```
long long powMod(long long N, long long P, long long M) {
        if(P==0)
                return 1;
        if(P%2==0) {
                long long ret = powMod(N, P/2, M)%M;
                return (ret * ret)%M;
        }
    return ((N%M) * (powMod(N, P-1, M)%M))%M;
}

unsigned long long Pow(unsigned long long N, unsigned long long P) {
        if(P == 0)
                return 1;
        if(P % 2 == 0) {
                unsigned long long ret = Pow(N, P/2);
                return ret*ret;
        }
    return N * Pow(N, P-1);
}

// calculate A mod B, where A : 0<A<(10^100000) (or greater)
// take input as string and process with aftermod
long long afterMod(string str, ll mod) {             // input as string, as it is big, mod is the Mod value (Mod-1 if
        long long ans = 0;                           // need mod an exponentiation)
        string :: iterator it;

        for(it = str.begin(); it != str.end(); it++)        // mod from first to last
                ans = (ans*10 + (*it -'0')) % mod;
    return ans;
}

// Extended Euclud
// a*x + b*y = gcd(a, b)
// Given a and b calculate x and y so that a * x + b * y = d   (where gcd(a, b) | c)
// x_ans = x + (b/d)n
// y_ans = y − (a/d)n
// Solution only exists if d | c (i.e : c is divisible by d)
ll gcdExtended(ll a, ll b, ll *x, ll *y) {              // C function for extended Euclidean Algorithm
    if (a == 0) {                                        // Base Case
        *x = 0, *y = 1;
        return b;
    }
    ll x1, y1;                                           // To store results of recursive call
    ll gcd = gcdExtended(b%a, a, &x1, &y1);
    *x = y1 - (b/a) * x1;
    *y = x1;
    return gcd;
}

ll modInverse(ll a, ll mod) {
    ll x, y;
    ll g = gcdExtended(a, mod, &x, &y);
    if (g != 1)                                           // ModInverse doesnt exist
        return -1;
    ll res = (x%mod + mod) % mod;          // m is added to handle negative x
    return res;
}
```