D. Double Agent Elite

Time limit: 5s
Memory limit: 2GB

The public enemy of the Flea Kingdom, the “Evil Flea Marshal”, has designed a new type of highly destructive mines, and many of them have been buried around the kingdom.

To completely defuse a mine, one must input the correct password. It is known that every mine’s password is a sequence of length m, where each element belongs to $\{1, 2, \ldots, n\}$.

To prevent the password from being cracked, the Evil Flea Marshal destroyed all passwords after setting them. The Flea Technology Institute managed to recover only very limited information.

For a given password, let c_i denote the number of occurrences of value i in the sequence. The institute only recovered the **pairwise comparisons** between c_{i-1} and c_i. Specifically:

You are given a clue sequence t of length $n - 1$, where each element is in $\{0, 1, 2\}$:

- $t_i = 0$ means $c_i < c_{i+1}$
- $t_i = 1$ means $c_i = c_{i+1}$
- $t_i = 2$ means $c_i > c_{i+1}$

After obtaining these clues, the military tech consultant “Flea Ford Hacker” claimed to recover all passwords in just 0.01s, restoring peace to the kingdom.

This suspicious speed raises doubts—you believe it is impossible to reconstruct all passwords so quickly with such limited information. You suspect that “Flea Ford Hacker” and the “Evil Flea Marshal” might actually be the same person!

To report this, you must gather evidence. Your task is:

For **every possible clue sequence** t, compute the number of valid original passwords consistent with t, modulo a prime p.

Input Format

A single line containing three integers:

$$n, m, p$$

Output Format

Due to the huge output size, a compressed output is required.

Let $f(t)$ denote the number of valid passwords corresponding to clue sequence t, modulo p.

You only need to output:

$$\bigoplus_{t \in \{0,1,2\}^{n-1}} (h(t) \times (f(t) + h(t)))$$

where

$$h(t) = \sum_{i=1}^{n-1} 3^{i-1} \cdot t_i$$

and $\oplus$ denotes bitwise XOR.

Sample Input 1

```
2 4 998244353
```

Sample Output 1

```
9
```

Explanation

Before compression:

- $t = [0]$ ($c_1 < c_2$): $f(t) = 5$
- $t = [1]$ ($c_1 = c_2$): $f(t) = 6$
- $t = [2]$ ($c_1 > c_2$): $f(t) = 5$

Sample Input 2

```
3 10 1000000007
```

Sample Output 2

```
112586
```

Explanation

Before compression:

- $t = [0, 0]$: $f(t) = 5365$
- $t = [1, 0]$: $f(t) = 5551$
- $t = [2, 0]$: $f(t) = 14132$
- $t = [0, 1]$: $f(t) = 3402$
- $t = [1, 1]$: $f(t) = 0$
- $t = [2, 1]$: $f(t) = 5551$
- $t = [0, 2]$: $f(t) = 16281$
- $t = [1, 2]$: $f(t) = 3402$
- $t = [2, 2]$: $f(t) = 5365$

Sample Input 3

```
5 100 998244353
```

Sample Output 3

```
67156055694
```

Sample Input 4

```
15 1000 1000000007
```

Sample Output 4

```
358612410956090
```

Constraints

For all test cases:

- $2 \leq n \leq 16$
- $1 \leq m \leq 1000$
- $0.99 \times 10^9 < p < 1.01 \times 10^9$
- p is a prime

Subtasks

Subtask	$n \leq$	$m \leq$	Score
1	6	10	2
2	8	30	3
3	10	100	5
4	12	100	10
5	12	300	10
6	12	1000	10
7	14	100	10

Subtask	$n \leq$	$m \leq$	Score
8	14	300	10
9	14	1000	10
10	16	100	10
11	16	300	10
12	16	1000	10

Note

A Barrett reduction template is provided to speed up modular multiplication:

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

using namespace std;

namespace FM{
    typedef unsigned long long ull;
    typedef __uint128_t L;
    struct FastMod{
        ull b, m;
        FastMod(ull b) : b(b), m(ull((L(1) << 64) / b)) {}
        ull reduce(ull a) {ull q = (ull)((L(m) * a) >> 64), r = a - q * b; return r >= b ? r - b
    };
    FastMod F(2);
}

unsigned int mod = 0;

struct modint{
    unsigned int x;
    modint() {x = 0;}
    modint(int o) {x = o >= 0 ? o : mod - (-o);}
    modint &operator = (int o) {return x = o, *this;}
    modint &operator += (modint o) {return x = x + o.x >= mod ? x + o.x - mod : x + o.x, *this;}
    modint &operator -= (modint o) {return x = x < o.x ? x - o.x + mod : x - o.x, *this;}
    modint &operator *= (modint o) {return x = FM :: F.reduce(1ull * x * o.x), *this;}
    modint &operator /= (modint o) {for(unsigned int t = mod - 2 ; t ; t >>= 1, o *= o) if(t & 1
    friend modint operator + (modint a, modint b) {return a += b;}
    friend modint operator - (modint a, modint b) {return a -= b;}
    friend modint operator * (modint a, modint b) {return a *= b;}
    friend modint operator / (modint a, modint b) {return a /= b;}
    modint operator - () {return x ? mod - x : 0;}
};

void initmod(){
    scanf("%u", &mod);
    FM :: F = FM :: FastMod(mod);
}
```

Warning: Do not abuse the judge (e.g., hacking with crafted inputs). Violations may result in account suspension.