基本数学知识

快速幂

快速幂是针对快速求解 A^b 结果的算法，对于 b 可以分解为 2 进制，例如对 $3^{11}=3^{2^3+2^1+2^0}$，由于 b 可以被分解后最多只会包含 $log_2 b$ 个 1，因此时间复杂度为 $O(log_2 b)$，而并非原本的 $O(b)$。

代码实现

```
int quickPow(int a, int b) {
    int res = 1;
    while (b) {
        if (b & 1) res *= a;
        a *= a;
        b >>= 1;
    }
    return res;
}
```

例题

【模板】快速幂

质数

质数（英文名：Primenumber）又称素数，是指在大于1的自然数中，除了1和它本身以外不再有其他因数的自然数。

判断方法

我们判断一个数 n 是不是素数时，可以采用试除法，即从 $i=2$ 开始，一直让 n 去 $\%i$，直到 i 到了 $n-1$.

```
#include <stdio.h>
signed main() {
    int n;
    for (int i = 2; i <= n - 1; i++) {
        if (n % i == 0) {
            printf("%d 不是素数",n);
            return 0;
        }
    }
    printf("%d 是素数", n);
}
```

区间素数预处理

问题引入

求 $1\sim n$ 区间内的所有质数。

问题分析

朴素筛法

很自然的一种想法就是对 1 到 n 之间的所有数进行一次素数检验，不过显然这种方式很低效，大致复杂度为 $O(n\sqrt{n})$

Eratosthenes筛法

埃筛用到的原理十分简单：一个质数 p 的任意 $x(x \geqslant 1)$ 倍都是合数，对于任意一个合数 c 都存在一个质数 p 使得 $c = xp(x \in \mathbb{N}, 1 < x < n)$。这句话的前半句保证了埃筛通过质数的若干倍筛去的数必定是合数，而后半句表示了埃筛必定可以在筛去所有的合数。采用从 1 到 n 的遍历顺序，就可以在遍历到某个合数之前筛去它，从而得到素数表。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1e8;
int prime[maxn], cnt;
bool vist[maxn + 1];
void Eratothenes(int n){
    for (int i = 2; i <= n; i ++){
        if (!vist[i]){
            prime[++ cnt] = i;
            for (long long j = 1ll * i * i; j <= n; j += i){
                vist[j] = true;
            }
        }
    }
    printf("%d\n", cnt);
}
int main(){
    Eratothenes(maxn);
    return 0;
}
```

Eratosthenes 筛法时间效率优化

1. 减小筛法上界

   需要求出 1 到 n 的所有质数，并不需要取质数进行筛选，只需要取小于 $\sqrt{n}$ 的质数筛选即可。

2. 只筛奇数

   由于非 2 的偶数都是合数，所以不用考虑奇数，只关心奇数即可。

Eratosthenes 筛法空间效率优化

1. 由质数个数函数的渐近 $\pi(x) \approx \frac{n}{\ln n}$ 可知，质数数组的长度可以比 n 小，大致是 $\frac{n}{\ln n}$ 量级，使用不定长数组可以稍稍节约空间。
2. 由于只需要取小于 $\sqrt{n}$ 的质数进行筛选，因此，只需要保存 1 11 到 $\sqrt{n}$ 的vist数组，然后对后面的数分成 $\lceil \frac{n}{s} \rceil$ 个大小为 s 块，用前几个质数筛每一个块里的数，因此每次只需要 $\lceil \frac{n}{s} \rceil$ 大小的数组，总空间复杂度为 $O(\sqrt{n} + s)$

Euler 筛法

即使对埃筛进行上述优化，依然会有合数会被多个质数筛掉。下面介绍Euler筛法，将筛法的复杂度优化至线性。Euler筛法，又称欧拉筛、线性筛法、线筛。是一种可以在线性时间复杂度内求出 $1 \sim n$ 之间所有质数的筛法。思路是：让每个合数只被其最小的质因子筛去，从而保证每个合数只被筛去一次，做到线性复杂度。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e8;
int prime[maxn], cnt;
bool vist[maxn + 1];
void Euler(int n){
    for (int i = 2; i <= n; i ++){
        if (!vist[i]) prime[++ cnt] = i;
        for (int j = 1; j <= cnt && i * prime[j] <= n; j ++){
            vist[i * prime[j]] = true;
```

```
            if (i % prime[j] == 0) break;
        }
    }
    printf("%d\n", cnt);
}
int main(){
    Euler(maxn);
    return 0;
}
```

约数

约数，又称因数。整数 a 除以整数 $b(b \neq 0)$ 除得的商正好是整数而没有余数，就说 a 能被 b 整除，或 b 能整除 a。a 称为 b 的倍数，b 称为 a 的约数。

GCD

gcd 的意思是最大公约数，通常用扩展欧几里得算法求，原理：$gcd(a,b) = gcd(b,a\%b)$。

证明：

$$令d = gcd(a,b) \rightarrow a = m \times d，b = n \times d$$
$$则m \times d = t \times n \times d + a\%b \rightarrow a\%b = d \times (m - t \times n)$$
$$gcd(b,a\%b) = gcd(n \times d,(m - t \times n) \times d)$$
$$令gcd(n,m - t \times n) = e \rightarrow n = x \times e，m - t \times n = y \times e$$
$$则m - x \times e \times n = y \times e \rightarrow m = e \times (x \times n + y)$$
$$由gcd(n,m) = 1知gcd(e \times (x \times n + y),e \times x) = 1$$
$$故e = 1$$
$$故gcd(n \times d,(m - t \times n) \times d) = d即gcd(b,a\%b) = gcd(a,b)$$

```
#include "bits/stdc++.h"
using namespace std;
int gcd(int a,int b)
{
    if(b == 0)  return a;
    else   return gcd(b,a % b);
}
int main()
{
    int a,b;
    cin >> a >> b;
    cout << gcd(a,b) << endl;
    return 0;
}
```

ExGCD

裴蜀定理：$ax + by = gcd(a,b)$，a,b 为非负整数且 $a + b > 0$

exgcd：$ax + by = c$，其中$c\%gcd(a,b) = 0$

所以 exgcd 是裴蜀定理的扩展版。

$$\begin{aligned}
&ax + by = gcd(a,b)\\
&\rightarrow bx' + (a\%b)y' = gcd(b,a\%b)\\
&\rightarrow bx' + (a - \lfloor \tfrac{a}{b} \rfloor \times b)y' = gcd(b,a\%b)\\
&\rightarrow ay' + b(x' - \lfloor \tfrac{a}{b} \rfloor y') = gcd(b,a\%b)\\
&令x = y'，\ y = x' - \lfloor \tfrac{a}{b} \rfloor y'，从gcd(b,a\%b)转移到了gcd(a,b)\\
&\rightarrow ax + by = gcd(a,b)
\end{aligned}$$

考虑已经通过exgcd得到一个解为：$x = x_0, y = y_0$
代入式后为：$ax_0 + by_0 = c$
那么当有$x_1 = x_0 - t$，$y_1 = \frac{c-ax_1}{b} = \frac{c-a(x_0-t)}{b} = \frac{c-ax_0+at}{b} = y_0 + \frac{at}{b}$
这里t为整数，故x_1也为整数，现在需要y_1也为整数。
这里$d = gcd(a,b), a' = \frac{a}{d}$，$b' = \frac{b}{d}$，$\frac{at}{b} = \frac{a't}{b'}$，
这里$gcd(a',b') = 1$，如果想要使得$\frac{a't}{b'}$为整数，那么t必须是b'的倍数，则$t = kb'$。
则有：

$$\begin{cases} x_1 = x_0 - k\frac{b}{gcd(a,b)} \\ y_1 = y_0 + k\frac{a}{gcd(a,b)} \end{cases}$$

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
int ex_gcd(int a,int b,int &x,int &y)
{
    if(!b) {x=1,y=0;return a;}
    int gcd=ex_gcd(b,a%b,x,y),tmp=x;
    x=y,y=tmp-a/b*y; return gcd;
}
int a,b,c,gcd,x1,y1;
int main()
{
    scanf("%d%d%d",&a,&b,&c),gcd=ex_gcd(a,b,x1,y1);
    if(c%gcd) {printf("-1\n");return 0;}
    x1*=c/gcd,y1*=c/gcd;
    printf("x=%d+k*%d\n",x1,b/gcd);
    printf("y=%d-k*%d\n",y1,a/gcd);
}
```

【模板】二元一次不定方程 (exgcd)

Euler 筛法的用途

Euler 函数

在数论中，欧拉函数，指小于等于 n 且与 n 互质的数的个数，用 $\varphi(n)$ 表示。

性质

1. $\varphi(1) = 1$
2. 当 n 是质数时，$\varphi(n) = n - 1$。
3. 当 n 的唯一分解形式为 $n = p^k$ 时，$\varphi(n) = p^k - p^{k-1}$
4. 欧拉函数是积性函数。即，若 $gcd(a,b) = 1$，则 $\varphi(a \times b) = \varphi(a) \times \varphi(b)$
5. 当 n 的唯一分解形式为 $n = p_1^{k_1} p_2^{k_2} \dots p_s^{k_s}$ 时，$\varphi(n) = n\Pi_{i=1}^{s}(1 - \frac{1}{p_i})$

通过 Euler 筛法

通过 $\varphi(i)$ 和 $\varphi(prime[j])$ 计算 $\varphi(i * prime[j])$ 分成两种情况。

1. case 1: $gcd(i, prime[j]) = 1$，由积性函数的性质有：

$$\varphi(i * prime[j]) = \varphi(i) * \varphi(prime[j]) = \varphi(i) * (prime[j] - 1)$$

2. case 2: $gcd(i, prime[j]) = prime[j]$，根据 $i * prime[j]$ 和 i 的唯一分解、欧拉函数的性质有：

$$\varphi(i * prime[j]) = (i * prime[j]) * \Pi_{i=1}^{k}(1 - \frac{1}{p_i}) = prime[j] * (i * \Pi_{i=1}^{k}(1 - \frac{1}{p_i})) = prime[j] * \varphi(i)$$

3.

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 10000;
int phi[maxn];
int prime[maxn], cnt;
bool vist[maxn + 1];
void Euler(int n){
    phi[1] = 1;
    for (int i = 2; i <= n; i ++){
        if (!vist[i]) prime[++ cnt] = i, phi[i] = i - 1;
        for (int j = 1; j <= cnt && i * prime[j] <= n; j ++){
            vist[i * prime[j]] = true;
            if (i % prime[j] != 0) phi[i * prime[j]] = phi[i] * (prime[j] - 1);
            else {
                phi[i * prime[j]] = phi[i] * prime[j];
                break;
            }
        }
    }
}
int main(){
    Euler(maxn);
    for (int i = 2; i <= maxn; i ++) printf("%d: %d\n", i, phi[i]);
    return 0;
}
```

约数个数函数

用 $d(x)$ 表示，其值为 x 的约数个数。根据 x 的唯一分解和乘法原理可得：若 $x = \Pi_{a=1}^{k} p_a^{c_a}$，则 $d(x) = \Pi_{i=a}^{k}(c_a + 1)$。易证约数个数函数为积性函数，尝试通过 Euler 筛法计算，分类讨论。

case 1: $gcd(i, prime[j]) = 1$，由积性函数性质有：$d(i * prime[j]) = d(i) * d(prime[j]) = 2d(i)$

case 2: $gcd(i, prime[j]) = prime[j]$，由 $i * prime[j]$ 和 i 的唯一分解、约数个数函数的性质有（由于 $prime[j]$ 为 i 的最小质因子，不妨设 $i = (prime[j])^{c_1} * \Pi_{a=2}^{k} p_a^{c_a}$ ：
$d(i * prime[j]) = (c_1 + 2) * \Pi_{a=2}^{k}(c_a + 1) = \frac{c_1+2}{c_1+1} * \Pi_{a=1}^{k} c_a = \frac{c_1+2}{c_1+1} d(i)$
因此，要计算 $d(x)$ 的值，只需要记录最小质因子对应的次数 c_1 后用同样的方式计算即可。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e6;
int d[maxn], c[maxn];
int prime[maxn], cnt;
bool vist[maxn + 1];
void Euler(int n){
    d[1] = 1;
    for (int i = 2; i <= n; i ++){
        if (!vist[i]) prime[++ cnt] = i, d[i] = 2, c[i] = 1;
        for (int j = 1; j <= cnt && i * prime[j] <= n; j ++){
            vist[i * prime[j]] = true;
            if (i % prime[j] != 0){
                c[i * prime[j]] = 1;
                d[i * prime[j]] = d[i] * 2;
            }
            else {
                c[i * prime[j]] = c[i] + 1;
                d[i * prime[j]] = d[i] / c[i * prime[j]] * (c[i * prime[j]] + 1);

                break;
            }
```

```
        }
    }
}
int main(){
    Euler(maxn);
    for (int i = 1; i <= 100; i ++) printf("%d: %d\n", i, d[i]);
    return 0;
}
```

约束和函数

约束和函数，记作 $sumd(x)$，其值为 x 的所有约数的和。根据 x 的唯一分解和加乘原理可得：若 $x = \Pi_{a=1}^{k} p_a^{c_a}$，则 $sumd(x) = \Pi_{a=1}^{k}(\Sigma_{b=0}^{c_a} p_a^b)$

case 1: $gcd(i, prime[j]) = 1$，由积性函数性质有：
$sumd(i * prime[j]) = sumd(i) * sumd(prime[j]) = sum(d) * (1 + prime[j])$

case 2: $gcd(i, prime[j]) = prime[j]$，由 $i * prime[j]$ 和 i 的唯一分解、约数个数函数的性质有（由于 $prime[j]$ 为 i 的最小质因子，不妨设 $i = (prime[j])^{c_1} * \Pi_{a=2}^{k} p_a^{c_a}$：

$$sumd(i * prime[j]) = \Sigma_{b=0}^{c_1+1} prime[j]^b * \Pi_{a=2}^{k}(\Sigma_{b=0}^{c_a} p_a^b) = \frac{\Sigma_{b=0}^{c_1+1} prime[j]^b}{\Sigma_{b=0}^{c_1} prime[j]^b} * \Pi_{a=1}^{k}(\Sigma_{b=0}^{c_a} p_a^b) = \frac{\Sigma_{b=0}^{c_1+1} prime[j]^b}{\Sigma_{b=0}^{c_1} prime[j]^b} * sumd(i)$$

又观察到：$\Sigma_{b=0}^{c_1+1} prime[j]^b = (\Sigma_{b=0}^{c_1} prime[j]^b) * prime[j] + 1$

因此，要计算 $sumd(x)$ 的值，只需要记录其最小质因子的累计幂级数的和 $\Sigma_{b=0}^{c_1} prime[j]^b$

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 100;
long long sumd[maxn], pre[maxn];
int prime[maxn], cnt;
bool vist[maxn + 1];

void Euler(int n){
    sumd[1] = 1;
    for (int i = 2; i <= n; i ++){
        if (!vist[i]) prime[++ cnt] = i, sumd[i] = 1 + i, pre[i] = 1 + i;
        for (int j = 1; j <= cnt && i * prime[j] <= n; j ++){
            vist[i * prime[j]] = true;
            if (i % prime[j] != 0){
                pre[i * prime[j]] = (1 + prime[j]);
                sumd[i * prime[j]] = sumd[i] * (1 + prime[j]);
            }
            else {
                pre[i * prime[j]] = pre[i] * prime[j] + 1;
                sumd[i * prime[j]] = sumd[i] / pre[i] * pre[i * prime[j]];
                break;
            }
        }
    }
}

int main(){
    Euler(maxn);
    for (int i = 1; i <= 100; i ++) printf("%d: %d\n", i, sumd[i]);
    return 0;
}
```

同余

数论中的重要概念。给定一个正整数 m，如果两个整数 a 和 b 满足 $a-b$ 能够被 m 整除，即 $(a-b)/m$ 得到一个整数，那么就称整数 a 与 b 对模 m 同余，记作 $a \equiv b(modm)$。对模 m 同余是整数的一个等价关系。

中国剩余定理

Th.

给出一元线性同余线性方程组

$x \equiv a1 \bmod m1$

$x \equiv a2 \bmod m2$

...

$x \equiv an \bmod mn$

定理指出假设素数$m1, m2...mn$两两互素,对任意的整数$a1, a2...an$

上述方程有解，并且可以通过如下步骤构造通解：

1>计算$M=\prod_{i=1}^{n} mi, Mi=\frac{M}{mi}$(即,Mi是除$mi$之外的其他整数的乘积)

2>计算ti为Mi模mi的逆,对于$i=1,2,..,n$,计算满足$tiMi \equiv 1(\bmod mi)$的ti

3>上述方程的通解为:

$x=a1t1M1+a2t2M2+...+antnMn+kM, k \in Z$

在模M的意义下,方程组只有一个解$x=a1t1M1+a2t2M2+...+antnMn$

扩展中国剩余定理

扩展中国剩余定理是解决向下面列出的一元线性同余方程组的一种数论知识，可以求出下面方程组中最下的正整数 x。但是扩展中国剩余定理和中国剩余定理有什么区别呢？中国剩余定理对于mod是有限制的，他对于 mod 要求为两两互质，然而扩展中国剩余定理对于 mod 没有要求。

$$\begin{cases} x \equiv x_1 \ （mod_1） \\ x \equiv x_2 \ （mod_2） \\ \quad \vdots \qquad \vdots \\ x \equiv x_3 \ （mod_3） \end{cases}$$

首先我们看上方列出的方程组，我们选择最上方的两个方程，看能不能合成。

我们将上面的通与方程进行变形可以得到下面的一个方程组。

$$\begin{cases} x = x_1 + k_1 \times mod_1 \\ x = x_2 + k_2 \times mod_2 \end{cases}$$

我们可以根据上面的方程组得到一个式子 $x_1 + k_1 \times mod_1 = x_2 + k_2 \times mod_2$，我们将这个式子进行移项可以得到 $k_1 \times mod_1 + (-k_2) \times mod_2 = x_2 - x_1$，这个式子很像 $ax + by = c$ 的形式，所以很容易想到用扩展 gcd，我们用扩展 gcd 可以求出来 k_1 的通项，但是前提是 $gcd(mod_1，mod_2) | (x_2 - x_1)$，如果上面的前提不满足，则这个同余方程组无法满足。

我们设 $K_1，K_2$ 为 $K_1 \times mod_1 + (-K_2) \times mod_2 = gcd(mod_1，mod_2)$ 的两个特解。我们将式子两边都乘以 $\frac{(x_2-x_1)}{gcd(mod_1，mod_2)}$，就能知道上述式子 $k_1 = K_1 \times \frac{(x_2-x_1)}{gcd(mod_1，mod_2)}$，$k_2 = K_2 \times \frac{(x_2-x_1)}{gcd(mod_1，mod_2)}$。我们将 $k_1，k_2$ 回带到原来的式子，能得到一个方程组（下方）。

$$\begin{cases} x = x_1 + K_1 \times \frac{(x_2-x_1)}{gcd(mod_1，mod_2)} \times mod_1 \\ x = x_2 + K_2 \times \frac{(x_2-x_1)}{gcd(mod_1，mod_2)} \times mod_2 \end{cases}$$

显然满足这个方程组的 $K_1，K_2$ 不只一组，并且每一组 $K_1，K_2$ 都对应一个 x，所以我们能知道每两个解 Δ 就是 $\frac{(x_2-x_1)}{gcd(mod_1，mod_2)} \times mod_1$ 和 $\frac{(x_2-x_1)}{gcd(mod_1，mod_2)} \times mod_2$ 的倍数，所以 Δ 一定是 $\frac{mod_1 \times mod_2}{gcd(mod_1，mod_2)}$ 的倍数，即 Δ 是 $lcm(mod_1，mod_2)$ 的倍数。我们就能知道 $x = k_1 \times mod_1 + x_1 + t \times lcm(mod_1，mod_2)$。这样我们就将上面两个同余方程合成为 $x \equiv k_1 \times mod_1 + x_1 \ （lcm(mod_1，mod_2)）$。

我们可以用 $O(log)$，运用 $Exgcd$ 合并两个同余方程，一共做 n 次，就能将所有的同余方程合并在一起了。最后我们得到的同余方程就是所有的 x 的通式。

代码（模板），下面以有 n 个同余方程为例。

```
#define ll long long
```

```
ll Exgcd(ll a,ll b,ll &x,ll &y)
{
    if(!b) {x=1,y=0;return a;}
    ll gcd=Exgcd(b,a%b,x,y),tmp=x;
    x=y,y=tmp-a/b*y;
    return gcd;
}
ll Ex_crt()
{
    ll lcm=mod[1],last_x=x[1];
    for(int i=2;i<=n;i++)
    {
        ll lcm_a=((x[i]-last_x)%mod[i]+mod[i])%mod[i],x,y,k=lcm;
        ll gcd=Exgcd(lcm,mod[i],x,y);
        x=(x*lcm_a/gcd%(mod[i]/gcd)+(mod[i]/gcd))%(mod[i]/gcd);
        lcm=lcm*mod[i]/gcd,last_x=(last_x+k*x)%lcm;
    }
    return (last_x%lcm+lcm)%lcm;
}
```

矩阵

矩阵乘法：

对于两个矩阵A和B：

$$A_{n*m}=\begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \qquad B_{m*p}=\begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix}$$

将他们相乘的结果记为C

$$A_{n*m}*B_{m*p}=C_{n*p}\begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{pmatrix}$$

不难发现，C的行数与A的行数相同，C的列数与B的列数相同，A的列数与B的行数相同。

那怎么求出C中的元素呢?

$$c_{ij}=a_{i1}*b_{1j}+a_{i2}*b_{2j}+\cdots+a_{im}*b_{mj}=\sum_{k=1}^{m}a_{ik}*b_{kj}$$

举个简单的例子：

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix}\times\begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix}=\begin{bmatrix} (1\times3+0\times2+2\times1) & (1\times1+0\times1+2\times0) \\ (-1\times3+3\times2+1\times1) & (-1\times1+3\times1+1\times0) \end{bmatrix}=\begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

咳咳，矩阵乘法满足的运算性质有：矩阵间的乘法结合律、数乘和矩阵乘结合律、矩阵间的乘法分配律，**但是不满足乘法交换律！！**

若矩阵A是n阶方阵，那么A的i次幂乘A的j次幂就是A的i+j次幂，A的i次幂的j次幂等于A的ij次幂。

矩阵转置：把矩阵 A 的行换成同序数的列得到的新矩阵，叫做A的转置矩阵，记作AT。

转置矩阵的运算性质：

1、（AT）T=A

2、（A+B）T=AT+BT

3、$(\lambda A)^T = \lambda A^T$

4、（AB）T=BTAT

矩阵加法：

对于两个形状相同的矩阵A，B：

$$A_{n*\mathrm{m}} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \qquad B_{n*\mathrm{m}} = \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$$

设两集合之和组成的集合为C：

$$A_{n*\mathrm{m}} + B_{n*\mathrm{m}} = C_{n*n} = \begin{pmatrix} c_{11} & \cdots & c_{1\mathrm{m}} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{n\mathrm{m}} \end{pmatrix}$$

C中每个元素都是A、B集合相应位置的元素之和，即：

$$C_{ij} = A_{ij} + B_{ij}$$

嘿嘿，如果这个矩阵的行数和列数均等于n，那我们就叫他n阶方阵，记作A_n

只有一行的就是行矩阵，只有一列的就是列矩阵，元素全是0的叫做零矩阵

如果两个矩阵行数列数均相同，他们就成为同型矩阵

如果两个同型矩阵的对应元素相等，那这两个矩阵就相等

如果**A是一个n阶方阵，如果满足：A=AT，即aij=aji，那么A就是一个对称阵**

如果**A=-AT，那么A就是反对称阵。**

很显然，对称阵的特点就是沿着从左上角到右下角的对角线，其两侧对称。

而对于反对称矩阵，就要满足对角线上的元素都为0，且关于对称轴位置对称的两个元素，其互为相反数。

```
#include<bits/stdc++.h>
using namespace std;
int a[110][110],b[110][110],c[110][110],m,n,k;
double s;
int main()
{
    cin>>n>>m>>k;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            cin>>a[i][j];
    for(int i=1;i<=m;i++)
        for(int j=1;j<=k;j++)
```

```
            cin>>b[i][j];
    for(int i=1;i<=n;i++)
        for(int j=1;j<=k;j++)
            for(int l=1;l<=m;l++)
                c[i][j]+=a[i][l]*b[l][j];//将新矩阵的每一个点都看作计数器来计算
    for(int i=1;i<=n;i++){
        for(int j=1;j<=k;j++)
            cout<<c[i][j]<<" ";
        cout<<endl;//注意换行
    }
    return 0;
}
```

行列式

行列式：有点难解释，下面会有所体现的

行列式求值：

如图：对于行列式A，他的值为：

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} = \sum_{p_1 p_2 \cdots p_n} (-1)^{t(p_1 p_2 \cdots p_n)} a_{1p_1} a_{2p_2} \cdots a_{np_n}$$

其中 t(p1,p2,...,pn)代表数列p1,p2,...,pn数列中逆序对的个数

如果t为偶数，那就加上这一项，如果t是奇数，那就减去这一项。

根据图中解释的那样，每行选取任意选取一个a[i,j]，且保证每行选取的元素不在同一列（选取的元素的j各不相同）。那么将枚举的元素乘起来，判断按照行数排列的列数数列中一共有几个逆序对来判断正负，最后将所有得到的乘积相加即可得到行列式的值。

偶对了，再提一句，**行列式的行数和列数要一样才行**

$$D_1 = \begin{vmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{vmatrix} \quad D_2 = \begin{vmatrix} 0 & 0 & 0 & a_{14} \\ 0 & 0 & a_{23} & 0 \\ 0 & a_{32} & 0 & 0 \\ a_{41} & 0 & 0 & 0 \end{vmatrix} \quad D_3 = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{vmatrix} \quad D_4 = \begin{vmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{32} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix}$$

不难发现，以上这四个行列式都等于a11+a22+a33+a44，这个结论对后面的行列式转移有很大帮助

行列式的性质：

1，**若我们将一个行列式D旋转90度得到新的行列式称作转置行列式（DT），那么D=DT**

2、**如果将行列式的两行（或两列）交换位置，那么新得到的行列式D1与原行列式D的关系为：D=-D1（交换后t的奇偶性变了）**

那么得到进一步推论：推论 **如果行列式有两行（列）完全相同，则此行列式为零。**

3、**如果行列式某一行（或列）的所有元素都乘了k，那么整个行列式的值就变成了原来的k倍。**（很好证明，每次都会选取这一行的某个元素相乘，总结果就会扩大k倍，举一个例子加以证明，如下：）

$$D_1=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ ka_{21} & ka_{22} & ka_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$\begin{aligned} &= a_{11}(ka_{22})a_{33}+a_{12}(ka_{23})a_{31}+a_{13}(ka_{21})a_{32} \\ &\quad -a_{13}(ka_{22})a_{31}-a_{12}(ka_{21})a_{33}-a_{11}(ka_{23})a_{32} \\ &= k\begin{pmatrix} a_{11}a_{22}a_{33}+a_{12}a_{23}a_{31}+a_{13}a_{21}a_{32} \\ -a_{13}a_{22}a_{31}-a_{12}a_{21}a_{33}-a_{11}a_{23}a_{32} \end{pmatrix} \\ &= kD \end{aligned}$$

通过这个结论我们也可以得出：

行列式的某一行（列）中所有元素的公因子可以提到行列式符号的外面

4、 **行列式中如果有两行（列）元素成比例，则此行列式为零**

证明起来也很简单，假设这两行的对应元素比值为k，由于性质3，我们可以将其中的k提出来，这样就有两行（列）是相同的，那么由性质2可以推出整个行列式的值为0

5、**若行列式的某一列（行）的元素都是两数之和,则可写作两个行列式之和**，例如：

$$D=\begin{vmatrix} a_{11} & a_{12}+b_{12} & a_{13} \\ a_{21} & a_{22}+b_{22} & a_{23} \\ a_{31} & a_{32}+b_{32} & a_{33} \end{vmatrix}, \quad D=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}+\begin{vmatrix} a_{11} & b_{12} & a_{13} \\ a_{21} & b_{22} & a_{23} \\ a_{31} & b_{32} & a_{33} \end{vmatrix}$$

证明也很简单：

$$D=\begin{vmatrix} a_{11} & a_{12}+b_{12} & a_{13} \\ a_{21} & a_{22}+b_{22} & a_{23} \\ a_{31} & a_{32}+b_{32} & a_{33} \end{vmatrix}$$

$$=\sum_{p_1p_2p_3}(-1)^{t(p_1p_2p_3)}a_{1p_1}(a_{2p_2}+b_{2p_2})a_{3p_3}$$

$$=\sum_{p_1p_2p_3}(-1)^{t(p_1p_2p_3)}a_{1p_1}a_{2p_2}a_{3p_3}+\sum_{p_1p_2p_3}(-1)^{t(p_1p_2p_3)}a_{1p_1}b_{2p_2}a_{3p_3}$$

$$=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}+\begin{vmatrix} a_{11} & b_{12} & a_{13} \\ a_{21} & b_{22} & a_{23} \\ a_{31} & b_{32} & a_{33} \end{vmatrix}$$

6、**把行列式的某一列（行）的各元素乘以同一个倍数然后加到另一列(行)对应的元素上去，行列式不变**

证明：

设：

$$D=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \quad D_1=\begin{vmatrix} a_{11} & a_{12}+ka_{13} & a_{13} \\ a_{21} & a_{22}+ka_{23} & a_{23} \\ a_{31} & a_{32}+ka_{33} & a_{33} \end{vmatrix}$$

由性质5可以将D1展开：

$$D1=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}+\begin{vmatrix} a_{11} & ka_{13} & a_{13} \\ a_{21} & ka_{23} & a_{23} \\ a_{31} & ka_{33} & a_{33} \end{vmatrix}$$

由性质4可知：

$$\begin{vmatrix} a_{11} & ka_{13} & a_{13} \\ a_{21} & ka_{23} & a_{23} \\ a_{31} & ka_{33} & a_{33} \end{vmatrix}=0$$

也就是说，D1就是前面的那一项，即：

$$D_1=\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}=D$$

证毕

通过这些性质，我们可以将任意一个行列阵转换成上三角形式：

比如：

$$D=\begin{vmatrix} 1 & -1 & 2 & -3 & 1 \\ -3 & 3 & -7 & 9 & -5 \\ 2 & 0 & 4 & -2 & 1 \\ 3 & -5 & 7 & -14 & 6 \\ 4 & -4 & 10 & -10 & 2 \end{vmatrix}$$

将第一列中的第一个作为基准，将其乘上某个数再加到其他的行中，是其他行中的第一个数都变成0，即：

$$=\begin{vmatrix} 1 & -1 & 2 & -3 & 1 \\ 0 & 0 & -1 & 0 & -2 \\ 0 & 2 & 0 & 4 & -1 \\ 0 & -2 & 1 & -5 & 3 \\ 0 & 0 & 2 & 2 & -2 \end{vmatrix}$$

交换第2行和第3行得：

$$=\begin{vmatrix} 1 & -1 & 2 & -3 & 1 \\ 0 & 2 & 0 & 4 & -1 \\ 0 & 0 & -1 & 0 & -2 \\ 0 & -2 & 1 & -5 & 3 \\ 0 & 0 & 2 & 2 & -2 \end{vmatrix}$$

```
long long calc(int n)
{
```

```
    long long tmp=1;
    for(int i=1;i<n;i++)
    {
        int j; for(j=i;j<n;j++) if(squ[j][i]) break;
        if(j==n) continue;
        if(j!=i) {for(int k=i;k<n;k++) swap(squ[i][k],squ[j][k]);tmp*=-1;}
        for(j=i+1;j<n;j++)
        {
            while(squ[j][i])
            {
                long long t=squ[j][i]/squ[i][i];
                for(int k=i;k<n;k++) squ[j][k]=(squ[j][k]-squ[i][k]*t%mod+mod)%mod;
                if(!squ[j][i]) break;
                for(int k=i;k<n;k++) swap(squ[i][k],squ[j][k]); tmp*=-1;
            }
        }
    }
    for(int i=1;i<n;i++) (tmp*=squ[i][i])%=mod; return tmp;
}
```