CSC 580
Cryptography and Computer Security

Math Basics for Cryptography

January 25, 2018

Overview

Today: Math basics (Sections 2.1-2.3)

To do before Tuesday:

- Complete HW1 problems
- Read Sections 3.1, 3.2 (can skip Hill Cipher), and 3.5

Longer term:

- Talk to classmates about teams for research project

The Big Picture...

Messages are typically strings of symbols from a finite alphabet

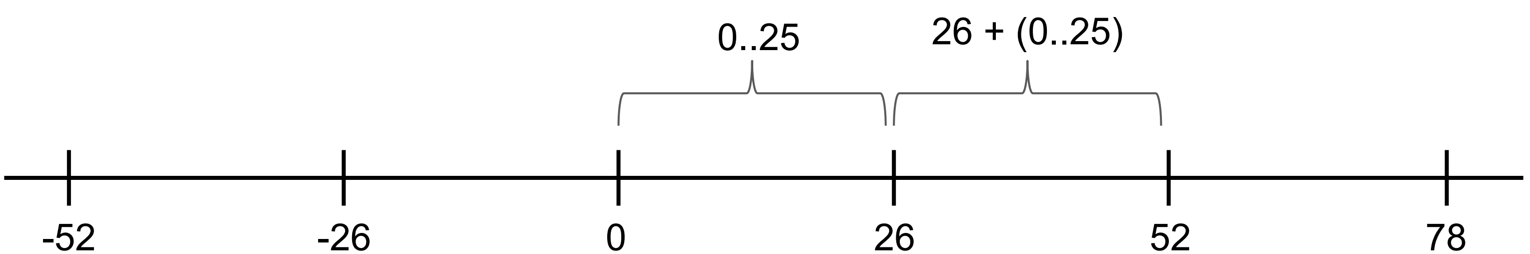
- Strings from the set of 26 letters (“classical cryptography”)
- Strings of bytes (256 possible values for each byte)
- Strings of larger blocks (e.g., 128-bit blocks for AES)

<u>Problem</u>: Doing arithmetic with values takes you out of the allowed range

- Caesar cipher adds 3 to each letter: 24 + 3 = *<u>27</u>* ← oops - not a valid letter!

<u>Solution</u>:

- View infinite number line in “pieces” of appropriate size
- All pieces give different representatives of same alphabet
- So above, 27=26+1 is treated the same as 1



<u>Modular arithmetic</u> - more useful than just “working with a finite alphabet”

You have all seen this before: Do you remember where?

Some Basic Ideas and Definitions

Divisibility, multiples, divisors, ...

Terminology: For integers a, b, and m, if a=m*b then

- a is a **multiple** of b
- b **divides** a   (written b | a)
- b is a **divisor** of a
- b is a **factor** of a

Every integer has a set of positive divisors (incl. at least 1)

- Example 1: Divisors of 15 are 1,3,5,15
- Example 2: Divisors of 18 are 1, 2, 3, 6, 9, 18
- Often interested in greatest common divisor (gcd(15,18)=3)

Modular Arithmetic

Definitions and some basic properties

For any a and b, there is a unique r such that

$a = q*b + r$, where $0 \le r < b$ (and $q = \lfloor a/b \rfloor$)

- q is the **quotient**
- r is the **remainder**

Two related notions:

- mod as a binary operator
  - a mod b is the remainder of a divided by b
  - 7 mod 5 = 2 ; 24 mod 7 = 3 ; 27 mod 9 = 0
- mod as a congruence relation
  - $a \equiv b \pmod{n}$ if and only if $(a-b) \mid n$
  - $7 \equiv 12 \pmod{5}$; $24 \equiv 3 \pmod{7}$; $128 \equiv 428 \pmod{100}$

Modular Arithmetic

Definitions and some basic properties

For any a and b, there is a unique r such that

$a = q*b + r$, where $0 \leq r < b$ (and $q = \lfloor a/b \rfloor$)

- q is the **quotient**
- r is the **remainder**

Warning: Best to always work with non-negative numbers with mod. Some languages (like C) say mod definition on negative numbers is "implementation dependent" (with certain restrictions - but it's unpredictable!).

Two related notions:

- mod as a binary operator
  - a mod b is the remainder of a divided by b
  - 7 mod 5 = 2 ; 24 mod 7 = 3 ; 27 mod 9 = 0
- mod as a congruence relation
  - $a \equiv b \pmod{n}$ if and only if $(a-b) \mid n$
  - $7 \equiv 12 \pmod{5}$; $24 \equiv 3 \pmod{7}$; $128 \equiv 428 \pmod{100}$

Greatest Common Divisor

A very important algorithm!

Numbers a and b are **relatively prime** if gcd(a,b) = 1

How to compute gcd fast?

Euclid's Algorithm

Assuming a > b:

```
gcd(a,b):
   if (b | a) then return b
   else return gcd(b, (a mod b))
```

Running time: O(log b)

Example: gcd(522,64)

a	b	(a mod b)
522	64	10
64	10	4
10	4	2
4	2	0

a mod b = 0 means b | a, so done

Final answer gcd(522,64) = 2

You try one:

Compute gcd(77,64)

Modular Arithmetic
A very important property

If you want the result of an algebraic formula modulo n, it doesn't matter if you do the mod operation mid-computation or just at the end!

So ((x*y+321)*71+z) mod n = ((x*y) mod n + 321)*71 + z) mod n

<u>*Application*</u>: Keep all intermediate results small

<u>*Example*</u>: I want to compute 1234^{16} mod 10000

- 1234^{16} is 50 digits long $\rightarrow$ overflows 64-bit integer
- Note that $1234^{16} = (((1234^2)^2)^2)^2$
- Can do $(((1234^2 \bmod 10000)^2 \bmod 10000)^2 \bmod 10000)^2 \bmod 10000$
- No intermediate result can be larger than 9999^2 = 99,980,001 (8 digits)

Modular Arithmetic

Other properties of modular addition

The “mod 7” addition table (notice how easy to do in Python!)

```
>>> np.asmatrix([[(i+j)%7 for j in range(7)] for i in range(7)])
matrix([[0, 1, 2, 3, 4, 5, 6],
        [1, 2, 3, 4, 5, 6, 0],
        [2, 3, 4, 5, 6, 0, 1],
        [3, 4, 5, 6, 0, 1, 2],
        [4, 5, 6, 0, 1, 2, 3],
        [5, 6, 0, 1, 2, 3, 4],
        [6, 0, 1, 2, 3, 4, 5]])
```

Properties

- 0 is the “identity” (for every x, 0 + x mod 7 = x)
- Each row/column contains all values, shifted by an appropriate amount
  - Each row/column includes a 0 → each element has an additive inverse
- Not obvious from table, but: operation is associative and commutative

Note: These properties hold for any modulus, not just 7

Modular Arithmetic

Other properties of modular multiplication

The “mod 7” multiplication table

```
>>> np.asmatrix([[(i*j)%7 for j in range(7)] for i in range(7)])
matrix([[0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6],
        [0, 2, 4, 6, 1, 3, 5],
        [0, 3, 6, 2, 5, 1, 4],
        [0, 4, 1, 5, 2, 6, 3],
        [0, 5, 3, 1, 6, 4, 2],
        [0, 6, 5, 4, 3, 2, 1]])
```

Properties of the “mod 7” multiplication table - for all elements except 0:

- 1 is the “identity” (for every x, 1 * x mod 7 = x)
- Each row/column contains all values, permuted
    - Each row/column includes a 1 → each element has a multiplicative inverse

Not obvious from table, but: operation is associative and commutative

Do these properties hold for any modulus?

Modular Arithmetic

Other properties of modular multiplication

The “mod 8” multiplication table

```
>>> np.asmatrix([[(i*j)%8 for j in range(8)] for i in range(8)])
matrix([[0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6, 7],
        [0, 2, 4, 6, 0, 2, 4, 6],
        [0, 3, 6, 1, 4, 7, 2, 5],
        [0, 4, 0, 4, 0, 4, 0, 4],
        [0, 5, 2, 7, 4, 1, 6, 3],
        [0, 6, 4, 2, 0, 6, 4, 2],
        [0, 7, 6, 5, 4, 3, 2, 1]])
```

Row doesn’t contain a 1!

Next: Try a few more moduli in Python… What’s the pattern for rows with 1’s?

Modular Arithmetic

Other properties of modular multiplication

The “mod 8” multiplication table

```
>>> np.asmatrix([[(i*j)%8 for j in range(8)] for i in range(8)])
matrix([[0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 2, 3, 4, 5, 6, 7],
        [0, 2, 4, 6, 0, 2, 4, 6],
        [0, 3, 6, 1, 4, 7, 2, 5],
        [0, 4, 0, 4, 0, 4, 0, 4],
        [0, 5, 2, 7, 4, 1, 6, 3],
        [0, 6, 4, 2, 0, 6, 4, 2],
        [0, 7, 6, 5, 4, 3, 2, 1]])
```

Row doesn’t contain a 1!

Next: Try a few more moduli in Python… What’s the pattern for rows with 1’s?

Answer: Row x has a 1 (i.e., x has a mult inverse) if and only if x is relatively prime to the modulus.

Important fact: Can use the “Extended Euclidean” algorithm to find x’s inverse mod n in O(log n) time. (details in book)

Number Sizes
Estimating with powers of two

Important values to know cold:

- 2^{10} is “about 1000” (actually 1024)
- 2^{20} is “about a million” (actually 1,048,576)
- 2^{30} is “about a billion”
- 2^{40} is “about a trillion”
- …

And the converse for dealing with base 2 logarithms:

- $\log_2(1000)$ is about 10
- $\log_2(1{,}000{,}000)$ is about 20
- $\log_2(1{,}000{,}000{,}000)$ is about 30
- ...

Number Sizes

Using for quick estimates - crypto example

Consider a "key cracking" machine that is clocked at 1 GHz, so can test 1 billion keys per second.

Attacking a cipher with 40-bit keys.

Question: How long to test all possible keys?

1. A billion keys/second is about 2^{30} keys/second
2. There are 2^{40} different 40-bit keys
3. Time required is then $2^{40} / 2^{30} = 2^{10}$ seconds
4. 2^{10} seconds is about 1,000 seconds
5. An hour has 3,600 seconds, so this is just a little over 15 minutes (not a very secure cipher!)

Number Sizes

More precise estimates

Know powers of 2 up to 2^{10} - a few important ones:

- $2^4 = 16$
- $2^5 = 32$
- $2^8 = 256$

Examples:

- What is 2^{25}? $2^{20} \cdot 2^5$ = approx 32 million
- What is 2^{38}? $2^{30} \cdot 2^8$ = approx 256 billion

Relation to a few other measures:

- One hour is 3,600 seconds, which is approx 2^{12}
- One day is 86,400, which is approx 2^{16} (closer: $2^{16.4}$)
- One year is approx 2^{25} seconds

So 8 trillion cycles on a 1GHz machine takes:

$2^{43} / 2^{30} = 2^{13}$ seconds $\rightarrow$ about 2 hours

Number Sizes
Algorithm understanding example

Need the multiplicative inverse of a number with 55-bit modulus

"Counting down" algorithm:

- For modulus n takes time $\Theta(n)$ time
- $n = 2^{55} \rightarrow 2^{55}$ computational steps
- At a billion steps / second $\rightarrow 2^{55}/2^{30} = 2^{25}$ seconds (1 year)

Euclid's algorithm:

- For modulus n, takes time $O(\log n)$ (specifically, $< 2 * \log_2(n)$ steps)
- n is $2^{55} \rightarrow$ less than 2*55 = 110 steps
- At a billion steps / second $\rightarrow$ Less than a millionth of a second

Your turn!

DES (which we'll look at next week) uses a 56-bit key. In 1998 a machine ("Deep Crack") was built that could test 90 billion keys per second.

How long does it take to test all keys? (Hint: round values sensibly!)

[redacted]

Number Sizes
Moore's Law

Moore's Law states that computing power doubles approximately every 18 months (1.5 years).

Example use:

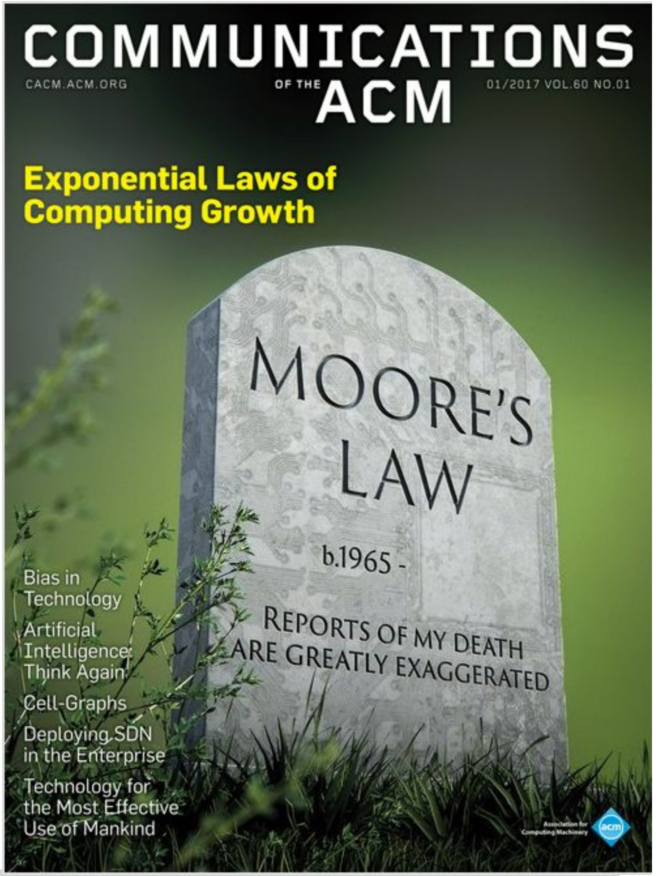
9 years from now, we will have had 6 "doublings", so computing power will be $2^6 = 64$ times faster than today.

Can this continue indefinitely?

No.

Are we near the end of Moore's Law?

Opinions vary....



Your turn #2! Moore’s Law and flipped around

A reasonable “clock speed” today is around 2-4 GHz, so assume that is the lower bound for a single core to test a key (really takes longer).

Custom hardware can give you a speed boost of, say, a million times.

Question: Assuming Moore’s Law continues, how many bits should a key have to be safe for the next 30 years? What if you wanted an extra “cushion” of a factor of 1000?

Number Sizes

Some really big numbers (impress your friends!)

Handout: “Large Numbers” from *Applied Cryptography* (Schneier)

Fun with large numbers….

- Randomly guessing a DES key: Probability of getting the correct key is half the probability of “winning the top prize in a U.S. state lottery and being killed by lightning in the same day.”
- Time to go through all 128-bit values at 1 trillion/second
  2^{128} / 2^{40} = 2^{88} seconds (or $2^{88}/2^{25}$ = 2^{53} years … or $2^{53}/2^{30}$ = 2^{23} or 8 million times the “time until the sun goes nova”)
- Factoring 1024-bit numbers (for breaking a small RSA key)
  Idea: Can we make a table of all prime factorizations?
  2^{1024} entries in the table. 2^{265} atoms in the universe. So not even remotely within the realm of possibility.

Number Sizes

Some really big numbers (impress your friends!)

A final thing to think about:

Finding a multiplicative inverse with a 2048-bit modulus is a very common operation in cryptography.

If we didn't know Euclid's algorithm, how long would the "counting down" algorithm take?