#include <cstdio>

#include <cmath>

#include <memory.h>

typedef long long typec;

///Lib functions

typec GCD(typec a, typec b)

{

return b ? GCD(b, a % b) : a;

}

typec extendGCD(typec a, typec b, typec& x, typec& y)

{

if(!b) return x = 1, y = 0, a;

typec res = extendGCD(b, a % b, x, y), tmp = x;

x = y, y = tmp - (a / b) \* y;

return res;

}

///for x^k

typec power(typec x, typec k)

{

typec res = 1;

while(k)

{

if(k&1) res \*= x;

x \*= x, k >>= 1;

}

return res;

}

///for x^k mod m

typec powerMod(typec x, typec k, typec m)

{

typec res = 1;

while(x %= m, k)

{

if(k&1) res \*= x, res %= m;

x \*= x, k >>=1;

}

return res;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Inverse in mod p^t system

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec inverse(typec a, typec p, typec t = 1)

{

typec pt = power(p, t);

typec x, y;

y = extendGCD(a, pt, x, y);

return x < 0 ? x += pt : x;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linear congruence theorem

x = a (mod p)

x = b (mod q)

for gcd(p, q) = 1, 0 <= x < pq

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec linearCongruence(typec a, typec b, typec p, typec q)

{

typec x, y;

y = extendGCD(p, q, x, y);

while(b < a) b += q / y;

x \*= b - a, x = p \* x + a, x %= p \* q;

if(x < 0) x += p \* q;

return x;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

prime table

O(n)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const int PRIMERANGE = 1000000;

int prime[PRIMERANGE + 1];

int getPrime()

{

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

for (int i = 2; i <= PRIMERANGE; i++)

{

if (!prime[i]) prime[++prime[0]] = i;

for (int j = 1; j <= prime[0] && prime[j] <= PRIMERANGE / i; j++)

{

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

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

}

}

return prime[0];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

get factor of n

O(sqrt(n))

factor[][0] is prime factor

factor[][1] is factor generated by this prime

factor[][2] is factor counter

need: Prime Table

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

///you should init the prime table before

int factor[100][3], facCnt;

int getFactors(int x)

{

facCnt = 0;

int tmp = x;

for(int i = 1; prime[i] <= tmp / prime[i]; i++)

{

factor[facCnt][1] = 1, factor[facCnt][2] = 0;

if(tmp % prime[i] == 0)

factor[facCnt][0] = prime[i];

while(tmp % prime[i] == 0)

factor[facCnt][2]++, factor[facCnt][1] \*= prime[i], tmp /= prime[i];

if(factor[facCnt][1] > 1) facCnt++;

}

if(tmp != 1)

factor[facCnt][0] = tmp, factor[facCnt][1] = tmp, factor[facCnt++][2] = 1;

return facCnt;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Easy Combination for

C(n, k) that not exceeds limit of typec

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combination(int n, int k)

{

typec res = 1, g;

if(k > n) return 0;

if(n - k < k) k = n - k;

for(int i = 0; i < k; i++)

{

if(res % (i + 1) == 0)

res /= i + 1, res \*= n - i;

else if( (n - i) % (i + 1) == 0)

res \*= (n - i) / (i + 1);

else

{

g = GCD(res, i + 1), res /= g;

res \*= (n - i) / ((i + 1) / g);

}

}

return res;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m for k < 50,

O(k\*k\*lgk) m \* m < typec\_MAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModS(typec n, typec k, typec m)

{

///larger gate for more optimization

///too large gate may overflow somewhere

const typec gate = 1LL << 50;

if(k > n || m == 1) return 0;

if(n - k < k) k = n - k;

typec d[k], tmp = 1, i = 0, j , h, g;

for(i = 0, j = n - k + 1; n - i >= j; i++)

{

d[i] = n - i;

while(gate / d[i] >= j && n - i != j)

d[i] \*= j, j++;

}

for( j = 2, h = k; j <= h; h--)

{

tmp \*= h;

while(gate /tmp >= j && h != j)

tmp \*= j, j++;

for(int s = 0; tmp != 1; s++)

{

g = GCD(tmp, d[s]);

d[s] /= g, tmp /= g;

}

}

int res = 1;

for(j = 0; j < i; j++)

d[j] %= m, res \*= d[j], res %= m;

return res;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m using prime table for sieving

O(n) m \* m < typecMAX

limited by primetable, n could not be too large

need: prime table

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModPri (typec n, typec k, typec m)

{

if(k > n || m == 1) return 0;

typec result = 1, cnt = 0, temp;

for(int i = 1; i < prime[0] && prime[i] <= n; i++)

{

temp = n, cnt = 0;

while(temp)

temp /= prime[i], cnt += temp;

temp = n - k;

while(temp)

temp /= prime[i], cnt -= temp;

temp = k;

while(temp)

temp /= prime[i], cnt -= temp;

temp = prime[i];

while(cnt)

{

if(cnt & 1)

result \*= temp, result %= m;

temp \*= temp, cnt >>= 1, temp %= m;

}

if(result == 0) return 0;

}

return result;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m

O(k\*lgm) m \* m < typec\_MAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModN(typec n, typec k, typec m)

{

if(k > n || m == 1) return 0;

k = (n - k < k)? n - k : k;

int pcnt = 0;

typec a = 1, b = 1, x, y, g;

typec pa = 1, pb = 1; ///may over flow

for(int i = 1; i <= k; i++)

{

a \*= n - i + 1, b \*= k - i + 1;

while( (g = GCD(a, m)) > 1) pa \*= g, a /= g;

while( (g = GCD(b, m)) > 1) pb \*= g, b /= g;

g = GCD(pa, pb), pa /= g, pb /= g;

while(pa % m == 0) pa /= m, pcnt++;

while(pb % m == 0) pb /= m, pcnt--;

b %= m, a %= m;

}

a \*= pa / pb, a %= m;

while(pcnt) return 0;

extendGCD(b, m, x, y);

if(x < m) x += m;

x \*= a, x %= m;

return x;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod p

O(k) p\*p <= typecMAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModP(typec n, typec k, typec p)

{

if(k > n) return 0;

if(n - k < k) k = n - k;

typec a = 1, b = 1, x, y;

int pcnt = 0;

for(int i = 1; i <= k; i++)

{

x = n - i + 1, y = i;

while(x % p == 0) x /= p, pcnt++;

while(y % p == 0) y /= p, pcnt--;

x %= p, y %= p, a \*= x, b \*= y;

b %= p, a %= p;

}

if(pcnt) return 0;

extendGCD(b, p, x, y);

if(x < 0) x += p;

a \*= x, a %= p;

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod p^t

O(k\*lgn/lgp) p^2t < typecMAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModPt(typec n, typec k, typec p, typec t)

{

if(k > n) return 0;

if(n - k < k) k = n - k;

typec pt = power(p, t);

typec a = 1, b = 1, x, y;

int pcnt = 0;

for(int i = 1; i <= k; i++)

{

x = n - i + 1, y = i;

while(x % p == 0) pcnt++, x /= p;

while(y % p == 0) pcnt--, y /= p;

x %= pt, y %= pt, a \*= x, b \*= y;

a %= pt, b %= pt;

}

if(pcnt >= t) return 0;

extendGCD(b, pt, x, y);

if(x < 0) x += pt;

a \*= x, a %= pt;

return a \* power(p, pcnt) % pt;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m

O(k\*lgn/lgp) m \* m < typecMAX

p is fractor of m (depends on the smallest one)

need:

prime table

factor table

combinationModPt()

linearCongruence

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

///you need to init the prime table

typec combinationModLi(typec n, typec k, typec m)

{

if(k > n || m == 1) return 0;

getFactors(m);

typec a, b, p, q;

for(int i = 0; i < facCnt; i++)

{

if(!i) a = combinationModPt(n, k, factor[i][0], factor[i][2]), p = factor[i][1];

else b = combinationModPt(n, k, factor[i][0], factor[i][2]), q = factor[i][1];

if(!i) continue;

a = linearCongruence(a, b, p, q), p \*= q;

}

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod p

Lucas's theorem for combination mod p

O(p \* lgn/lgp)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec lucas(typec n, typec k, typec p)

{

typec res = 1;

while (n && k && res)

{

res \*= combinationModP(n % p, k % p, p);

res %= p, n /= p, k /= p;

}

return res;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a = n \* (n - 1) \* ... \* (n - k + 1)

b = m \* (m - 1) \* ... \* (m - k + 1)

c = ? from input

if a \* c / b is an integer

this function will calculate this value module p^t

and the c input is moduled by p^t, be sure that gcd(c, p^t) = 1

O(len \* lgn/lgp) , p^2t < typecMAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

///the parameter &pcnt caches the factors consists of p

typec productQuotient(typec n, typec m, typec len, typec p, typec pt, typec &c, typec &pcnt)

{

if(!c || n < len) return c = 0;

typec &a = c, b = 1, x, y;

for(int i = 1; i <= len; i++)

{

x = n - i + 1, y = m - i + 1;

while(x % p == 0) x /= p, pcnt++;

while(y % p == 0) y /= p, pcnt--;

x %= pt, y %= pt, a \*= x, b \*= y;

a %= pt, b %= pt;

}

extendGCD(b, pt, x, y);

if(x < 0) x += pt;

a \*= x, a %= pt;

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod p^t

generalized Lucas's theorem for combination mod p

O(p^t \* lgn/lgp), p^2t < typecMAX

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec generalizedLucas(typec n, typec k, typec p, typec t)

{

if(k > n) return 0;

if(n - k < k) k = n - k;

if(t == 1) return lucas(n, k, p);

typec pt = power(p, t);

typec c = 1, pcnt = 0, ktable[100], ntable[100], ltable[100];

int cnt = 0;

for(; n || k; cnt++)

{

ktable[cnt] = k, ntable[cnt] = n, ltable[cnt] = k % pt;

n -= k % pt, k -= k % pt, n /= p, k /= p;

}

for(--cnt; c && cnt >= 0; cnt--)

productQuotient(ntable[cnt], ktable[cnt], ltable[cnt], p, pt, c, pcnt);

if(!c || pcnt >= t) return 0;

return c \* power(p, pcnt) % pt;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m

O(min(k, p^t \* lgn/lgp)) m \* m < typecMAX

p^t is fractor of m

need:

prime table

factor table

generalizedLucas

linearCongruence

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

///you need to init the prime table

typec combinationModLucas(typec n, typec k, typec m)

{

if(m == 1 || k > n) return 0;

if(n - k < k) k = n - k;

getFactors(m);

typec a, b, p, q;

for(int i = 0; i < facCnt; i++)

{

if(!i) a = generalizedLucas(n, k, factor[i][0], factor[i][2]), p = factor[i][1];

else b = generalizedLucas(n, k, factor[i][0], factor[i][2]), q = factor[i][1];

if(!i) continue;

a = linearCongruence(a, b, p, q), p \*= q;

}

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const typec PTMAX = 10000;

typec facmod[PTMAX];

void initFacMod(typec p, typec t = 1)

{

typec pt = power(p, t);

facmod[0] = 1 % pt;

for(int i = 1; i < pt; i++)

{

if(i % p) facmod[i] = facmod[i - 1] \* i % pt;

else facmod[i] = facmod[i - 1];

}

}

///you should init the facmod[] before

typec factorialMod(typec n, typec &pcnt, typec p, typec t = 1)

{

typec pt = power(p, t), res = 1;

typec stepCnt = 0;

while(n)

{

res \*= facmod[n % pt], res %= pt;

stepCnt += n / pt, n /= p, pcnt += n;

}

res \*= powerMod(facmod[pt - 1], stepCnt, pt);

return res %= pt;

}

typec combinationModPtFac(typec n, typec k, typec p, typec t = 1)

{

if(k > n || p == 1) return 0;

if(n - k < k) k = n - k;

typec pt = power(p, t), pcnt = 0, pmcnt = 0;

if(k < pt) return combinationModPt(n, k, p, t);

initFacMod(p, t);

typec a = factorialMod(n, pcnt, p, t);

typec b = factorialMod(k, pmcnt, p, t);

b \*= factorialMod(n - k, pmcnt, p, t), b %= pt;

pcnt -= pmcnt;

if(pcnt >= t) return 0;

a \*= inverse(b, p, t), a %= pt;

return a \* power(p, pcnt) % pt;

}

typec combinationModFac(typec n, typec k, typec m)

{

getFactors(m);

typec a, b, p, q;

for(int i = 0; i < facCnt; i++)

{

if(!i) a = combinationModPtFac(n, k, factor[i][0], factor[i][2]), p = factor[i][1];

else b = combinationModPtFac(n, k, factor[i][0], factor[i][2]), q = factor[i][1];

if(!i) continue;

a = linearCongruence(a, b, p, q), p \*= q;

}

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C(n, k) mod m with genelizedLucas and combinationModFac

O(min(k, p^t))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typec combinationModPtAuto(typec n, typec k, typec p, typec t)

{

if(t > 6) return combinationModPtFac(n, k, p, t);

return generalizedLucas(n, k, p, t);

}

typec combinationModOdds(typec n, typec k, typec m)

{

if(k > n || m == 1) return 0;

getFactors(m);

typec a, b, p, q;

for(int i = 0; i < facCnt; i++)

{

if(!i) a = combinationModPtAuto(n, k, factor[i][0], factor[i][2]), p = factor[i][1];

else b = combinationModPtAuto(n, k, factor[i][0], factor[i][2]), q = factor[i][1];

if(!i) continue;

a = linearCongruence(a, b, p, q), p \*= q;

}

return a;

}