CMPSCI 250: Introduction to Computation

Lecture #13: Modular Arithmetic
David Mix Barrington
19 February 2014

Modular Arithmetic

- Arithmetic on Congruence Classes
- Verifying the Operations
- The Euclidean Algorithm
- Proving that the EA Gives the GCD
- The Inverse Algorithm
- Practicality for Large Inputs

Arithmetic on Congruence Classes

- We've seen that the **congruence relation** for any modulus is an equivalence relation, meaning that it divides the naturals into equivalence classes called **congruence classes**.
- Modulo 3, for example, there are three classes: {0, 3, 6, 9,...}, {1, 4, 7, 10,...}, and {2, 5, 8, 11,...}. Modulo k, there are k classes.
- We'll now develop a new kind of arithmetic by treating these classes as numbers.

Arithmetic on Classes

- We can add classes -- if I take any two numbers in {1, 4, 7,...}, for example, their sum will be in {2, 5, 8,...}.
- There is an addition operation on classes, because it doesn't matter which element of the input classes we take as long as we only care about the class of the output.
- The same thing works for multiplication, as we'll soon show. We can add, subtract, and multiply classes. But division is different!

Verifying the Operations

- The statement that we can add classes can be written in logical symbols as follows:
- $\forall m: \forall a: \forall b: \forall c: \forall d: (a \equiv b \pmod{m}) \wedge (c \equiv d \pmod{m}) \rightarrow (a+c \equiv b+d \pmod{m})$.
- This says that if we change elements of the input classes in any possible way, the output *class* does not change.

The Proof for Addition

- To prove this, we let m, a, b, c, and d be arbitrary and assume that $a \equiv b$ and $c \equiv d$ modulo m.
- This means that $a = b + im$ and $c = d + jm$ for some integers (possibly negative) i and j.
- Then by arithmetic, adding the two equations, we get $a + c = b + d + (i + j)m$, and we have shown that $a + c \equiv b + d \pmod{m}$.

The Other Operations

- The multiplication statement and proof are the same. Starting from the same assumptions, we compute that $ac = (b + im)(d + jm) = bd + (id + bj + ijm)m$, and so we know that $ac \equiv bd \pmod{m}$.
- Subtraction works just like addition. But what about division?

Greatest Common Divisors

- The **greatest common divisor** of two naturals is the largest number that divides both of them. For example, gcd(9, 15) = 3.
- Two naturals are **relatively prime** if their gcd is 1. A prime number like 7 is relatively prime to any natural except one of its own multiples. Composite numbers, like 9 and 25, can be relatively prime to each other.

Clicker Question #1

- A set of more than two naturals is called **pairwise relatively prime** if every pair of two different naturals taken from the set are relatively prime. Which of these sets is *not* pairwise relatively prime?
- (a) {13, 15, 21}
- (b) {7, 11, 25}
- (c) {9, 10, 11}
- (d) {8, 49, 65}

Answer #1

- A set of more than two naturals is called **pairwise relatively prime** if every pair of two different naturals taken from the set are relatively prime. Which of these sets is *not* pairwise relatively prime?
- *(a) {13, 15, 21} (3 divides both 15 and 21)*
- (b) {7, 11, 25}
- (c) {9, 10, 11}
- (d) {8, 49, 65}

The Euclidean Algorithm

- The **Euclidean Algorithm** takes two positive naturals as input and determines their gcd, and hence whether they are relatively prime.
- The idea is simple -- at any time during the algorithm you have two naturals. You divide the smaller one into the larger and take the remainder. Your two new numbers are the smaller one and the remainder.
- You keep going until one number is 0.

Euclidean Algorithm Examples

- If we start with 14 and 8, we take 14 % 8 = 6, and our next pair is 8 and 6. Then 8 % 6 = 2, so we have 6 and 2. Finally 6 % 2 = 0. The gcd is the last number we have before we get 0 -- in this case gcd(14, 8) = 2.
- But if we start with 17 and 7, we take 17 % 7 = 3, so our next pair is 7 and 3. Then 7 % 3 = 1, so we have 3 and 1. Finally 3 % 1 = 0. The last number before 0 was 1, so gcd(17, 7) = 1 and we see that 17 and 7 are relatively prime.

Some Longer EA Examples

- We can carry out this procedure on any two numbers, without a computer or calculator as long as we can divide one natural by another.
- The procedure has to stop at some point because the numbers only get smaller (though proving that will take induction). Sometimes there are big jumps downward, sometimes (as at right) we take a while to get to 0.

```
119 % 65 = 54
65 % 54 = 11
54 % 11 = 10
11 % 10 = 1
10 % 1 = 0
gcd(119, 65) = 1
```

```
610 % 233 = 144
233 % 144 = 89
144 % 89 = 55
89 % 55 = 34
55 % 34 = 21
34 % 21 = 13
21 % 13 = 8
13 % 8 = 5
8 % 5 = 3
5 % 3 = 2
3 % 2 = 1
2 % 1 = 0
gcd(610, 233) = 1
```

Some More EA Examples

```
1001 % 417 = 167
417 % 167 = 83
167 % 83 = 1
83 % 1 = 0
gcd(1001, 417) = 1
```

```
1001 % 418 = 165
418 % 165 = 88
165 % 88 = 77
88 % 77 = 11
gcd(1001, 418) = 11
```

```
1001 % 416 = 169
416 % 169 = 78
169 % 78 = 13
78 % 13 = 0
gcd(1001, 416) = 13
```

```
1001 % 415 = 171
415 % 171 = 73
171 % 73 = 25
73 % 25 = 23
25 % 23 = 2
23 % 2 = 1
2 % 1 = 0
gcd(1001, 415) = 1
```

```
119 % 77 = 42
77 % 42 = 35
42 % 35 = 7
35 % 7 = 0
gcd(119, 77) = 7
```

Proving that EA Gives the GCD

- How can we be confident that this algorithm actually provides the gcd?
- Let a and b be the two original numbers, and let g be the real gcd. Let r be the result of the Euclidean Algorithm, the last number before 0.
- Since g divides both a and b, it also divides the third number, which is a - qb for some number q. By the same reasoning, g divides all the numbers that occur in the algorithm, and so divides r.

Proving that EA Gives the GCD

- The next-to-last number z in the algorithm is a multiple of r, since dividing it by r gave 0 remainder. Look at the number before -- dividing it by z gave r, so it is zq + r for some q, and hence also a multiple of r. Working backward, every number in the EA is a multiple of r, including the original a and b.
- So r is a common divisor, and the greatest common divisor g divides it -- this can only be true if r and g are the same number.

Multiplicative Inverses

- Now back to division. What would it mean to divide one class by another?
- When we divide one real number x by another (nonzero) real number y, we are multiplying x by the **multiplicative inverse** of y, written as y^{-1} or 1/y.
- Multiplication by y and multiplication by y^{-1} are **inverse functions**, because one undoes the other.

The Inverse of a Class

- So dividing one congruence class [x] by another class [y] means finding a class [z] such that multiplication by [z] undoes multiplication by [y] -- then the class “[x]/[y]” can be defined as [x][z] or [xz].
- For example, modulo 7, [3] has the inverse [5], because [3 × 5] = [15] = [1], since 15 ≡ 1 (mod 7).

Clicker Question #2

- We have just defined x to be "the inverse of y, mod m" if xy is congruent to 1, modulo m. Which of the following statements is not true?
- (a) 7 is its own inverse, mod 16
- (b) 7 is the inverse of 3, mod 10
- (c) 7 is the inverse of 10, mod 69
- (d) 7 is the inverse of 8, mod 19

Answer #2

- We have just defined x to be "the inverse of y, mod m" if xy is congruent to 1, modulo m. Which of the following statements is not true?
- (a) 7 is its own inverse, mod 16
- (b) 7 is the inverse of 3, mod 10
- (c) 7 is the inverse of 10, mod 69
- *(d) 7 is the inverse of 8, mod 19 ($7 \cdot 8 = 56 \equiv -1$)*

The Inverse Theorem

- We don't always have inverses, though, just as 0 has no multiplicative inverse in the real numbers.
- The **Inverse Theorem** says that a number z has an inverse modulo m if and only if z and m are relatively prime.
- The Euclidean Algorithm lets us test whether gcd(z, m) = 1, and with a little more work it will also let us prove the Inverse Theorem and find inverses when they exist.

Proving the Inverse Theorem

- First note that one half of the Inverse Theorem (gcd > 1 → no inverse) is easy to prove.
- If z and m have a common divisor g that is greater than 1, then g will always divide az + bm for any integers a and b, and so g will divide anything congruent to az modulo m.
- Since g doesn't divide 1, [az] can't be [1] and thus [z] has no inverse.

The Inverse Algorithm

- We prove the other half by finding the inverse when z and m are relatively prime.
- First note that each of our equations in the Euclidean Algorithm, such as “z % m = y”, can be rewritten “z = km + y” for some natural k.
- We can use these equations to write each of the numbers in the Euclidean Algorithm as a **linear combination** of z and m, that is, an expression of the form az + bm where a and b are integers.

The Inverse Algorithm

- Since 1 is one of these numbers when z and m are relatively prime, we will wind up with 1 = az + bm for some a and b.
- But then we can see that az ≡ 1 (mod m) and thus [a] is the inverse of [z] modulo m.
- Let's work this out in an example, to find the inverse of 65 modulo 119.

An Inverse Theorem Example

- We take the EA equations and rewrite them to express each new number in terms of the preceding two numbers. Then we express each number as a linear combination of 119 and 65.
- The first two are obvious. For the third, we use the fact that 11 is 65 - 1×54 and make a new combination for 11 by subtracting the combination for 54 from the one for 65.

```
119 % 65 = 54       119 = 1×65 + 54       119 = 1×119 + 0×65
65 % 54 = 11        65 = 1×54 + 11        65 = 0×119 + 1×65
54 % 11 = 10        54 = 4×11 + 10        54 = 1×119 - 1×65
11 % 10 = 1         11 = 1×10 + 1         11 = -1×119 + 2×65
10 % 1 = 0          10 = 10×1 + 0         10 = 5×119 - 9×65
                                          1 = -6×119 + 11×65
```

Continuing the Example

- To get a combination for 10, we use 10 = 54 - 4×11 by subtracting four times the combination for 11 from the combination for 54.
- We find that -6 is an inverse for 119 modulo 65, and 11 an inverse for 65 modulo 119.

```
119 % 65 = 54        119 = 1×65 + 54      119 = 1×119 + 0×65
65 % 54 = 11         65 = 1×54 + 11       65 = 0×119 + 1×65
54 % 11 = 10         54 = 4×11 + 10       54 = 1×119 - 1×65
11 % 10 = 1          11 = 1×10 + 1        11 = -1×119 + 2×65
10 % 1 = 0           10 = 10×1 + 0        10 = 5×119 - 9×65
                                          1 = -6×119 + 11×65
```

Clicker Question #3

- Suppose we are using the Inverse Algorithm to compute the inverse of 12, modulo 19. Our first two linear combinations are "$19 = 1 \cdot 19 + 0 \cdot 12$" and "$12 = 0 \cdot 19 + 1 \cdot 12$". What is the third linear combination?
- (a) $1 = -5 \cdot 19 + 8 \cdot 12$
- (b) $-5 = 1 \cdot 19 - 2 \cdot 12$
- (c) $7 = -1 \cdot 19 + 2 \cdot 12$
- (d) $7 = 1 \cdot 19 - 1 \cdot 12$

Answer #3

- Suppose we are using the Inverse Algorithm to compute the inverse of 12, modulo 19. Our first two linear combinations are "$19 = 1 \cdot 19 + 0 \cdot 12$" and "$12 = 0 \cdot 19 + 1 \cdot 12$". What is the third linear combination?
- (a) $1 = -5 \cdot 19 + 8 \cdot 12$
- (b) $-5 = 1 \cdot 19 - 2 \cdot 12$
- (c) $7 = -1 \cdot 19 + 2 \cdot 12$
- *(d) $7 = 1 \cdot 19 - 1 \cdot 12$*

Practicality for Large Inputs

- We have a general algorithm to test whether a number is prime, but it is wholly impractical for very large inputs.
- If a number has 100 digits, we would have to check every possible prime divisor up to its square root, which would be a number of about 50 digits.
- Since a sizable fraction of all such numbers are prime, this would take us eons even if we could test a trillion per second.

Practicality for Large Inputs

- There are better ways to **test for primality**, mentioned in CMPSCI 501.
- The most practical one is **randomized**, and actually has a small chance of falsely claiming that a composite number is prime.
- But **factoring** appears to be an even harder problem -- if I multiply two 100-digit primes together, there is no practical method known to get the factors back.

Practicality For Large Inputs

- By contrast, testing *relative* primality is very practical even for very large inputs (once you have a data structure to work with numbers too big for an `int` or a `long`).
- We'll see later in the course that on inputs with n digits, the Euclidean Algorithm takes O(n) time -- on inputs of 100 digits it will take a few hundred steps at worst. The worst case is when the inputs are **Fibonacci numbers**, as in our example of 610 and 233.