CS 512: Computing Modular Inverses

16:198:512
Instructor: Wes Cowan

On quiz two, you were given the following facts:

> Fact 1: The multiplicative inverse of a mod N exists iff $\text{GCD}(a, N) = 1$.

> Fact 2: $\text{GCD}(a, b) = \text{GCD}(a, b \bmod a)$.

You were then asked to prove the following proposition:

$$\text{GCD}(a, b) = 1 \text{ iff for some } x, y \in Z,\ ax + by = 1.$$

It is worth noting that the most direct proof of this result is relatively straightforward, *given only Fact 1* and does not follow the outline as given in the quiz:

- By Fact 1, $\text{GCD}(a, b) = 1$ implies that the multiplicative inverse of a mod b exists. Call this x.
- We have $ax \equiv 1 \pmod{b}$.
- In other words, b divides $ax - 1$, or *for some integer* k, $ax - 1 = bk$. Taking $y = -k$, we have $ax + by = 1$.

The proof the other direction is just as straightforward - x is clearly the inverse of a mod b, and by its existence we have $\text{GCD}(a, b) = 1$.

The core idea of the proof is the notion that if $ax + by = 1$, it must by true that x is the inverse of a mod b, and y is the inverse of b mod a (and vice versa):

$$\text{GCD}(a, b) = 1 \iff a * \mathbf{inv}(a, b) + b * \mathbf{inv}(b, a) = 1.$$

Note, as an immediate property of this, we get the following relationship that (if it exists),

$$\mathbf{inv}(a, b) = [1 - b * \mathbf{inv}(b, a)]/a.$$

This proof is direct, simply chasing definitions, but it does not lend itself well to the development of an algorithm for computing these values - if we have one inverse value, we can get the other, but how do we get the first? Obviously, we could simply utilize the **ExtendedEuclideanAlgorithm** to compute (x, y) given such an (a, b) - note, this approach implies that computing the modular inverse *is no more complex than the extended euclidean algorithm.* But modifying the proof yields an alternative approach.

This leads to the proof as outlined in the paper - which utilizes Fact 2 to reduce the problem to a smaller version of the same problem (the basis of any good recursion) but can utilize the same proof structure:

- Let $c = b \bmod a$, i.e., the remainder when b is divided by a.
- By Fact 2, we have that $\text{GCD}(a, b) = \text{GCD}(a, c) = 1$.
- By Fact 1, we have that $\text{GCD}(a, c) = 1$ implies that the inverse of c mod a exists. Call this x'.
- We have that $cx' \equiv 1 \pmod{a}$.

- In other words, a divides $cx' - 1$, or *for some integer* j, $cx' - 1 = aj$. Taking $y' = -j$, we have

$$cx' + ay' = 1.$$

- Observing that $c = b - \lfloor b/a \rfloor a$ (why?), this gives

$$(b - \lfloor b/a \rfloor a)\, x' + ay' = 1,$$

  or

$$a(y' - \lfloor b/a \rfloor x') + bx' = 1.$$

- Taking $x = y' - \lfloor b/a \rfloor x'$ and $y = x'$, we have $ax + by = 1$, as desired.

Note the following relations, that $x' = \mathbf{inv}(c, a)$ and $y' = \mathbf{inv}(a, c)$. Hence,

$$\mathbf{inv}(a, b) = \mathbf{inv}(a, c) - \lfloor b/a \rfloor \mathbf{inv}(c, a)$$

and

$$\mathbf{inv}(b, a) = \mathbf{inv}(c, a).$$

The second result is important because it gives an important rule for 'switching' arguments when computing inverses, i.e., $\mathbf{inv}(b, a) = \mathbf{inv}(b \bmod a, a)$; in the case the larger argument is given first, you can reduce it to the case that the smaller argument is given first.

At this point, development of an algorithm could go a number of ways. You could use the above relations to define a recursive solution, but notice that (subject to some base cases), computing **inv** would require (in the worst case) two recursive invocations of the **inv** function. This seems inefficient. However, combining this argument-switching relation with the earlier relation, we get:

$$\mathbf{inv}(a, b) = \frac{1 - b * \mathbf{inv}(b \bmod a, a)}{a}.$$

Note in this case, we are computing **inv** in *one* recursive call, with strictly smaller arguments. This is an excellent basis for a recursive computation. Including some Important base cases - $\mathbf{inv})1, x) = 1$, and $\{\mathbf{inv}(x.0), \mathbf{inv}(0, x)\}$ don't exist (why?), we get the following algorithm:

```
def inv(a,b):
    if a == 1:
        return 1
    if a == 0 or b == 0:
        return 0

    return (1 - b*inv(b mod a, a) ) / a
```

Note, this algorithm includes an implicit execution of the Euclidean algorithm for computing the GCD (why?) - thus cannot beat it in complexity. But it has additional computational overhead in terms of the extra multiplication by b and divisuion by a as well.