Modulo

Stephen Huan
Edited by: Udbhav Muthakana

January 8, 2020

1 Definition

A number a ***modulo*** m is defined as the remainder after calculating $\frac{a}{m}$, written in math as $a \mod m \equiv c$, but represented in CS as $a \ \% \ m == c$. Modulo is defined for integral a and m (either can be negative), but not for $m = 0$. It is considered a constant time calculation for Big-O analysis.

1.1 Properties

$$\begin{aligned} a + b \mod m &\equiv (a \mod m) + (b \mod m) \mod m \\ a - b \mod m &\equiv (a \mod m) - (b \mod m) \mod m \\ a \cdot b \mod m &\equiv (a \mod m) \cdot (b \mod m) \mod m \end{aligned}$$

2 Applications

Problems that ask you to compute all possibilities of something will usually involve very large numbers, and thus ask you to compute the answer modulo $m = 1000000007$. Make sure to use

```
print(x if x >= 0 else x + m)
```

for output, as they expect the answer to be positive.

Modulo is used in many number theoretic algorithms, like the Euclidean algorithm to find the greatest common divisor of two integers.

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Modulo is also commonly used to implement "cyclic" behavior, because the values repeat with a period of m and $a \mod m \equiv 0$ implies a is divisible by m.

2.1 Modulo Exponentiation

It is possible to compute $b^e \mod m$ in $O(\log e)$. Decompose e into a binary number - write it in the form $\sum_{i=0}^{n-1} a_i 2^i$, where a_i is 1 or 0 depending the corresponding bit.

Then, by exponent rules,

$$b^e = b^{\sum_{i=0}^{n-1} a_i 2^i} = \prod_{i=0}^{n-1} b^{a_i 2^i} = \prod_{i=0}^{n-1} (b^{2^i})^{a_i}$$

In order to calculate b^{2^i}, note that

$$b^{2^i} = b^{2 \cdot 2^{i-1}} = b^{2^{i-1}+2^{i-1}} = (b^{2^{i-1}})^2$$

Start with the base case of $i = 0$, which is b, and then just square the base repeatedly.

The final algorithm is then

```
def mod_exp(b: int, e: int, m: int) -> int:
    if m == 1: return 0
    rtn = 1
    b %= m
    while e > 0:
        # if bit on in the binary representation of the exponent
        if e & 1 == 1:
            rtn = (rtn*b) % m
        e >>= 1
        b = (b*b) % m
    return rtn
```

2.2 Modular Multiplicative Inverse

We have seen how to perform addition, subtraction, multiplication, and exponentiation with modulos. How do we do division quickly? The solution is to find some number x such that $\frac{a}{b} \mod m \equiv a \cdot x \mod m$. This number x is then the ***modular multiplicative inverse*** of b. In order to compute x, we use the following algorithm.

```
def extended_gcd(a: int, b: int) -> int:
    s, sp, t, tp, r, rp = 0, 1, 1, 0, b, a

    while r != 0:
        q = rp//r
        rp, r = r, rp - q*r
        sp, s = s, sp - q*s
        tp, t = t, tp - q*t

    return sp
```

To use it,

```
(a/b) % m == (a*extended_gcd(b, m)) % m
```

3 Sample Problems

1. 2019 ICT In-house "Havish's Exponents"
   Solution: Direct application of modular exponentiation.

2. 2019 ICT In-house "Permutations":
   Find the number of permutations of a given string with length n.

   Solution: The number of permutations will be $n!$ divided by $(\prod_{ch} c_{ch})!$, where c_{ch} is the number of times a particular character ch appears in the string.

   Calculate $n!$ by doing repeated multiplication while modding to keep it small. Then, precompute the modular multiplicative inverse for each size i up to n. Finally, for each distinct character of the original string, count how many times it appears in the string and multiply n by the multiplicative modular inverse of each number i up to the count. Initially, this seems to be a n^2 algorithm, but since the sum of the counts is exactly n, the algorithm does $O(n)$ work.

4 Works Cited

1. Wikipedia - "Modular exponentiation"

2. Wikipedia - "Extended Euclidean algorithm"