**NUMBER THEORY – MATH**

**/\* binary power modular (b ^ e) % mod \*/**

**int** power ( **int** b, **int** e ) {

**int** ret = 1;

**while** ( e ) {

**if** ( e & 1 )

ret = ( 1ll \* ret \* b ) % mod;

e >>= 1;

b = ( 1ll \* b \* b ) % mod;

}

**return** ret;

}

**/\* inverse modular \*/**

**int** inv ( **int** x ) {

**return** power ( x, mod-2 );

}

**/\* calculate, if exist, x such that xa == 1 (mod m) \*/**

**int** invMod (**int** a, **int** m) {

**int** x, y;

**if** (extGCD(a, m, x, y) == 1 )

**return** (x + m) % m;

**return** 0; // unsolvable

}

**/\* convert integer to any base \*/**

**typedef** **vector**<**char**> vc;

**vc** convert(**int** a, **int** b) {

**vc** ans;

**bool** sign;

if((sign = (a < 0)))

a = -a;

**do** {

ans.**push\_back**("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"[a % b]);

} **while**(a /= b);

**if**(sign) ans.**push\_back**('-');

**reverse**(ans.begin(), ans.end());

**return** ans;

}

**/\* convert binary to integer \*/**

**typedef** **vector**<**int**> vi;

**int** binToInt ( **vi** &bin ) {

**int** a = 0;

**int** len = bin.size();

**for** ( **int** i = 0; i < len; i++) {

a |= bin[i];

**if** ( i != len – 1 )

a <<= 1;

}

**return** a;

}

**/\* convert integer to binary \*/**

**vi** intToBin(**int** a)

{

**int** one = 0;

**int** c = 1;

**vi** bin(**sizeof**(a) \* 8);

int len = bin.**size**();

**for** (**int** i = len – 1; i >= 0; i--) {

one = one + (a & c);

bin[i] = (a & c) ? 1 : 0;

c <<= 1;

}

**return** bin;

}

**/\* carmichael function \*/**

// el menor entero m tal que a^m == 1 (mod n)

**int** carmichaelLambda (**int** n) {

**int** ans = 1;

**if** (n % 8 == 0 )

n /= 2 ;

**for** (**int** d = 2 ; d <= n; ++d)

**if** (n % d == 0 ) {

**int** y = d - 1;

n /= d;

**while** (n % d == 0 )

n /= d, y \*= d;

ans = lcm(ans, y);

}

**return** ans;

}

**Función de Carmichael**

En Teoría de números, la función de Carmichael de un entero positivo n, denotada λ(n), se define como el menor entero m tal que cumple:

para cada número entero a coprimo con n.

Los primeros valores de λ(n) son 1, 1, 2, 2, 4, 2, 6, 2, 6, 4, 10, 2, 12, 6, 4, 4, 16, 6, 18, 4, 6, 10, 22, 2, 20, 12.

**Teorema de Carmichael**

Si a es un número coprimo con n, entonces donde es la función de Carmichael.

**/\* Máximo común divisor de a y b, coeficientes de la ecuación ax + by = d. \*/**

**int** extGCD(**int** a, **int** b, **int**\* x, **int**\* y) {

a = abs(a);

b = abs(b);

**if** (b == 0) {

\*x = 1;

\*y = 0;

**return** a;

}

**int** d, x1, y1;

d = GCD(b, a % b, &x1, &y1);

\*x = y1;

\*y = x1 - a / b \* y1;

**return** d;

}

**/\* Máximo común divisor de a y b \*/**

ll gcd (ll a, ll b) {

**while** ( b ) {

a %= b;

**swap**( a, b );

}

**return** a;

}

ll gcd (ll a, ll b) { **/\* Best way \*/**

**while** (a && b) {

**if** (a > b)

a %= b;

**else**

b %= a;

}

**return** a + b;

}

ll mcm(ll a, ll b) {

**return** (a \* b) / gcd(a, b);

}

**/\* Sieve of Eratosthenes (GET PRIMES) \*/**

**const** **int** N = 100000000;

**int** get\_prime() // get primes from 1 to N in liner time

{

//memset (prime, 0, **sizeof** (**int**) \* (N + 1));

**for** (**int** i = 2; i <= N; i++) {

**if** (!**prime**[i])

prime[++prime[0]] = i;

**for** (**int** j = 1; j <= prime[0] && prime[j] \* i <= N; j++) {

prime[prime[j]\*i] = 1;

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

}

}

**return** prime[0]; // size of prime

}

**/\* Sieve of Eratosthenes (ASK ISPRIME) \*/**

**const** **int** MAXN = 100000000;

**const** **int** P1 = (MAXN + 7) >> 4;

**const** **int** P2 = (MAXN + 1) >> 1;

**const** **int** P3 = 5000; // ceil(ceil(sqrt(MAXN))/2);

**char** sieve[P1] = {0};

#define GET(a) (~(sieve[(a) >> 3] >> ((a) & 7)) & 1)

#define isPrime(a) (a == 2 || (a & 1) && a > 2 && GET((a - 1) >> 1))

**inline** **void** make() {

**unsigned** **int** i, j, k;

**for** (i = 1; i <= P3; ++i) **if**(GET(i))

**for** (j = (i \* (i + 1)) << 1, k = (i << 1) + 1; j < P2; j += k)

sieve[j >> 3] |= 1 << (j & 7);

}

**MATRIX**