j <= n; j++) a[i][n + 1] -= a[i][j] * a[j][n + 1];
        a[i][n + 1] /= a[i][i];
    }
    return true;
}
```

4.2.4.3 高斯-约旦消元法

此算法是基于高斯消元的基础上改进而成的，唯一不同之处是多进行了几次矩阵变换使得化为行最简型矩阵。相比于高斯消元法此方法效率较低，但是不用回带求解。

此算法的过程大概为：首先确定每个 $a[i][i]$ 不为 0（如果为 0 就和下面的行替换），接着将每个 $a[i][i]$ 通过行变换化为 1（该行都除以 $a[i][i]$），然后通过行变换将每列除了 $a[i][i]$ 以外的都消成 0，重复 n 次即可。

对于一般的题目不需要这么麻烦，化为行阶梯矩阵即可。

```
const double eps = 1e-8;
double a[1005][1005];

bool gauss_jordan(int n) {
    int tmp;
    for (int i = 1; i <= n; i++) {
        tmp = i;
        while (fabs(a[tmp][i]) < eps && tmp <= n) tmp++;
        if (tmp == n + 1) return false;
        for (int j = 1; j <= n + 1; j++) swap(a[i][j], a[tmp][j]);
        double p = a[i][i];
        for (int j = 1; j <= n + 1; j++) a[i][j] = a[i][j] / p;
        for (int j = 1; j <= n; j++) {
            if (i != j) {
```

```
                double q = a[j][i];
                for (int k = 1; k <= n + 1; k++) a[j][k] -= q * a[i][k];
            }
        }
    }
    return true;
}
```

4.2.4.4 高斯消元判断解的情况

常规的高斯消元只适用于快速求出有解的情况，但是对于区分无穷解和无解以及在无穷解下有多少个自由变元比较难搞，因此需要对高斯消元作出适当改进：

一般的高斯消元都是考虑主对角线的每个位置 (i,i)，如果有解那么行列的下标永远是相同的，但是如果出现了无穷解，即代表某行为 $0,0,0,\ldots,0$，此时我们应该将它暂时放置，行不变但是列继续下移，直到能在下面找到 $a[r][c]!=0$，那么将全为 0 的这行交换到下面，重复这个操作，最后就能保证所有的自由变元的行都在矩阵的下方，例如：

$$\begin{bmatrix}1&2&-3&0&1\\0&0&0&0&0\\0&0&1&3&4\\0&0&0&1&6\end{bmatrix}\Rightarrow\begin{bmatrix}1&2&-3&0&1\\0&0&1&3&4\\0&0&0&1&6\\0&0&0&0&0\end{bmatrix}$$

显然若记录行的向量最后没有到达 $n+1$ 代表出现了无穷解或者无解，若某行为 $0,0,0,\ldots,0,1$ 则代表无解；若某行为 $0,0,0,\ldots,0$ 则代表有无穷解，设此时列的下标为 i，那么自由变元的个数就为 $n-i+1$。

```
int n;
double a[105][105];

int guass() {
    int r = 1;
    for (int c = 1; c <= n; c++) {
        int temp = r;
        for (int i = r + 1; i <= n; i++) {
            if (fabs(a[i][c]) > fabs(a[temp][c])) temp = i;   // 每行第一个非零元素越大误差越小
        }
        if (fabs(a[temp][c]) < eps) continue;   // 这步是实现行列异步的关键
        for (int j = c; j <= n + 1; j++) swap(a[r][j], a[temp][j]);
        for (int j = n + 1; j >= c; j--) a[r][j] /= a[r][c];
        for (int i = r + 1; i <= n; i++) {
            for (int j = n + 1; j >= c; j--)
                if (fabs(a[i][c]) >= eps) {
                    a[i][j] -= a[i][c] * a[r][j];
                }
```

```
        }
        r++;
    }
    if (r <= n) {
        for (int i = 1; i <= n; i++) {
            bool ok = 0;
            for (int j = 1; j <= n; j++)
                if (fabs(a[i][j]) > eps) ok = 1;
            if (ok) continue;
            if (fabs(a[i][n + 1]) >= eps)
                return -1;   // 无解
            else
                return 0;   // 无穷解
        }
    }
    // 有唯一解，回代求解
    for (int i = n; i >= 1; i--) {
        for (int j = i + 1; j <= n; j++) a[i][n + 1] -= a[i][j] * a[j][n + 1];
        a[i][n + 1] /= a[i][i];
    }
    return 1;
}
```


4.2.4.5 异或高斯消元

给出如下异或线性方程组 $a_{ij} = b_k = 1/0$，若无解输出 -1，否则输出解的个数。

$$\begin{cases} a_{11}x_1 \oplus a_{12}x_2 \oplus \ldots \oplus a_{1n}x_n = b_1 \\ a_{21}x_1 \oplus a_{22}x_2 \oplus \ldots \oplus a_{2n}x_n = b_2 \\ \cdots\cdots \\ a_{n1}x_1 \oplus a_{n2}x_2 \oplus \ldots \oplus a_{nn}x_n = b_n \end{cases}$$

异或是不进位加法，不难得出任何 $x_i = 1/0$，那么仍可以使用高斯消元的思想去解异或方程组：构造增广矩阵然后对 (i,i) 位置上进行消元，因为系数和最后的常数要么是 0 要么是 1，因此消元时的减法替换为异或即可。

位运算，每位要么为 0 要么为 1，显然可以使用 $bitset$ 或者用整形数状压表示每一行，又因为这时的按位异或和交换的操作都是 $O(1)$，因此时间复杂度可以被优化到 $O(n^2)$。

因为 $bitset$ 和整形数的位的保存从左到右是 $[n-1,0]$，那么将第 0 为作为常数矩阵，前面为 $n * n$ 的矩阵且列号是从 n 开始的。

1. 状压解法

常规的高斯消元判断解的情况转化为位操作即可。

```
int a[maxn], ans[maxn];

```

```
int xor_guass() {
    int r = 1;
    for (int c = n; c >= 1; c--) {
        int temp = r;
        for (int i = r; i <= n; i++)
            if ((a[i] >> c) & 1) {
                temp = i;
                break;
            }
        if (!((a[temp] >> c) & 1)) continue;
        swap(a[r], a[temp]);
        for (int i = r + 1; i <= n; i++) {
            if ((a[i] >> c) & 1) a[i] ^= a[r];
        }
        r++;
    }
    if (r <= n) {
        for (int i = 1; i <= n; i++) {
            if (a[i] == 1) return -1;
            if (a[i] == 0) return 1 << (n - i + 1);
        }
    }
    for (int i = 1; i <= n; i++) {
        int res = a[i] & 1;
        for (int j = i - 1; j; j--)
            res ^= ((a[i] >> j) & 1) * ans[j];
        ans[i] = res;
    }
    return 1;
}
```


2. bitset 解法

$bitset$ 上 $[1,n+1]$ 分别对应系数 $a_{i1},a_{i2},\ldots,a_{in},b_i$，如下代码固定将自由变元取 1，这样得到其中之一的解。

```
int ans[maxn], n;
bitset<maxn> a[maxn];

bool xor_guass() {
    int r = 1;
    for (int c = 1; c <= n; c++) {
        int temp = r;
        for (int i = r; i <= n; i++) {
            if (a[i][c]) {
                temp = i;
```

```
                break;
            }
        }
        if (!a[temp][c]) {
            ans[c] = 1;   // 固定将自由变元取 1
            continue;
        }
        if (temp != r) swap(a[r], a[temp]);
        for (int i = r + 1; i <= n; i++) {
            if (a[i][c]) a[i] ^= a[r];
        }
        r++;
    }
    if (r <= n) {
        for (int i = n; i >= 1; i--) {
            if (!a[i].count()) continue;
            int pos = a[i]._Find_first(), res = a[pos][n + 1];
            if (pos == n + 1) return 0;
            for (int j = pos + 1; j <= n; j++) {
                res ^= a[i][j] * ans[j];
            }
            ans[pos] = res;
        }
    } else {
        for (int i = n; i >= 1; i--) {
            int res = a[i][n + 1];
            for (int j = i + 1; j <= n; j++) res ^= a[i][j] * ans[j];
            ans[i] = res;
        }
    }
    return 1;
}
```


4.2.4.6 矩阵求逆

矩阵 A 可逆的充要条件为 $|A| \neq 0$，其中 $|A|$ 称为矩阵的行列式

对于 n 阶方阵 A ,若存在一个 n 阶方阵 B ， 使得 $AB = BA = E$,其中 E 为单位矩阵，则称 A 为可逆阵且称 B 为 A 的逆矩阵，记作 A^{-1}

特殊的，对于二阶方阵 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$，其逆矩阵为 $\frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

对于高阶方阵来说，定义 A^* 称为 A 的伴随矩阵，其求法为 $A_{i,j} = (-1)^{i+j}|B|$，其中 B 矩阵是除去第 i 行第 j 列剩下的数按照上下左右不变的关系凑成的 $n-1$ 阶矩阵。那么

$A^{-1} = \frac{1}{|A|}A^*$

因此求解思路如下：

- 将 A 的同阶单位 矩阵和 A 水平拼放在一起形成 $n * 2n$ 的新矩阵 B
- 对 B 进行矩阵初等变换使 B 的左半部分变为单位矩阵
- 此时 B 的右半部分即为逆矩阵 A^{-1}
- $AI \Rightarrow IA^{-1}$

这时使用高斯-约旦消元即可解决，时间复杂度 $O(n^3)$

```
const int Mod = 1e9 + 7;

ll a[505][1005];

ll qkp(ll x, ll n, ll p) {
    ll ans = 1;
    x %= p;
    while (n) {
        if (n & 1) ans = ans * x % p;
        x = x * x % p;
        n >>= 1;
    }
    return ans;
}

void print(int n,int m){
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            printf("%lld%c", a[i][j], j == 2 * n ? '\n' : ' ');
        }
}

bool gauss_jordan(int n, int m) {
    int tmp;
    for (int i = 1; i <= n; i++) {
        tmp = i;
        while (!a[tmp][i] && tmp <= n) tmp++;
        if (tmp == n + 1) return false;
        for (int j = 1; j <= m; j++)
            swap(a[i][j], a[tmp][j]);
        ll p = a[i][i], inv = qkp(a[i][i], Mod - 2, Mod);
        for (int j = 1; j <= m; j++)
            a[i][j] = a[i][j] * inv % Mod;
        for (int j = 1; j <= n; j++) {
            if (i != j) {
```

```
                ll q = a[j][i];
                for (int k = 1; k <= m; k++)
                    a[j][k] = (a[j][k] - q * a[i][k] % Mod + Mod) % Mod;
            }
        }
        //print(n,m);
    }
    return true;
}
```

4.2.5 常系数线性递推

1. 常系数线性递推数列的定义

由初值$A_1 = p_1, A_2 = p_2, .., A_k = p_k$，以及递推方程$A = F(A_{n-k}, A_{n-k+1}, \dots, A_{n-1})$构成的数列，我们称之为常系数线性递推数列，其中函数$F(A_{n-k}, A_{n-k+1}, \dots, A_{n-1}) = q_1 A_{n-k} + q_2 A_{n-k+1} + \cdots + q_k A_{n-1}$的系数 q 均为常数。

例如斐波那契数列就是常系数线性递推数列。

定理 1：若数列 $\{f_n\}$ 满足递推方程 F，则数列 $\{\lambda * f_n\}$ 也满足递推方程 F。

证明：两边都同时乘上系数 λ 即可。

定理 2：若数列 $\{f_n\}, \{g_n\}$ 满足递推方程 F，则数列 $\{f_n + g_n\}$ 也满足递推方程 F。

证明：由于递推方程的系数是一样的，所以可以用乘法分配律合并。

由以上两条定理可以推得如下引理：若数列 $\{a_n\}, \{b_n\}, \{c_n\}, \dots$ 满足递推方程 F，则数列 $\{s_1 a_n + s_2 b_n + s_3 c_n + \dots\}$ 也满足递推方程 F。

2. 常系数线性递推数列的解法

数列的前 k 项是已知的常数，那么根据引理，我们不妨用解线性方程组的方法求解数列的通项公式。

我们假设数列 $\{f_{1n}\}, \{f_{2n}\}, \dots, \{f_{kn}\}$ 都是递推方程 F 的解，我们可以认为原数列的通项公式可以表示为 $A = \{s_1 f_{1n} + s_2 f_{2n} + \cdots + s_k f_{kn}\}$。那么我们就可以带入原数列的前 k 项联立线性方程组，解出系数 $s_1, s_2, \dots, s_k$，就可以得到原数列的通项公式。

4.2.5.1 BM 算法

Berlekamp-Massey 算法，常简称为 BM 算法，是用来求解一个数列的最短线性递推式的算法。

BM 算法可以在 $O(n^2)$ 的时间内求解一个长度为 n 的数列的最短线性递推式。

```
#define rep(i, a, n) for (int i = a; i < n; i++)
#define SZ(x) ((int)(x).size())
```

```
typedef vector<int> VI;
typedef long long ll;
const int N = ? ;   // 传入多少项这里设置

ll powMOD(ll a, ll b) {
    ll ans = 1;
    for (; b; b >>= 1, a = a * a % Mod)
        if (b & 1) ans = ans * a % Mod;
    return ans;
}

namespace linear_seq {
ll res[N], base[N], _c[N], _md[N];

vector<int> Md;

void mul(ll *a, ll *b, int k) {
    rep(i, 0, k + k) _c[i] = 0;
    rep(i, 0, k) if (a[i]) rep(j, 0, k) _c[i + j] = (_c[i + j] + a[i] * b[j]) % Mod;
    for (int i = k + k - 1; i >= k; i--)
        if (_c[i])
            rep(j, 0, SZ(Md)) _c[i - k + Md[j]] = (_c[i - k + Md[j]] - _c[i] *
_md[Md[j]]) % Mod;
    rep(i, 0, k) a[i] = _c[i];
}

int solve(ll n, VI a, VI b) {
    ll ans = 0, pnt = 0;
    int k = SZ(a);
    assert(SZ(a) == SZ(b));
    rep(i, 0, k) _md[k - 1 - i] = -a[i];
    _md[k] = 1;
    Md.clear();
    rep(i, 0, k) if (_md[i] != 0) Md.push_back(i);
    rep(i, 0, k) res[i] = base[i] = 0;
    res[0] = 1;
    while ((1ll << pnt) <= n) pnt++;
    for (int p = pnt; p >= 0; p--) {
        mul(res, res, k);
        if ((n >> p) & 1) {
            for (int i = k - 1; i >= 0; i--) res[i + 1] = res[i];
            res[0] = 0;
            rep(j, 0, SZ(Md)) res[Md[j]] = (res[Md[j]] - res[k] * _md[Md[j]]) % Mod;
        }
```

```
46.     }
47.     rep(i, 0, k) ans = (ans + res[i] * b[i]) % Mod;
48.     if (ans < 0) ans += Mod;
49.     return ans;
50. }
51.
52. VI BM(VI s) {
53.     VI C(1, 1), B(1, 1);
54.     int L = 0, m = 1, b = 1;
55.     rep(n, 0, SZ(s)) {
56.         ll d = 0;
57.         rep(i, 0, L + 1) d = (d + (ll)C[i] * s[n - i]) % Mod;
58.         if (d == 0)
59.             ++m;
60.         else if (2 * L <= n) {
61.             VI T = C;
62.             ll c = Mod - d * powMOD(b, Mod - 2) % Mod;
63.             while (SZ(C) < SZ(B) + m) C.pb(0);
64.             rep(i, 0, SZ(B)) C[i + m] = (C[i + m] + c * B[i]) % Mod;
65.             L = n + 1 - L;
66.             B = T;
67.             b = d;
68.             m = 1;
69.         } else {
70.             ll c = Mod - d * powMOD(b, Mod - 2) % Mod;
71.             while (SZ(C) < SZ(B) + m) C.pb(0);
72.             rep(i, 0, SZ(B)) C[i + m] = (C[i + m] + c * B[i]) % Mod;
73.             ++m;
74.         }
75.     }
76.     return C;
77. }
78.
79. int gao(VI &a, ll n) {   // 调用这个函数，传入打表前几项的 vector，返回答案(默认对 int 数
取模答案为 int)
80.     VI c = BM(a);
81.     c.erase(c.begin());
82.     rep(i, 0, SZ(c)) c[i] = (Mod - c[i]) % Mod;
83.     return solve(n, c, VI(a.begin(), a.begin() + SZ(c)));
84. }
85.
86. };   // namespace linear_seq
```

4.2.5.2 黑科技算法

此算法暂时未得知名字，采用 BM，矩阵快速幂，牛顿迭代法等多种算法求解常系数线性递推公式。对于大部分此类题目该算法表现良好。

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

#define rep(i, a, b) for(ll i=a;i<=b;i++)
using namespace std;
typedef long long ll;
const ll N = 25;
const ll mo = 1e9 + 7;

//快速幂
ll fpow(ll a, ll b) {
    ll ans = 1;
    while (b > 0) {
        if (b & 1)ans = ans * a % mo;
        b >>= 1;
        a = a * a % mo;
    }
    return ans;
}

//BM 算法 求线性递推数列的递推公式
vector<ll> BM(const vector<ll> &s) {
    vector<ll> C = {1}, B = {1}, T;
    ll L = 0, m = 1, b = 1;
    rep(n, 0, s.size() - 1) {
        ll d = 0;
        rep(i, 0, L)d = (d + s[n - i] % mo * C[i]) % mo;
        if (d == 0)m++;
        else {
            T = C;
            ll t = mo - fpow(b, mo - 2) * d % mo;
            while (C.size() < B.size() + m)C.push_back(0);
            rep(i, 0, B.size() - 1)C[i + m] = (C[i + m] + t * B[i]) % mo;
            if (2 * L > n)m++;
            else L = n + 1 - L, B = T, b = d, m = 1;
        }
    }
    return C;
}

```

```
40. //矩阵快速幂求解数列单项值,s 为初值,C 为递推公式
41. ll MatrixSolve(const vector<ll> &s, const vector<ll> &C, ll n) {
42.     if (n < s.size())return s[n];
43.     ll k = C.size() - 1, b = n - k + 1;
44.     static ll B[N][N], t[N], a[N], c[N][N];
45.     rep(i, 0, k - 1)a[i] = s[i];
46.     rep(i, 0, k - 1)rep(j, 0, k - 1)B[i][j] = j == 0 ? ((C[i + 1] > 0) * mo - C[i +
1]) : (i == j - 1);
47.     while (b) {
48.         if (b & 1) {
49.             rep(i, 0, k - 1)t[i] = 0;
50.             rep(i, 0, k - 1)rep(j, 0, k - 1)t[i] = (t[i] + a[k - j - 1] * B[j][k -
i - 1]) % mo;
51.             rep(i, 0, k - 1)a[i] = t[i];
52.         }
53.         b >>= 1;
54.         rep(i, 0, k - 1)rep(j, 0, k - 1)c[i][j] = 0;
55.         rep(r, 0, k - 1)rep(j, 0, k - 1)if (B[r][j])rep(i, 0, k - 1)c[i][j] =
(c[i][j] + B[i][r] * B[r][j]) % mo;
56.         rep(i, 0, k - 1)rep(j, 0, k - 1)B[i][j] = c[i][j];
57.     }
58.     return a[k - 1];
59. }
60.
61. //更快的矩阵快速幂求解数列单项值,s 为初值,C 为递推公式
62. ll fastMatrixSolve(const vector<ll> &s, const vector<ll> &C, ll n) {
63.     if (n < s.size())return s[n];
64.     ll k = C.size() - 1, b = n - k + 1;
65.     static ll B[N][N][64], t[N], a[N], c[N][N], init_flag = 1;
66.     if (init_flag) {
67.         init_flag = 0;
68.         rep(i, 0, k - 1)rep(j, 0, k - 1)B[i][j][0] = j == 0 ? ((C[i + 1] > 0) * mo
- C[i + 1]) : (i == j - 1);
69.         for (ll it = 1; it < 64; it++) {
70.             rep(i, 0, k - 1)rep(j, 0, k - 1)B[i][j][it] = 0;
71.             rep(r, 0, k - 1)
72.                 rep(j, 0, k - 1)
73.                     if (B[r][j][it - 1])
74.                         rep(i, 0, k - 1)
75.                             B[i][j][it] = (B[i][j][it] + B[i][r][it - 1] *
B[r][j][it - 1]) % mo;
76.         }
77.     }
78.     rep(i, 0, k - 1)a[i] = s[i];
```

```
79.     ll it = 0;
80.     while (b) {
81.         if (b & 1) {
82.             rep(i, 0, k - 1)t[i] = 0;
83.             rep(i, 0, k - 1)rep(j, 0, k - 1)t[i] = (t[i] + a[k - j - 1] * B[j][k -
i - 1][it]) % mo;
84.             rep(i, 0, k - 1)a[i] = t[i];
85.         }
86.         b >>= 1;
87.         it++;
88.     }
89.     return a[k - 1];
90. }
91.
92. //牛顿多项式插值
93. class NewtonPoly {
94. public:
95.     ll f[N], d[N], x[N], n = 0;
96.
97.     void add(ll X, ll Y) {
98.         x[n] = X, f[n] = Y % mo;
99.         rep(i, 1, n)f[n - i] = (f[n - i + 1] - f[n - i]) % mo * fpow((x[n] - x[n -
i]) % mo, mo - 2) % mo;
100.        d[n++] = f[0];
101.    }
102.
103.    ll operator()(ll X) const {
104.        ll ans = 0, t = 1;
105.        rep(i, 0, n - 1)ans = (ans + d[i] * t) % mo, t = (X - x[i]) % mo * t % mo;
106.        return ans + mo * (ans < 0);
107.    }
108. };
109.
110. //检验线性递推数列的通项公式是否为多项式,C 为递推公式
111. bool polyCheck(const vector<ll> &C) {
112.     ll m = mo - C[1], last = 1;
113.     rep(i, 1, C.size() - 1) {
114.         last = last * (m - i + 1) % mo * fpow(i, mo - 2) % mo;
115.         if (last != (i & 1 ? (mo - C[i]) : C[i]))return false;
116.     }
117.     return true;
118. }
119.
120. //矩阵
```

```
121. class intMatrix {
122. public:
123.     ll a[N][N], n, m;
124.
125.     bool operator<(const intMatrix &b) const {
126.         rep(i, 0, n - 1)rep(j, 0, m - 1)if (a[i][j] != b.a[i][j])return a[i][j] <
b.a[i][j];
127.         return false;
128.     }
129.
130.     bool operator==(const intMatrix &b) const {
131.         return !(*this < b) && !(b < *this);
132.     }
133.
134.     static intMatrix Eye(ll n) {
135.         intMatrix c;
136.         c.n = c.m = n;
137.         rep(i, 0, n - 1)rep(j, 0, n - 1)c.a[i][j] = i == j;
138.         return c;
139.     }
140.
141.     intMatrix operator*(const intMatrix &b) const {
142.         assert(m == b.n);
143.         intMatrix c;
144.         c.n = n, c.m = b.m;
145.         rep(i, 0, c.n - 1)rep(j, 0, c.m - 1)c.a[i][j] = 0;
146.         rep(i, 0, c.n - 1)rep(j, 0, c.m - 1)rep(k, 0, m - 1)c.a[i][j] = (c.a[i][j]
+ a[i][k] * b.a[k][j]) % mo;
147.         return c;
148.     }
149.
150.     intMatrix operator^(int t) const {
151.         assert(n == m);
152.         intMatrix c = Eye(n), b = *this;
153.         while (t) {
154.             if (t & 1)c = c * b;
155.             b = b * b;
156.             t >>= 1;
157.         }
158.         return c;
159.     }
160.
161.     void print() const {
162.         rep(i, 0, n - 1)rep(j, 0, m - 1)cout << a[i][j] << " \n"[j == m - 1];
```

```
163.         cout << endl;
164.     }
165. };
166.
167. //求同余意义下不依赖初值的周期,C 为递推公式,d 为步长
168. ll periodSolve(const vector<ll> &C, ll d) {
169.     static map<intMatrix, ll> tab;
170.     tab.clear();
171.     ll k = C.size() - 1;
172.     assert(C[k] != 0);
173.     intMatrix A, B, T = intMatrix::Eye(k), D = T;
174.     A.n = A.m = B.n = B.m = k;
175.     rep(i, 0, k - 1)rep(j, 0, k - 1)A.a[i][j] = j == 0 ? ((C[i + 1] > 0) * mo - C[i
+ 1]) : (i == j - 1);
176.     rep(i, 0, k - 1)rep(j, 0, k - 1)B.a[i][j] = j == k - 1 ? ((mo - C[i]) *
fpow(C[k], mo - 2) % mo) : (i == j + 1);
177.     rep(i, 1, d) {
178.         T = T * B;
179.         if (T == D)return i;
180.         tab[T] = i;
181.     }
182.     intMatrix Ad = A ^d;
183.     ll x = 0;
184.     while (1) {
185.         if (tab.find(D) != tab.end())return x * d + tab[D];
186.         D = D * Ad;
187.         x++;
188.     }
189.     return -1;
190. }
191.
192. ll polySolve(const vector<ll> &s, const vector<ll> &C, ll n) {
193.     if (n < s.size())return s[n];
194.     static ll g[N], f[N], d[N];
195.     ll k = (ll) C.size() - 1, w = 1;
196.     rep(i, 0, k)f[i] = i == 1, d[i] = i == k ? 1 : C[k - i];
197.     while ((w << 1) <= n)w <<= 1;
198.     while (w >>= 1) {
199.         rep(i, 0, k + k - 2)g[i] = 0;
200.         rep(i, 0, k - 1)if (f[i])rep(j, 0, k - 1)(g[i + j] += f[i] * f[j]) %= mo;
201.         for (ll i = k + k - 2; i >= k; i--)if (g[i])rep(j, 1, k)(g[i - j] -= g[i] *
d[k - j]) %= mo;
202.         rep(i, 0, k - 1)f[i] = g[i];
203.         if (w & n)
```

```
204.             for (ll i = k; i >= 0; i--)
205.                 f[i] = i == k ? f[i - 1] : (i == 0 ? -f[k] * d[i] : (f[i - 1] -
f[k] * d[i])) % mo;
206.     }
207.     ll ans = 0;
208.     rep(i, 0, k - 1)(ans += f[i] * s[i]) %= mo;
209.     return ans + (ans < 0) * mo;
210. }
211.
212. class Sequence {
213. public:
214.     ll poly;
215.     vector<ll> A, C;
216.     NewtonPoly P;
217.
218.     void build(const vector<ll> &s) {
219.         A = s;
220.         C = BM(A);
221.         poly = polyCheck(C);
222.         if (poly)rep(i, 0, s.size() - 1)P.add(i, s[i]);
223.     }
224.
225.     ll operator()(ll n) const {
226.         return poly ? P(n) : polySolve(A, C, n);
227.     }
228.
229.     friend ostream &operator<<(ostream &o, const Sequence &b) {
230.         o << "f(n)";
231.         rep(i, 1, b.C.size() - 1)o << "+(" << b.C[i] << ")*f(n-" << i << ")";
232.         o << "=0 (mod " << mo << ")";
233.         return o;
234.     }
235.
236.     ll period() const {
237.         ll M = periodSolve(C, (ll) sqrt(mo)), ans = M;
238.         return M;
239.         for (ll i = 1; i * i <= M; i++)
240.             if (M % i == 0) {
241.                 ll d = i, flag = 1;
242.                 rep(i, 0, (ll) C.size() - 2)if ((*this)(d + i) != A[i])flag = 0;
243.                 if (flag)ans = min(ans, d);
244.                 d = M / i, flag = 1;
245.                 rep(i, 0, (ll) C.size() - 2)if ((*this)(d + i) != A[i])flag = 0;
246.                 if (flag)ans = min(ans, d);
```

```
            }
        return ans;
    }
} F;

int main() {
    ios::sync_with_stdio(false);
    F.build({1, 1, 2, 3, 5, 8, 13, 21});
    cout << F << endl;
    F.build({0,1,4,9,16,25,36,49});
    cout<<F<<endl;
    F.build({0,1,5,15,35,70,126,210,330,495,715});
    cout<<F<<endl;
    F.build({0,1,2,0,1,2,0,1,2});
    F.build({4,14,52,194,724,2702,10084,37634});
    cout<<F<<endl;
    return 0;
}
```

4.2.5.3 例题

斐波那契和（牛客 NC206026）

$Fib(i)$ 表示斐波那契函数，$Fib(n) = Fib(n-1) + Fib(n-2)$，如 $Fib(1) = 1$，$Fib(2) = 1$，$Fib(3) = 2$，$Fib(4) = 3$，$Fib(5) = 5$，$Fib(6) = 8$。

给定正整数n和k，求：

$$\sum_{i=1}^{n} i^k Fib(i)$$

由于结果太大，你需要把求和的结果对 998,244,353 取余。

【输入格式】

输入一行，包含两个整数$n,k(1 \le n \le 10^{18}，1 \le k \le 100)$

【输入格式】

输出一个整数，表示求和对 998,244,353 取余的结果。

【分析】

套用线性递推模板直接求解。

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

using namespace std;
#define pb push_back
```

```
typedef long long ll;
const int Mod = 998244353;
const int maxn = 2e5 + 10;

#define rep(i, a, n) for (int i = a; i < n; i++)
#define SZ(x) ((int)(x).size())
typedef vector<int> VI;
const int N = 1e5 + 10;

ll powMOD(ll a, ll b) {
    ll ans = 1;
    for (; b; b >>= 1, a = a * a % Mod)
        if (b & 1) ans = ans * a % Mod;
    return ans;
}

namespace linear_seq {
ll res[N], base[N], _c[N], _md[N];

vector<int> Md;
void mul(ll *a, ll *b, int k) {
    rep(i, 0, k + k) _c[i] = 0;
    rep(i, 0, k) if (a[i]) rep(j, 0, k) _c[i + j] = (_c[i + j] + a[i] * b[j]) %
Mod;
    for (int i = k + k - 1; i >= k; i--)
        if (_c[i])
            rep(j, 0, SZ(Md)) _c[i - k + Md[j]] = (_c[i - k + Md[j]] - _c[i] *
_md[Md[j]]) % Mod;
    rep(i, 0, k) a[i] = _c[i];
}
int solve(ll n, VI a, VI b) {
    ll ans = 0, pnt = 0;
    int k = SZ(a);
    assert(SZ(a) == SZ(b));
    rep(i, 0, k) _md[k - 1 - i] = -a[i];
```

```
38.     _md[k] = 1;
39.     Md.clear();
40.     rep(i, 0, k) if (_md[i] != 0) Md.push_back(i);
41.     rep(i, 0, k) res[i] = base[i] = 0;
42.     res[0] = 1;
43.     while ((1ll << pnt) <= n) pnt++;
44.     for (int p = pnt; p >= 0; p--) {
45.         mul(res, res, k);
46.         if ((n >> p) & 1) {
47.             for (int i = k - 1; i >= 0; i--) res[i + 1] = res[i];
48.             res[0] = 0;
49.             rep(j, 0, SZ(Md)) res[Md[j]] = (res[Md[j]] - res[k] * _md[Md[j]]) %
Mod;
50.         }
51.     }
52.     rep(i, 0, k) ans = (ans + res[i] * b[i]) % Mod;
53.     if (ans < 0) ans += Mod;
54.     return ans;
55. }
56. VI BM(VI s) {
57.     VI C(1, 1), B(1, 1);
58.     int L = 0, m = 1, b = 1;
59.     rep(n, 0, SZ(s)) {
60.         ll d = 0;
61.         rep(i, 0, L + 1) d = (d + (ll)C[i] * s[n - i]) % Mod;
62.         if (d == 0)
63.             ++m;
64.         else if (2 * L <= n) {
65.             VI T = C;
66.             ll c = Mod - d * powMOD(b, Mod - 2) % Mod;
67.             while (SZ(C) < SZ(B) + m) C.pb(0);
68.             rep(i, 0, SZ(B)) C[i + m] = (C[i + m] + c * B[i]) % Mod;
69.             L = n + 1 - L;
70.             B = T;
71.             b = d;
```

```
72.             m = 1;
73.         } else {
74.             ll c = Mod - d * powMOD(b, Mod - 2) % Mod;
75.             while (SZ(C) < SZ(B) + m) C.pb(0);
76.             rep(i, 0, SZ(B)) C[i + m] = (C[i + m] + c * B[i]) % Mod;
77.             ++m;
78.         }
79.     }
80.     return C;
81. }
82. int gao(VI &a, ll n) {
83.     VI c = BM(a);
84.     c.erase(c.begin());
85.     rep(i, 0, SZ(c)) c[i] = (Mod - c[i]) % Mod;
86.     return solve(n, c, VI(a.begin(), a.begin() + SZ(c)));
87. }
88. };   // namespace linear_seq
89.
90. ll f[maxn];
91.
92. void init() {
93.     f[1] = f[2] = 1;
94.     for (int i = 3; i < N; i++) f[i] = (f[i - 1] + f[i - 2]) % Mod;
95. }
96.
97. int main() {
98.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
99.     ll n, k;
100.    init();
101.    cin >> n >> k;
102.    vector<int> g(N);
103.    g[0] = 0;
104.    for (int i = 1; i < N; i++) g[i] = (g[i - 1] + powMOD(i, k) * f[i] % Mod) %
Mod;
105.    int ans = linear_seq::gao(g, n);
```

```
    cout << ans << "\n";
    return 0;
}

```

4.3 概率论

期望的线性性，条件概率

4.3.1 基本概念

在研究具体的随机现象时我们通常着重关注以下要素:

- 样本空间 Ω，指明随机现象所有可能出现的结果。
- 事件域 $\mathcal{F}$，表示我们所关心的所有事件。
- 概率 P，描述每一个事件发生的可能性大小。

1. 样本空间、随机事件

定义：一个随机现象中可能发生的不能再细分的结果被称为样本点。所有样本点的集合称为样本空间,通常用来表示。

一个随机事件是样本空间 Ω 的子集，它由若干样本点构成，用大写字母 A,B,C,...表示。

对于一个随机现象的结果 ω 和一个随机事件 A，我们称事件 A 发生了当且仅当 $\omega \in A$。

例如，掷一次骰子得到的点数是一个随机现象，其样本空间可以表示为 $\Omega = \{1,2,3,4,5,6\}$。设随机事件 A 为“获得的点数大于 4”，则 A ={5,6}。若某次掷骰子得到的点数 $\omega = 3$，由于$\omega \notin A$，故事件 A 没有发生。

2. 事件的运算

由于我们将随机事件定义为了样本空间几的子集，故我们可以将集合的运算(如交、并、补等)移植到随机事件上。记号与集合运算保持一致。

特别的，事件的并 $A \cup B$ 也可记作 $A + B$，事件的交 $A \cap B$ 也可记作AB，此时也可分别称作和事件和积事件。

4.3.2 条件概率与独立性

1. 条件概率

定义：若已知事件 A 发生，在此条件下事件 B 发生的概率称为条件概率，记作 $P(B|A)$。

在概率空间 $(\Omega, \mathcal{F}, P)$ 中，若事件 $A \in \mathcal{F}$ 满足$P(A) > 0$，则条件概率$P(A)$定义为：

$$P(B|A)=\frac{P(AB)}{P(A)},\forall B\in\mathcal{F}$$

可以验证根据上式定义出的$P(A)$是$(\Omega,\mathcal{F})$上的概率函数。

根据条件概率的定义可以直接推出下面两个等式：

- 概率乘法公式：在概率空间$(\Omega,\mathcal{F},P)$中，若$P(A)>0$，则对任意事件 B 都有$P(AB)=P(A)P(B|A)$。
- 全概率公式：在概率空间$(\Omega,\mathcal{F},P)$中，若一组事件 $A_1,\dots,A_n$ 两两不交且和为 Ω，则对任意事件 B 都有$P(B)=\sum_{i=1}^{n}P(A_i)P(B|A_i)$。

2. Bayes 公式

一般来说，设可能导致事件 B 发生的原因为 $A_1,\dots,A_n$，则在 $P(A_i)$ 和 $P(B|A_i)$ 已知时可以通过全概率公式计算事件 B 发生的概率。但在很多情况下，我们需要根据“事件 B 发生”这一结果反推其各个原因事件的发生概率。于是有：

$$P(A_i|B)=\frac{P(A_iB)}{P(B)}=\frac{P(A_i)P(B|A_i)}{\sum_{j=1}^{n}P(A_j)(B|A_j)}$$

上式即 Bayes 公式。

3. 事件的独立性

在研究条件概率的过程中，可能会出现 $P(B|A)=P(B)$ 的情况。从直观上讲就是事件 B 是否发生并不会告诉我们关于事件 A 的任何信息，即事件 B 与事件 A 无关。于是我们就有了下面的定义

定义：若同一概率空间中的事件 A,B 满足 $P(AB)=P(A)P(B)$，则称 A,B 独立。对于多个事件$A_1,\dots,A_n$，我们称其独立，当且仅当对任意一组事件$\{A_{ik}:1\le i_1<i_2<\dots<i_k\le n\}$都有

$$P(A_{i1}A_{i2}\dots A_{ir})=\prod_{k=1}^{r}P(A_{ik})$$

4. 多个事件的独立性

对于多个事件，一般不能从两两独立推出这些事件独立。考虑以下反例:

有一个正四面体骰子，其中三面被分别涂成红色、绿色、蓝色，另一面则三色皆有。现在扔一次该骰子，令事件 A,B,C 分别表示与桌面接触的一面包含红色、绿色、蓝色。

不难计算$P(A)=P(B)=P(C)=\frac{1}{2}$，而 $P(AB)=P(BC)=P(CA)=P(ABC)=\frac{1}{4}$。显然 A,B,C 两两独立，但由于$P(ABC)\neq P(A)P(B)P(C)$，故 A,B,C 不独立。

4.3.3 数学期望

4.3.3.1 随机变量

随机变量

给定概率空间 $(\Omega,\mathcal{F},P)$，定义在样本空间 Ω 上的函数 $X:\Omega\to\mathbb{R}$ 若满足:对任意 $t\in\mathbb{R}$都有 $\{\omega\in\Omega:X(\omega)\ \le t\}\in\mathcal{F}$

则称 X 为随机变量。

对于随机变量 X，称函数 $F(x)\ =\ P(X\le x)$ 为随机变量 X 的分布函数。记作 $X\sim F(x)$。

分布函数具有以下性质:

- 右连续性：$F(x)\ =F(x+0)$
- 单调性：在$\mathbb{R}$上单调递增(非严格)
- $F(-\infty)=\ 0,F(+\infty)=\ 1$

随机变量的分类

随机变量按其值域(根据定义，随机变量是一个函数)是否可数分为离散型和连续型两种。

（1）离散型随机变量

设X为离散型随机变量，其所有可能的取值为 $x_1,x_2,\ldots$，则我们可以用一系列形如 $P\{X=x_1\}=p_i$ 的等式来描述 X。这就是我们在高中课本中学过的分布列。

（2）连续型随机变量

设 X 为离散型随机变量，考察 $P\{X=\ x\}$ 往往是无意义的(因为这一概率很可能是 0)。

随机变量的独立性

前面讨论了随机事件的独立性。由于随机变量和随机事件紧密联系，我们还可以类似地给出随机变量独立性的定义。

定义：若随机变量 X,Y 满足对任意的 $x,y\in\mathbb{R}$ 都有 $P(X\le x,Y\le y)\ =P(X\le x)P(Y\le y)$，则称随机变量 X,Y 独立。

4.3.3.2 期望

这里主要讨论离散型随机变量的期望。

定义：若随机变量 X 的取值有 $x_1,x_2,\ldots$，一个随机事件可表示为 $X=x_i$，其概率为 $P\{X=x_1\}=p_i$，则称 $E(X)=\sum p_ix_i$ 为随机变量 X 的数学期望。通俗地讲，数学期望是随机变量取值与概率的乘积之和。

例如在掷两枚骰子的点数实验中，样本空间是由 36 个样本点构成的集合，每个样本点可写作 (a,b)，其中$1\le a,b\le 6$。随机变量有多种，不妨以“掷出的点数之和 X”为例，

则随机变量 X 的取值为 2~12。随机事件可描述为“掷出 X 点”，即由 $a+b=X$ 的样本点(a,b) 构成的子集。掷出 8 点的概率 $P(X=8)=5/36$。掷出的点数的数学期望为 7。

对于条件概率、方差以及各种常见的离散型随机变量的分布，读者在高中数学课本中应该已经学到，这里就不再赘述。

期望的性质：

（1）线性性

若随机变量 X,Y 的期望存在，则对任意实数 a,b，有：

- $E(aX+b)=aE(X)+b$
- $E(X+Y)=E(X)+E(Y)$

（2）随机变量乘积的期望

若随机变量 X,Y 的期望存在且 X,Y 相互独立，则有 $E(XY)=E(X)*E(Y)$

注意：上述性质中的独立性并非必要条件。

有了期望的性质，计算上述“掷两个骰子的点数”的数学期望将会非常容易。随机变量 X 表示掷一枚骰子的点数,显然期望值 $E(X)=(1+2+3+4+5+6)/6=3.5$。掷两枚骰子的点数可表示为随机变量 2X，于是 $E(2X)=2*E(X)=2*3.5=7$。

绿豆蛙的归宿（AcWing 217）

给出一个有向无环的连通图，起点为 1，终点为 N，每条边都有一个长度。

数据保证从起点出发能够到达图中所有的点，图中所有的点也都能够到达终点。

绿豆蛙从起点出发，走向终点。

到达每一个顶点时，如果有 K 条离开该点的道路，绿豆蛙可以选择任意一条道路离开该点，并且走向每条路的概率为 $1/K$。

现在绿豆蛙想知道，从起点走到终点所经过的路径总长度的期望是多少？

【输入格式】

第一行: 两个整数 N,M，代表图中有 N 个点、M 条边。

第二行到第 $1+M$ 行: 每行 3 个整数 a,b,c，代表从 a 到 b 有一条长度为 c 的有向边。

【输入格式】

输出从起点到终点路径总长度的期望值，结果四舍五入保留两位小数。

【分析】

根据概率的线性性。一般起点是唯一的，终点不唯一，我们可以从终点往起点开始逆推，设$f[i]$表示从i到终点的期望，则$f[i]=\sum 1/k(w[i]+f[j])$其中$f(N)=0$，$res=f(1)$，使用记忆化搜索就可以实现从终点一直逆推到起点，我们可以使用一个$dout$数组来统计每一个点的出度，这样后面在递归的时候可以计算出当前点往下递归的时候$1/k$的值。

```
1. #include <bits/stdc++.h>
2.
3. using namespace std;
```

```cpp
const int N = 100010, M = 200010;
int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dout[N];
double f[N];

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
}

double dp(int u) {
    if (f[u] >= 0) return f[u];
    f[u] = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        f[u] += (w[i] + dp(j)) / dout[u];
    }
    return f[u];
}

int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c);
        dout[a]++;
    }

    memset(f, -1, sizeof f);
    printf("%.2lf\n", dp(1));
    return 0;
}

```

Fireworks（牛客 NC216006）

做烟花，做一个需要的时间是n，这一个能被成功点燃的概率是$p*10^{-4}$，点燃一次需要m的时间，问至少有一个烟花被成功点燃的最小期望时间。

【输入格式】

第一行输入一个正整数 T，代表测试用例的数量。

接下来 T 行，每行包括三个正整数$n,m,p(1\le n,m\le 10^9,1\le p\le 10^4)$。

【输入格式】

输出 T 行，每行一个浮点数代表最小期望时间，误差精度要求不超过10^{-4}。

【分析】

先制作x个，接着点燃一次。设成功点燃的概率是p，那么不成功的概率是$1-p$，连续x个都不成功的概率是$(1-p)^x$所以至少有一个被点燃的概率就是$1-(1-p)^x$。

我们所需要的总时间是$x*n+m$，那么期望就是$\frac{x*n+m}{1-(1-p)^x}$（总时间除以概率），s 其中x为变量，其他的n,m,p都是定值。

对上述方程求导，根据数学知识得知该函数是一个凹函数，因此使用三分求解最小值即可。

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

#define double long double
#define int long long
#define endl "\n"
const int max_n = 1e5 + 100;
const int inf = 0x3f3f3f3f;
const int mod = 1e9 + 7;

int n, m, p;
double qm(double b, int po) {
    double res = 1;
    while (po) {
        if (po & 1) res = res * b;
        po >>= 1;
        b = b * b;
    }
    return res;
}
```

```
21.
22. double get(int x) {
23.     return (n * x + m) * 1.0 / (1 - qm((1 - (p * 1e-4)), x));
24. }
25.
26. signed main() {
27.     ios::sync_with_stdio(false);
28.     cin.tie(0);
29.     cout.tie(0);
30.
31.     int T;
32.     cin >> T;
33.     while (T--) {
34.         cin >> n >> m >> p;
35.         double ans = inf;
36.         int l = 0, r = inf;
37.         int cnt = 100;
38.         while (cnt--) {
39.             int m1 = (l * 2 + r) / 3, m2 = (l + r * 2) / 3;
40.             double t1 = get(m1), t2 = get(m2);
41.             if (t1 >= t2)
42.                 l = m1;
43.             else
44.                 r = m2;
45.             ans = min(t1, t2);
46.         }
47.         for (int i = l; i <= r; i++) ans = min(ans, get(i));
48.         cout << fixed << ans << endl;
49.     }
50.
51.     return 0;
52. }
53.
```

4.4 组合数学

4.4.1 组合数

组合数公式

$$C_n^m = \frac{n!}{m!\,(n-m)!}$$

考虑递推求解 $C_n^m = \frac{n!}{m!(n-m)!}$，$C_n^{m-1} = \frac{n!}{(m-1)!(n-m+1)!} = \frac{m}{n-m+1} * C_n^m$，即：

$$C_n^m = \frac{n-m+1}{m} * C_n^{m-1}$$

那么从C_n^0开始递推即可：

```
1. ll cal(int n, int m) {
2.     ll ans = 1;
3.     for (int i = 1; i <= m; i++) ans = ans * (n - i + 1) / i;
4.     return ans;
5. }
```

组合数也常用 $\binom{n}{m}$ 表示，读作 n 选 m，即 $C_n^m = \binom{n}{m}$。实际上，后者表意清晰明了，美观简洁，因此现在数学界普遍采用 $\binom{n}{m}$ 的表示形式。

杨辉三角

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

当数据范围允许时，直接使用杨辉三角打表，思路:

$$2^n = C_n^0 + C_n^1 + C_n^2 + \ldots + C_n^{n-1} + C_n^n$$

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$$

```
1. ll C[1005][1005];
2. const int Mod = 1e9 + 7;
3.
4. void cal() {
5.     C[0][0] = 1;
6.     for (int i = 1; i <= 1000; i++) {
```

```
        C[i][0] = C[i][i] = 1;
        for (int j = 1; j <= (i >> 1); j++) {
            C[i][j] = C[i][i - j] = (C[i - 1][j - 1] + C[i - 1][j]) % Mod;
        }
    }
}
```

杨辉三角形（AcWing 3418）

如果我们把杨辉三角形按从上到下、从左到右的顺序把所有数排成一列，可以得到如下数列：1, 1, 1, 1, 2, 1, 1, 3, 3, 1, 1, 4, 6, 4, 1, ...给定一个正整数 N，请你输出数列中第一次出现 N 是在第几个数？

【输入格式】

输入一个正整数 N。

【输入格式】

输出一个整数表示答案。

【分析】

本题的切入点为n的范围——n一定是一个小于 1e9 的数，对杨辉三角大概前五十项打表，实际上到三十几项时，中间的数已经超过 1e9，然后再次联想到$\binom{10000}{2}$大概也就到 1e9，因此我们只需要保存杨辉三角表中小于 1e9 的数，这些数的个数是很少的完全存的下，而且不难证明第n行的数的个数一定小于等于第$n-1$行，那么递推时大于 1e9 的数不保存也不会影响下面的递推。对于表里面找不到的数，一定是$\binom{n}{1}$第一次出现，那么答案就是$1+2+3+\ldots+n+2$。

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

using namespace std;
#define ENDL "\n"
typedef long long ll;
const int maxn = 2e4 + 10;
const int limit = 1e9;

ll c[maxn][50];
int len[maxn];

void init() {
    c[0][0] = c[1][0] = c[1][1] = 1;
    len[0] = 1, len[1] = 2;
    for (int i = 2, k; i < maxn; i++) {
```

```
16.         c[i][0] = 1, k = i + 1;
17.         for (int j = 1; j <= i; j++) {
18.             c[i][j] = c[i - 1][j - 1] + c[i - 1][j];
19.             if (c[i][j] > limit) {
20.                 k = j;
21.                 break;
22.             }
23.         }
24.         len[i] = k;
25.     }
26. }
27.
28. ll cal(int n) {
29.     for (int i = 0; i < maxn; i++) {
30.         for (int j = 0; j < len[i]; j++) {
31.             if (c[i][j] == n) {
32.                 return 1LL * i * (i + 1) / 2 + j + 1;
33.             }
34.         }
35.     }
36.     return 1LL * n * (n + 1) / 2 + 2;
37. }
38.
39. int main() {
40.     init();
41.     int n;
42.     cin >> n;
43.     cout << cal(n) << endl;
44.     return 0;
45. }
46.
```

4.4.2 组合数学基本定理

4.4.2.1 加法&乘法原理

1. 加法原理

完成一件工作有 n 类方法，第 i 种方法有 m_i 种不同的方法，完成这件工作共有 $n = \sum_{i=1}^{n} m_i$ 种方法。

2. 乘法原理

完成一件工作共需 n 个步骤，第 i 个步骤有 m_i 种方法，完成这件工作共需 $\prod_{i=1}^{n} m_i$ 种方法。

4.4.2.2 二项式定理

二项式定理描述了二项式的幂的代数展开。根据该定理，可以将两个数之和的整数次幂诸如 $(x+y)^n$ 展开为类似 ax^by^c 项之和的恒等式，其中 b,c 均为非负整数且 $b+c=n$。系数 a 是依赖于 n 和 b 的正整数。当某项的指数为 0 时，通常略去不写。二项式定理的定义如下：

$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i}b^i$。

例如：

$$(a+b)^1 = a+b$$

$$(a+b)^2 = a^2+2ab+b^2$$

$$(a+b)^3 = a^3+3a^2b+3ab^2+b^3$$

$$(a+b)^4 = a^4+4a^3b+6a^2b^2+4ab^3+b^4$$

二项式展开式中的系数就是组合数。比如：a^2b 的系数，其实就是在三个因数中选一个b 的可能性，或者说选两个a的可能性，都是 3 种；ab^2的系数，其实就是在三个因数中选一个a的可能性，或者说选两个b的可能性，也是 3 种；a^3或者b^3其实就是三个因数全是a或者全是b的可能性，也就是 1 种。联系到杨辉三角，我们发现 $(a+b)^n$ 的展开式系数刚好对应于杨辉三角的第 n 层。

4.4.2.3 鸽巢定理

鸽巢定理又称为抽屉定理，它常被用于证明存在性证明和求最坏情况下的解。

1. 定理一

把 $m(m \geq n+1)$ 件物品放入 n 个抽屉里，则至少有一个抽屉里有两种物品。

2. 定理二

把 $x(x \geq n*m+1)$ 件物品放入 n 个抽屉，至少有一个抽屉里有不少于两件物品。

3. 定理三

把无穷物品放入 n 个抽屉至少有一个抽屉里有无穷个物品。

4. 应用

（1）如何证明：从 1 到 20 这 20 个数任取 11 个数，必有两个数，其中一个数是另外一个数的倍数。

证明：11 个数是物品，抽屉应该按照“同一抽屉中任意两个数都具有倍数关系”的原则，那么最多可以分为十个抽屉：$\{1,2,4,8,16\},\{3,6,12\},\{5,10,20\},\{7,14\},\{9,18\},\{11\},\{13\},\{15\},\{17\},\{19\}$。显然任取 11 个数，一定满足必有两数是倍数关系。

扩展：给出正整数 n，求出 $1-n$ 中至少选择多少个数使得必存在两个数是倍数关系？按照上面的构造方式，答案是 $(n>>1)+(n\&1)+1$。

（2）有一个 3 行十列的表格，每个小方格可以涂上两种颜色，试证明无论怎样涂，至少两列法涂色方法相同。

证明：每列有三格，显然有 $2^3=8$ 种涂法，这 8 种涂法就相当于是抽屉，10 列相当于物品，由抽屉原理可知显然成立。

（3）问题：对任意的五个数，证明其中必有三个数的和能被3整除。

证明：模 3 的剩余系为 $\{0,1,2\}$ 为抽屉，五个数为物品，五个数分别对三取模，然后分类讨论：

- 若五个余数分布在其中的三个抽屉中，只需各取一个。
- 若五个余数分布在其中的两个抽屉中，其中必有一个抽屉至少包含三个数，那么这三个数的和也一定是三的倍数。
- 若五个余数分布在其中的 1 个抽屉中，由上面知显然成立。

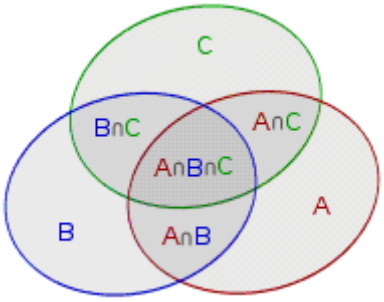
4.4.2.4 容斥定理

1. 容斥定理

问题引入：给出一个班上的总人数 n，有三项活动大家选择参加。参与某项活动的参与人数 x_i，给出同时参与两项活动的人数 y_i，求三项都参与的人数 z。

根据题意可知 $\sum_{i=1}^{n}x_i-\sum_{i=1}^{n}y_i+z=n$。

显然对于上式来看就是奇数次加上，偶数次减去。用 A、B、C 分别表示三个活动参与学生的集合，那么可以表示为如下图所示：



定义：设 U 中元素有 n 种不同的属性，而第 i 种属性称为 P_i，拥有属性 P_i 的元素构成集合 S_i，那么：

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots + (-1)^{m-1} \sum_{a_i<a_{i+1}} \left|\bigcap_{i=1}^{m} S_{a_i}\right| + \cdots$$
$$+ (-1)^{n-1} |S_1 \cap S_2 \cap \ldots \cap S_n|$$

2. 容斥定理应用

问题：给出 n，求出 $[1,n]$ 中不是 2,3,5 的倍数的个数。

证明：$\lfloor \frac{n}{2} \rfloor$ 求出的是 n 以内 2 的倍数，同理 3,5，然后我们对三个集合求交集，减去两个数重复的部分，再加上三个数重复的部分。根据容斥定理，答案为：$n - \left[\left(\lfloor \frac{n}{2} \rfloor\right) + \lfloor \frac{n}{3} \rfloor + \lfloor \frac{n}{5} \rfloor\right) - \left(\lfloor \frac{n}{6} \rfloor + \lfloor \frac{n}{10} \rfloor + \lfloor \frac{n}{15} \rfloor\right) + \lfloor \frac{n}{30} \rfloor\Big]$。

4.4.2.5 多重集的排列组合

设 $S = \{n_1 * a_1, n_2 * a_2, n_3 * a_3, \ldots, n_k * a_k\}$ 是由 n_1 个 a_1、n_2 个 a_2、…、n_k 个 a_k 组成的集合。

1. 多重集的排列数

设 $n = n_1 + n_2 + n_3 + \ldots + n_k$，那么全排列的个数为 $\frac{n!}{n_1! n_2! \cdots n_k!}$

2. 多重集的组合数

（1）定理一：从多重集 S 中取出 $r(r \le n_i, \forall i \in [1,k])$ 个数构成一个多重集的方案个数为 $C_{k+r-1}^{k-1} = C_{k+r-1}^{r}$

证明：因为 $r \le n_i$，所以对于该问题可以等价于有 r 个小球以及 $k-1$ 个隔板，这 $r+k-1$ 个物品排列后隔板将小球分成 $[1,k]$ 份，这个方案数实际上就是我们要求的组合数，即 $\frac{(k+r-1)!}{r!(k-1)!} = C_{k+r-1}^{r} = C_{k+r-1}^{k-1}$

（2）定理二：从多重集 S 中取出 $r(r \le n)$ 个数构成一个多重集的方案个数为 $C_{k+r-1}^{k-1} - \sum_{i=1}^{k} C_{k+r-n_i-2}^{k-1} + \sum_{1 \le i < j \le k} C_{k+r-n_i-n_j-3}^{k-1} - \cdots + (-1)^k C_{k+r-\sum_{i=1}^{k} n_i-(k+1)}^{k-1}$

证明：不考虑 n_i 的限制，即从集合 $S'=\{\infty*a_1,\infty*a_2,\ldots,\infty*a_k\}$取出 r 个元素，根据定理一可以得到方案数为 C_{k+r-1}^{k-1}。但是这个方案数可能是偏大的，因为可能对于某种元素 a_i 我们取出了 $x>n_i$ 个元素，考虑减去每个元素的这一部分：设对于 a_i 已经取出了 n_i+1 个元素，还需要从 S' 中取出 $r-n_i-1$ 个元素，这样就满足了仅对于 a_i 非法取多的情况，同理对于所有的 $a_i(1\le i\le k)$，那么此时计算得到的方案数为 $C_{k+r-1}^{k-1}-\sum_{i=1}^{k}C_{k+r-n_i-2}^{k-1}$。但是这不是答案，因为如果在考虑拿至少 n_i+1 个 a_i 后，也同样计算了至少拿 n_j+1 个 a_j 的情况，那么就多减了每两种情况的交。

4.4.2.6 例题

Co-prime（HDU 4135）

求 $[l,r]$中和n互质的数的个数

【输入格式】

第一行输入一个正整数 T 表示测试用例的数量。

接下来 T 行每行包含三个正整数 $l,r,n(1\le l\le r\le 10^{15},1\le n\le 10^9)$

【输入格式】

输出共 T 行，每行输出一个整数表示答案。

【分析】

根据容斥定理问题可以转化为$[1,x]$中和n互质的个数，然后 $f(r)-f(l-1)$即可。但是互质的数不好求，问题又变成了求出不互质的数的个数，然后拿n减去就得到了互质的个数。

x,y不互质等价于$gcd(x,y)\neq 1$，考虑最大公约数在质因数分解下的意义，也就是考虑n的每一种质数的组合，求出n以内有多少个它的倍数。但是这样会重复计算。假设$n=2^x*3^y$，我们已经统计了n以内2的倍数的个数$[\frac{n}{2}]$，然后再求n以内3的倍数的个数$[\frac{n}{3}]$，显然前者和后者有重复的统计部分，这个重复的部分就是n以内$2*3$的倍数的个数，因此我们考虑所有的组合状态，若素数种类数为奇数就加上统计的部分，否则减去统计的部分。

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

using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;

vector<ll> prime;

void init(ll n) {
```

```
10.     prime.clear();
11.     for (ll i = 2; i * i <= n; i++) {
12.         if (n % i == 0) {
13.             prime.push_back(i);
14.             while (n % i == 0) n /= i;
15.         }
16.     }
17.     if (n > 1) prime.push_back(n);
18. }
19.
20. ll solve(ll r) {
21.     ll ans = 0;
22.     for (ll s = 1; s < (1 << prime.size()); s++) {
23.         ll num = 1, cnt = 0;
24.         for (int i = 0; i < prime.size(); i++) {
25.             if (s & (1 << i)) {
26.                 cnt++;
27.                 num *= prime[i];
28.             }
29.         }
30.         ll res = r / num;
31.         if (cnt & 1)
32.             ans += res;
33.         else
34.             ans -= res;
35.     }
36.     return r - ans;
37. }
38.
39. int main() {
40.     ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
41.     int t, kase = 0;
42.     ll l, r, n;
43.     cin >> t;
44.     while (t--) {
```

```
45.         cin >> l >> r >> n;
46.         init(n);
47.         cout << "Case #" << ++kase << ": " << solve(r) - solve(l - 1) << "\n";
48.     }
49.     return 0;
50. }
51.
```

4.4.5 康托展开

1. 康托展开

康托展开可以用来求一个 $1\sim n$ 的任意排列的排名。排列的排名指的是，把 $1\sim n$ 的所有排列按字典序排序，排完序的位次就是它的排名，很明显 $[1,2,\ldots,n]$ 的排名为 1，$[n,n-1,\ldots,1]$ 的排名为 $n!$。

康托展开可以在 $O(n^2)$ 的时间复杂度内求出一个排列的排名，在用树状数组优化时可以做到 $O(n\log n)$。

思路：因为排列是按字典序排名的，因此越靠前的数字优先级越高。也就是说如果两个排列的某一位之前的数字都相同，那么如果这一位如果不相同，就按这一位排序。比如 4 的排列，$[2,3,1,4]<[2,3,4,1]$，因为在第 3 位出现不同，则 [2,3,1,4] 的排名在 [2,3,4,1] 前面。

我们知道长为 5 的排列 [2,5,3,4,1] 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 4! 种。这是非常好理解的。但是我们对第二位的 5 而言，它大于 第一位与这个排列相同的，而这一位比 5 小的所有排列。不过我们要注意的是，这一位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了。因此这一位为 1,3 或 4，第一位为 2 的所有排列都比它要小，数量为 $3\times 3!$。

按照这样统计下去，答案就是 $1+4!+3\times 3!+2!+1=46$。注意我们统计的是排名，因此最前面要加一。

注意到我们每次要用到**当前有多少个小于它的数还没有出现**，这里用树状数组统计比它小的数出现过的次数就可以了。

2. 逆康托展开

因为排列的排名和排列是一一对应的，所以康托展开满足双射关系，是可逆的。可以通过类似上面的过程倒推回来。

如果我们知道一个排列的排名 x，就可以推出这个排列。因为 4! 是严格大于 $3\times 3!+2!+1!$ 的，所以可以认为对于长度为 5 的排列，排名 x 除以 4! 向下取整就是有多少个数小于这个排列的第一位，以此类推。

引用上面展开的例子：首先让 $46-1=45$，45 代表着有多少个排列比这个排列小。$\left\lfloor\frac{45}{4!}\right\rfloor=1$，有一个数小于它，所以第一位是 2。此时让排名减去 $1\times4!$ 得到 21，$\left\lfloor\frac{21}{3!}\right\rfloor$，有 3 个数小于它，去掉已经存在的 2，这一位是 5。$21-3\times3!=3$，$\left\lfloor\frac{3}{2!}\right\rfloor=1$，有一个数小于它，那么这一位就是 3。让 $3-1\times2!=1$，有一个数小于它，这一位是剩下来的第二位，4，剩下一位就是 1。即 [2,5,3,4,1]。

实际上我们得到了形如**有两个数小于它**这一结论，就知道它是当前第 3 个没有被选上的数，这里也可以用线段树维护，时间复杂度为 $O(n\log n)$。

[USACO11FEB]Cow Line S（洛谷 3014）

$N(1\le N\le20)$头牛，编号为$1\ldots N$，正在与 FJ 玩一个疯狂的游戏。奶牛会排成一行（牛线），问 FJ 此时的行号是多少。之后，FJ 会给牛一个行号，牛必须按照新行号排列成线。

行号是通过以字典序对行的所有排列进行编号来分配的。比如说：FJ 有 5 头牛，让他们排为行号 3，排列顺序为：

1：1 2 3 4 5

2：1 2 3 5 4

3：1 2 4 3 5

因此，牛将在牛线 1 2 4 3 5 中。

之后，奶牛排列为“1 2 5 3 4”，并向 FJ 问他们的行号。继续列表：

4：1 2 4 5 3

5：1 2 5 3 4

FJ 可以看到这里的答案是 5。

FJ 和奶牛希望你的帮助玩他们的游戏。他们需要$K(1\le K\le10000)$组查询，查询有两个部分：C_i 将是“P”或“Q”的命令。

如果C_i是 'P'，则查询的第二部分将是一个整数$A_i(1\le A_i\le N!)$，它是行号。此时，你需要回答正确的牛线。

如果C_i是'Q'，则查询的第二部分将是 N 个不同的整数$B_{ij}(1\le B_{ij}\le N)$。这将表示一条牛线，此时你需要输出正确的行号。

【分析】

康托展开模板题。

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. const int maxn = 50;
4. long long n, k, a[maxn], ans, sum[maxn], b[maxn], y;
5. void get() {            // 正向展开
```

```
    long long fj = 1;   // fj 是阶乘之积
    long long ki = 0;
    for (long long i = 1; i <= n; i++) {
        for (long long j = i + 1; j <= n; j++) {
            fj *= (n - j + 1);
            if (a[i] > a[j]) ki++;   // 求 a[i]后面有多少个比 a[i]大得数
        }
        ans += (ki * fj);   // 加上每一位的值
        ki = 0;             // 清楚为初始值
        fj = 1;             // 同上
    }
}
void get2() {           // 反向求
    long long ct = 0;   // ct 表示当前以求得个数
    long long m = 0;    // m 为所得的商
    long long p = 0;    // p 为余数
    m = (y / sum[1]);
    p = (y % sum[1]);
    for (long long i = 1; i <= n;
         i++) {   // 如果有与这个商相同的就是这一位，如果不清楚为什么，看一下大佬们讲解的逆展开
        if (b[i] == m) {
            a[++ct] = i;
            break;
        }
    }
    for (long long i = 1; i <= n; i++) {   // 比这个数大的所有数都少了一个比本身小的数
        if (i >= a[ct]) b[i]--;
    }
    while (ct < n - 1) {   // 开始每一位求数，求不到最后一位
        ct++;
        m = p / sum[ct];
        p = p % sum[ct];
        for (long long i = 1; i <= n; i++) {
```

```
            if (b[i] == m) {
                a[ct] = i;
                break;
            }
        }
        for (long long i = 1; i <= n; i++) {
            if (i >= a[ct]) b[i]--;
        }
    }
}
int main() {
    scanf("%d%d", &n, &k);
    for (long long i = 1; i <= n; i++) {
        sum[i] = 1;
        b[i] = i - 1;
        for (long long j = i + 1; j <= n; j++) {
            sum[i] *= (j - i);   // sum 相当是每一位的阶层，我这里没有用
        }
    }
    sum[n] = 0;
    for (long long i = 1; i <= k; i++) {
        char opt;
        scanf("%c", &opt);
        while (opt == ' ' || opt == '\n') scanf("%c", &opt);
        if (opt == 'P') {
            scanf("%lld", &y);
            y -= 1;
            for (long long i = 1; i <= n; i++) {
                b[i] = i - 1;
            }
            get2();
            for (long long i = 1; i <= n - 1; i++) {
                printf("%lld ", a[i]);
            }
            for (
```

```
74.                 long long i = 1; i <= n;
75.                 i++) {    // 特别输出最后一位，因为之前所有的数，在 b 的变化中，最终的值都会为-1，只有最好用一个数的 b 值为 0，因此，找到这个数输出来。
76.                 if (b[i] == 0) {
77.                     printf("%lld", i);
78.                     break;
79.                 }
80.             }
81.             cout << endl;
82.         } else if (opt == 'Q') {
83.             for (long long i = 1; i <= n; i++) {
84.                 scanf("%lld", &a[i]);
85.             }
86.             ans = 1;
87.             get();
88.             printf("%lld\n", ans);
89.         }
90.     }
91.     return 0;
92. }
93.
```


4.4.6 常见数列

OEIS 是一个在线的数列百科网站，在这里几乎能找到所有已发现的数列，只需要输入数列的前几项，即可找到数列的包括通项公式、性质、推导公式等在内的详细信息。

本节将学习斐波那契数列、错位排列、卡特兰数、拆分数、斯特林数、贝尔数、伯努利数。

4.4.6.1 斐波那契数列

斐波那契数列形式如下：

$$F(n)=\begin{cases}1 & n=1\\ 1 & n=2\\ F(n-1)+F(n-2) & n\geq 3\end{cases}$$

4.4.6.1.1 斐波那契数列的求法

1. 递归求法

时间复杂度$O(2^n)$

```
1. int Fibonacci(int n){
2.     return n > 2 ? Fibonacci(n - 1) + Fibonacci(n - 2) : 1;
3. }
```

2. 线性求法

```
1. int F[maxn];
2. int num;
3. void Fibonacci() {
4.     F[1] = F[2] = 1;
5.     for (int i = 3; i <= num; i++) {
6.         F[i] = F[i - 1] + F[i - 2];
7.     }
8. }
```

3. 公式求法

斐波那契数列的通项公式为:

$$F(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

对于斐波那契数列这样的二阶线性递推数列，可采用**特征方程法**：

对于递推公式为 $x_n=ax_{n-1}+bx_{n-2}$ 且 a,b,x_1,x_2 已知的递推数列，尝试待定系数法，验证 $\{x_n-kx_{n-1}\}$ 是否为等比数列。设存在 r,s 使得 $x_n-rx_{n-1}=s(x_{n-1}-rx_{n-2})$。所以 $x_n=(s+r)x_{n-1}-srx_{n-2}$，那么可以得到 $a=s+r,b=-sr$，根据韦达定理（$x_1+x_2=-\frac{b}{a},x_1x_2=\frac{c}{a}$），得到 s,r 满足方程：$y^2-ay-b=0$。因此 $\{x_n-rx_{n-1}\},\{x_n-sx_{n-1}\}$ 均是等比数列，分别列出等比数列的通项公式然后作差即可得到 x_n。

如下是结论：

- 若方程有两相异根 p、q，则：

$$x_n=Ap^n+Bq^n,A=\frac{x_2-qx_1}{p(p-q)},B=\frac{px_1-x_2}{q(p-q)}$$

- 若方程有两等根 p，则：

$$x_n=(A+Bn)p^n,A=\frac{2px_1-x_2}{p^2},B=\frac{x_2-px_1}{p^2}$$

对于斐波那契数列的特征方程 $r^2-r-1=0$，解得两不等根$p=\frac{1+\sqrt{5}}{2},q=\frac{1-\sqrt{5}}{2}$，可以直接代入上式求解 A,B，也可以根据 x_1,x_2 已知直接利用结论，联立如下两个方程 $x_1=Ap+Bq,x_2=Ap^2+Bq^2$ 求解 A,B。

最后我们得出其通项公式：$F(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

由于 n 增大时涉及大量浮点运算，会导致精度损失，因此无法通过浮点数计算斐波那契数列。

但是通项公式还是可用的，只是要结合模意义下的二次剩余使用，将在后面章节讲到。

4. 矩阵快速幂

已经在 4.2.3 矩阵快速幂作为示例讲解。

5. 快速倍增法

斐波那契数列有如下性质：

$$F_{2n}=F_n(2F_{n+1}-F_n)$$

$$F_{2n+1}=F_{n+1}^2+F_n^2$$

那么可以通过递归增的方法求解二元组 $\{F(n),F(n+1)\}$。

```
1. typedef long long ll;
2. typedef pair<ll, ll> pii;
3. const int Mod = 1e9 + 7;
4.
5. pii fib(ll n) {
6.     if (n == 0) return {0, 1};
7.     auto p = fib(n >> 1);
8.     ll c = (2 * p.second % Mod - p.first + Mod) % Mod * p.first % Mod;
9.     ll d = (p.first * p.first % Mod + p.second * p.second % Mod) % Mod;
10.    if (n & 1) return {d, c + d};
11.    else return {c, d};
12. }
```


4.4.6.1.2 斐波那契数列的性质

1. 设 $f[i]$ 表示斐波那契数列的第 i 项，从第 0 项开始任何一个正整数 n 都能被若干个不同的斐波那契数的和表达，且该表示方法唯一，贪心构造即可。
2. 相邻两项互质：$gcd(f[n],f[n-1])=1$
3. 前缀和公式：$\sum_{i=0}^{n}f[i]=f[n+2]-1$
   a) 奇数项前缀和公式：$f[1]+f[3]+f[5]+\ldots+f[2n-1]=f[2n]-1$
   b) 偶数项前缀和公式：$f[2]+f[4]+f[6]+\ldots+f[2n]=f[2n+1]-1$
4. 平方前缀和公式：$f[0]^2+f[1]^2+..+f[n]^2=f[n]*f[n+1]$
   a) 单项平方公式：$f[n]^2=f[n-1]*f[n+1]+(-1)^{n-1}$
5. 有关 gcd 的重要结论 $gcd(f[n],f[m])=f[gcd(n,m)]$
6. 若 $x|f[k]$，那么 $x|f[k*i],i\in Z_+$
7. $f[n+m]=f[n+1]*f[m]+f[n]*f[m-1]$

8. $f[2n-1]=f[n]^2-f[n-2]^2$
9. $3*f[n]=f[n+1]+f[n-2]$

4.4.6.2 错位排列

1. 定义

错位排列(derangement)是没有任何元素出现在其有序位置的排列。即，对于$1\sim n$的排列P，如果满足$P_i\neq i$，则称 P 是 n 的错位排列。

例如，三元错位排列有{2,3,1}和{3,1,2}。四元错位排列有{2,1,4,3}、{2,3,4,1}、{2,4,1,3}、{3,1,4,2}、{3,4,1,2}、{3,4,2,1}、{4,1,2,3}、{4,3,1,2}和{4,3,2,1}。错位排列是没有不动点的排列，即没有长度为 1 的循环。

2. 容斥原理的计算

全集 U 即为$1\sim n$的排列，$|U|=n!$；属性就是$P_i\neq i$。套用补集的公式，问题变成求$|\cup_{i=1}^{n}\overline{S_i}|$。

可以知道，$\overline{S_i}$ 的含义是满足 $P_i=i$ 的排列的数量。用容斥原理把问题式子展开，需要对若干个特定的集合的交集求大小，即:

$$\left|\bigcap_{i=1}^{k}S_{ai}\right|$$

其中省略了$a_i<a_{i+1}$的条件以方便表示。上述 k 个集合的交集表示有k个变量满足$P_{ai}=a_i$的排列数，而剩下$n-k$个数的位置任意，因此排列数：

$$\left|\bigcap_{i=1}^{k}S_{ai}\right|=(n-k)!$$

那么选择 k 个元素的方案数为 $\binom{n}{k}$，因此有:

$$\left|\bigcup_{i=1}^{k}\overline{S_i}\right|=\sum_{k=1}^{n}(-1)^{k-1}\sum_{a_{1,\dots,k}}\left|\bigcap_{i=1}^{k}S_{ai}\right|$$

$$=\sum_{k=1}^{n}(-1)^{k-1}\binom{n}{k}(n-k)!$$

$$=\sum_{k=1}^{n}(-1)^{k-1}\frac{n!}{k!}$$

$$=n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!}$$

因此 n 的错位排列数为：

$$D_n = n! - n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!} = n!\sum_{k=0}^{n}\frac{(-1)^{k-1}}{k!}$$

错位排列数列的前几项为 0,1,2,9,44,265（OEIS A000166）。

3. 递推的计算

把错位排列问题具体化，考虑这样一个问题：n封不同的信，编号分别是1,2,3,4,5，现在要把这五封信放在编号1,2,3,4,5的信封中，要求信封的编号与信的编号不一样。问有多少种不同的放置方法?

假设考虑到第n个信封，初始时暂时把第n封信放在第n个信封中，然后考虑两种情况的递推：

- 前面$n-1$个信封全部装错;
- 前面$n-1$个信封有一个没有装错其余全部装错。

对于第一种情况，前面$n-1$个信封全部装错:因为前面n一1个已经全部装错了，所以第n封只需要与前面任意一个位置交换即可，总共有$D_{n-1}\times(n-1)$种情况。

对于第二种情况，前面$n-1$个信封有一个没有装错其余全部装错：考虑这种情况的目的在于，若$n-1$个信封中如果有一个没装错，那么把那个没装错的与n交换，即可得到一个全错位排列情况。

其他情况，不可能通过一次操作来把它变成一个长度为n的错排。

于是可得，错位排列数满足递推关系：

$$D_n = (n-1)(D_{n-1} + D_{n-2})$$

这里也给出另一个递推关系：

$$D_n = nD_{n-1} + -1^n$$

4.4.6.3 卡特兰数

n 个数的进栈出栈问题:

给出 $1-n$ 这 n 个数和一个栈，每个数都要入栈一次且出栈一次，进栈的顺序为 $1,2,3,\ldots,n$，可能的出栈序列有多少种。

DP 解法：在某一时刻，我们只需要关心有多少个数尚未入栈，以及栈内还有多少个数。设 $f(i,j)$ 表示有 i 个数尚未进栈，有 j 个数还在栈里，以及有 $n-i-j$ 个数出栈的方案总数。最终状态下，所有数以及出栈，那么 $f(0,0)=1$，我们需要求出的初始状态下的方案总数 $f(n,0)$，每一步的两种决策是把一个数入栈和把栈顶的数出栈，因此：$f(i,j) = f(i-1,j+1) + f(i,j-1)$

递推解法：考虑"1"这个数排在最终出栈序列中的位置，如果最后一个"1"这个数排在第 k 个，那么整个序列的进出栈序列过程为：

"1"入栈。

后面跟 $[2,k]$ 这 $k-1$ 个数按某种顺序进出栈。

"1"出栈，排在第 k 个。

整数 $[k+1,n]$ 这 $n-k$ 个数按某种顺序进出栈。

于是整个问题被划分为 $k-1$ 个数进出栈和 $n-k$ 个数进出栈，那么可以得到递推公式：$f(n)=\sum_{k=1}^{n} f(k-1)f(n-k)$。

定义：

卡特兰数的定义就是形如 1,2,5,14,42,132... 的序列的几种通项公式，包括组合数递推等形式：

1. $f(n)=\frac{C_{2n}^{n}}{n+1}=C_{2n}^{n}-C_{2n}^{n-1},n\geq 1$（证明从组合数计算公式下手）
2. $f(n)=\sum_{i=0}^{n-1} f(i)*f(n-i-1)=\sum_{i=1}^{n} f(i-1)*f(n-i)$
3. $f(n+1)=\frac{f(n)*(4n-6)}{n},f(2)=1,n\geq 2$

递推法：卡特兰数第 i 项为 $f[i]$。

```
ll f[1005];

void katelan(int n) {
    f[0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++) {
            f[i] += (f[j] * f[i - 1 - j]) % Mod;
            f[i] %= Mod;
        }
}
```

组合数法

```
ll C[1005][1005];

ll katelan(int n) {
    C[0][0] = 1;
    for (int i = 1; i < 1005; i++) {
        C[i][0] = C[i][i] = 1;
        for (int j = 1; j <= (i >> 1); j++) {
            C[i][j] = C[i][i - j] = (C[i - 1][j - 1] + C[i - 1][j]) % Mod;
        }
    }
    return (C[2 * n][n] - C[2 * n][n - 1] + Mod) % Mod;    // 不要忘记减法的取模
}
```

线性求法：卡特兰数的第 i 项为 $f[i+2]$

```
ll f[maxn], inv[maxn];

```

```
void katelan(int n, int p) {
    f[2] = inv[1] = 1LL;
     for (int i = 2; i < n + 2; i++)
     inv[i] = (p - p / i) * inv[p % i] % p;
    for (int i = 2; i < n + 2; i++)
        f[i + 1] = (4 * i - 6) * f[i] % p * inv[i] % p;
}
```

卡特兰数其他应用形式：

1. n 个节点的二叉树形态个数；
2. n 对括号的匹配个数问题；
3. 对凸 $n+2$ 边形进行三角剖分的方案个数；
4. $n \times n$ 的网格中从 $(1,1)$ 走到 (n,n) 且不越过第一象限平分线的移动方案数；
5. 在圆上选择 $2n$ 个点，将这些点成对连起来使得所得到的 n 条线段不相交的方法数；

4.4.6.4 斯特林数

1. 第二类斯特林数

第二类斯特林数（斯特林子集数）$\begin{Bmatrix} n \\ k \end{Bmatrix}$，也可记做$S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。

递推式：

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} + k \begin{Bmatrix} n-1 \\ k \end{Bmatrix}$$

边界是 $\begin{Bmatrix} n \\ k \end{Bmatrix} = [n=0]$。

考虑用组合意义来证明。

我们插入一个新元素时，有两种方案：

- 将新元素单独放入一个子集，有$\begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$种方案。
- 将新元素放入一个现有的非空子集，有$k\begin{Bmatrix} n-1 \\ k \end{Bmatrix}$种方案。

根据加法原理，将两式相加即可得到递推式。

通项公式：

$$\begin{Bmatrix} n \\ m \end{Bmatrix} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!\,(m-i)!}$$

使用容斥原理证明该公式。设将 n 个两两不同的元素，划分到 i 个两两不同的集合(允许空集)的方案数为 G_i，将 n 个两两不同的元素，划分到 i 个两两不同的非空集合(不允许空集)的方案数为 F_i。显然

$$G_i = \sum_{j=0}^{i} \binom{i}{j} F_j$$

根据二项式反演

$$F_i = \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} G_j$$

$$= \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} j^n$$

$$= \sum_{j=0}^{i} \frac{i!\,(-1)^{i-j} j^n}{j!\,(i-j)!}$$

考虑 F 与 $\{{}^{n}_{i}\}$ 的关系。第二类斯特林数要求集合之间互不区分，因此 F_i 正好就是 $\{{}^{n}_{i}\}$ 的 $i!$ 倍。于是：

$$\{{}^{n}_{m}\} = \frac{F_m}{m!} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!\,(m-i)!}$$

1) 同一行第二类斯特林数的计算

“同一行”的第二类斯特林数指的是，有着不同的 i，相同的 n 的一系列$\{{}^{n}_{i}\}$。求出同一行的所有第二类斯特林数，就是对 $i = 0 \ldots n$ 求出了将 n 个不同元素划分为 i 个非空集的方案数。

根据上面给出的通项公式，卷积计算即可。该做法的时间复杂度为 $O(n \log n)$。

2) 同一列第二类斯特林数的计算

“同一列”的第二类斯特林数指的是，有着不同的 i，相同的 k 的一系列$\{{}^{i}_{k}\}$。求出同一列的所有第二类斯特林数，就是对 $i = 0..n$ 求出了将 i 个不同元素划分为 k 个非空集的方案数。

利用指数型生成函数计算。

一个盒子装 i 个物品且盒子非空的方案数是 $[i > 0]$。我们可以写出它的指数型生成函数为 $F(x) = \sum_{i=1}^{+\infty} \frac{x^i}{i!} = e^x - 1$。经过之前的学习，我们明白 $F^k(x)$ 就是i个有标号物品放到k个有标号盒子台里的指数型生成函数，那么除掉$k!$就是i个有标号物品放到k个无标号盒子里的指数型生成函数。

$\{{}^{i}_{k}\} = \frac{\left[\frac{x^i}{i!}\right] F^k(x)}{k!}$，$O(n \log n)$ 计算多项式幂即可。

另外，$exp\ F(x)$ $\sum_{i=1}^{+\infty} \frac{F^i(x)}{i!}$ 就是i个有标号物品放到任意多个无标号盒子里的指数型生成函数(EXP 通过每项除以一个 $i!$ 去掉了盒子的标号)。这其实就是贝尔数的生成函数。

2. 第一类斯特林数

第一类斯特林数(斯特林轮换数)$\begin{bmatrix}n\\k\end{bmatrix}$，也可记做$s(n,k)$，表示将$n$个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D]=[B,C,D,A]=[C,D,A,B]=[D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D]\neq[D,C,B,A]$。

递推式：

$$\begin{bmatrix}n\\k\end{bmatrix}=\begin{bmatrix}n-1\\k-1\end{bmatrix}+(n-1)\begin{bmatrix}n-1\\k\end{bmatrix}$$

边界是 $\begin{bmatrix}n\\0\end{bmatrix}=[n=0]$。

该递推式的证明可以考虑其组合意义。

我们插入一个新元索时，有两种方案:

- 将该新元素置于一个单独的轮换中，共有 $\begin{bmatrix}n-1\\k-1\end{bmatrix}$ 种方案；
- 将该元素插入到任何一个现有的轮换中，共有$(n-1)$种方案。$(n-1)\begin{bmatrix}n-1\\k\end{bmatrix}$。

根据加法原理，将两式相加即可得到递推式。

第一类斯特林数没有通项公式。

1) 同一行第一类斯特林数的计算

类似第二类斯特林数，我们构造同行第一类斯特林数的生成函数，即 $F_n(x)=\sum_{i=0}^{n}\begin{bmatrix}n\\i\end{bmatrix}x^i$。

根据递推公式，不难写出：

$$F_n(x)=(n-1)F_{n-1}(x)+xF_{n-1}(x)$$

于是

$$F_n(x)=\prod_{i=0}^{n-1}(x+i)=\frac{(x+n-1)!}{(x-1)!}$$

这其实是 x 的 n 次上升阶乘幂，记做 $x^{\bar{n}}$。这个东西自然是可以暴力分治乘 $O(n\log^2 n)$ 求出的，但用上升幂相关做法可以$O(n\log n)$求出。

2) 同一列第一类斯特林数的计算

仿照第二类斯特林数的计算，我们可以用指数型生成函数解决该问题。注意，由于递推公式和行有关，我们不能利用递推公式计算同列的第一类斯特林数。

显然，单个轮换的指数型生成函数为

$$F(x)=\sum_{i=1}^{n}\frac{(i-1)!\,x^i}{i!}=\sum_{i=1}^{n}\frac{x^i}{i}$$

它的k次幂就是 $\begin{bmatrix}i\\k\end{bmatrix}$ 的指数型生成函数，$O(n\log n)$计算即可。

3. 斯特林数的应用

1) 上升幂与普通幂的相互转化

我们记上升阶乘幂 $x^{\bar{n}} = \prod_{k=0}^{n-1}(x + k)$。

则可以利用下面的恒等式将上升幂转化为普通幂：

$$x^{\bar{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^{\bar{k}}$$

2) 下降幂与普通幂的相互转化

我们记下降阶乘幂 $x^{\underline{n}} = \frac{x!}{(x-n)!} = \prod_{k=0}^{n-1}(x - k)$

则可以利用下面的恒等式将普通幂转化为下降幂：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式：

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

4.4.6.5 贝尔数

贝尔数以埃里克-坦普尔.贝尔命名，是组合数学中的一组整数数列，开首是(OEIS A000110)：

$$B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, B_6 = 203, \ldots$$

B_n是基数为n的集合的划分方法的数目。集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是S。例如$B_3 = 5$因为 3 个元素的集合 a, b, c 有 5 种不同的划分方法：

$$\{\{a\}, \{b\}, \{c\}\}$$
$$\{\{a\}, \{b, c\}\}$$
$$\{\{b\}, \{a, c\}\}$$
$$\{\{c\}, \{a, b\}\}$$
$$\{\{a, b, c\}\}$$

B_0是 1 因为空集正好有 1 种划分方法。

贝尔数适合递推公式：

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

证明：

B_{n+1} 是含有$n+1$个元素集合的划分个数，设 D_n 的集合为 $\{b_1, b_2, b_3, \ldots, b_n\}$，$D_{n+1}$的集合为 $\{b_1, b_2, b_3, \ldots, b_n, b_{n+1}\}$，那么可以认为 D_{n+1}是有D_n增添了一个b_{n+1}而产生的，考虑

元素b_n。

- 假如它被单独分到一类，那么还剩下n个元素，这种情况下划分数为$\binom{n}{n}B_n$；
- 假如它和某 1 个元素分到一类，那么还剩下$n-1$个元素，这种情况下划分数为$\binom{n}{n-1}B_{n-1}$；
- 假如它和某 2 个元素分到一类，那么还剩下$n-2$个元素，这种情况下划分数为$\binom{n}{n-2}B_{n-2}$以此类推就得到了上面的公式。

每个贝尔数都是相应的第二类斯特林数的和。因为第二类斯特林数是把基数为n的集合划分为正好 k 个非空集的方法数目。

$$B_n = \sum_{k=0}^{n} S(n,k)$$

贝尔三角形：用以下方法构造一个三角矩阵(形式类似杨辉三角形):

第一行第一项为$1(a_{1,1}=1)$；

对于$n>1$，第n行第一项等于第$n-1$行的第 $n-1$项$(a_{n,1}=a_{n-1,n-1})$；

对于$m,n>1$，第n行的第m项等于它左边和左上角两个数之和$(a_{n,m}=a_{n,m-1}+a_{n-1,m-1})$

部分结果如下：

```
1
1       2
2       3       5
5       7       10      15
15      20      27      37      52
52      67      87      114     151     203
203     255     322     409     523     674     877
```

每行的首项是贝尔数。可以利用这个三角形来递推求出 Bell 数。

```
const int maxn = 2000 + 5;
int bell[maxn][maxn];

void f(int n) {
    bell[1][1] = 1;
    for (int i = 2; i <= n; i++) {
        bell[i][1] = bell[i - 1][i - 1];
        for (int j = 2; j <= i; j++) bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
    }
}
```

4.4.6.6 伯努利数

1. 伯努利数

伯努利数 B,是一个与数论有密切关联的有理数序列。前几项被发现的伯努利数分别为：

$B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_3 = 0, B_4 = -\frac{1}{30}$

2. 等幂求和

伯努利数是由雅各布.伯努利的名字命名的，他在研究m次幂和的公式时发现了奇妙的关系。我们记

$$S_m(n) = \sum_{k=0}^{n-1} k^m = 0^m + 1^m + \cdots + (n-1)^m$$

伯努利观察了如下一列公式，勾画出一种模式：

$S_0(n) = n$

$S_1(n) = \frac{1}{2}n^2 - \frac{1}{2}n$

$S_2(n) = \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$

$S_3(n) = \frac{1}{4}n^4 - \frac{1}{2}n^3 + \frac{1}{4}n^2$

$S_3(n) = \frac{1}{5}n^5 - \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{30}n$

可以发现，在$S_m(n)$中n^{m+1}的系数总是m，n^m的系数总是$-\frac{1}{2}$，n^{m-1}的系数总是$\frac{m}{12}$，n^{m-3}的系数是$\frac{-m(m-1)(m-2)}{720}$，n^{m-4}的系数总是 0 等。

而 n^{m-k} 的系数总是某个常数乘以$m^{\underline{k}}$，$m^{\underline{k}}$表示下降阶乘幂，即$\frac{m!}{(m-k)!}$。

3. 递推公式

$S_m(n) = \frac{1}{m+1}\left(B_0 n^{m+1} + \binom{m+1}{1} B_1 n^m + \cdots + \binom{m+1}{n} B_m n^m\right) = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k} B_k n^{m+1-k}$

伯努利数由隐含的递推关系定义：

$$\begin{cases} \sum_{j=0}^{m}\binom{m+1}{j} B_j = 0, (m > 0) \\ B_0 = 1 \end{cases}$$

```
1. typedef long long ll;
2. const int maxn = 10000;
3. const int mod = 1e9 + 7;
4. ll B[maxn];          // 伯努利数
5. ll C[maxn][maxn];    // 组合数
```

```
ll inv[maxn];       // 逆元（计算伯努利数）

void init() {
    // 预处理组合数
    for (int i = 0; i < maxn; i++) {
        C[i][0] = C[i][i] = 1;
        for (int k = 1; k < i; k++) {
            C[i][k] = (C[i - 1][k] % mod + C[i - 1][k - 1] % mod) % mod;
        }
    }
    // 预处理逆元
    inv[1] = 1;
    for (int i = 2; i < maxn; i++) {
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
    }
    // 预处理伯努利数
    B[0] = 1;
    for (int i = 1; i < maxn; i++) {
        ll ans = 0;
        if (i == maxn - 1) break;
        for (int k = 0; k < i; k++) {
            ans += C[i + 1][k] * B[k];
            ans %= mod;
        }
        ans = (ans * (-inv[i + 1]) % mod + mod) % mod;
        B[i] = ans;
    }
}
```