# CSCI3381-Cryptography

Lecture 6: Basic Number Theory

September 29, 2014

Practically everything is in the book, Sections 3.1-3.7, Section 6.3 on the Miller-Rabin primality test, and Section 6.1 on raw RSA. Later in the semester we will return to Section 3.9. In the next unit we will look at RSA in a more practical setting. These notes cover the major points, with a bit more emphasis on computational matters.

This is the number theory we need to explain RSA. We will return to number theory later when we discuss primitive elements and discrete logs. This is what we will require to explain Diffie-Hellman key exchanges, and the Digital Signature Algorithm.

## 1 Big Numbers, Hard and Easy Algorithms

- Public-key cryptography depends on exact arithmetic with big integers. For our purposes 'big integers' have between a hundred and a few thousand decimal digits. They are **too big to count up to** by any conceivable assemblage of computers, but **not too big to write down**.

- For computational purposes can measure the *size* of an integer $N$ in two different ways, either by its absolute magnitude $N$ or by the *length* of $N$: this is the number of digits $\log N$. We mean $\log_{10} N$ when we talk about decimal digits, and $\log_2 N$ when we talk about bits, but these only differ by a small constant factor ($\log_2 10 \approx 3.3$.)

- *Easy* algorithms are those whose running time is measured by the length—something like $(\log N)^\alpha$ basic steps, where $\alpha > 0$. *Hard* algorithms are those whose running time is measured by the magnitude, typically $N^\alpha$ steps, where $\alpha > 0$.

**Example 1** *Multiplication and Division.* If we have to multiply two integers with $k$ digits each, the standard grade-school algorithm requires about $k^2$ steps, or $2k^2$ if you count both one-digit multiplications and one-digit additions. This is $(\log N)^2$ where $N$ is the magnitude, so *multiplication is easy.* Integer division to compute the quotient and remainder similarly requires about $(\log N)^2$ steps.

**Example 2** *Naïve Primality Testing and Factorization.* To test if $N$ is prime, or to find a factor of $N$, divide $N$ by every integer $1 < m < \sqrt{N}$. If no factor is found, $N$ is prime.

This requires $\sqrt{N}$ divisions, each of which requires approximately $(\log N)^2$ steps, so the running time is proportional to $(\log N)^2 N^{\frac{1}{2}}$. The crucial factor is $N^{\frac{1}{2}}$, which makes this a *hard algorithm.*

**Example 3** *Square Roots.* To find the closest integer to $\sqrt{N}$, repeatedly split the interval from 1 to $N$ in half, computing the midpoint $m$ of the interval at each iteration, and continuing in the right subinterval or the left subinterval depending on whether or not $m^2 > N$. (This is called the *bisection method*, a version of *binary search.* ) After $\log_2 N$ iterations, successive midpoints differ by less than 1, which gives us the closest integer. We have one multiplication, requiring $(\log N)^2$ steps at each iteration, so the total running time is proportional to $(\log N)^3$. Thus *finding square roots is easy.* This is in contrast to the problem of computing square roots mod $n$, which we will take up later.

# 2 Three Fundamental Facts

1. *There are a lot of primes.* Euclid's *Elements* (c. 300 BC) contains a proof that there are infinitely many primes. If you study tables of prime numbers, you find that primes get rarer and rarer as the magnitude increases, but this theorem says although they thin out, they never completely disappear. What is important for us is that they do not thin out very fast: If we denote by $\pi(N)$ the number of primes less than $N$, then

$$\pi(N) \approx \frac{N}{\ln N},$$

where $\ln N$ denotes the natural logarithm of $N$, approximately 2.3 times the base 10 logarithm. This means that for 100-digit numbers, roughly 1 out of every $100 \ln 10 \approx 230$ is a prime.

2. *Testing whether a given integer is prime is easy.* You may believe that we just got through saying it's hard! But we merely gave a hard algorithm for the problem. Later on we'll see the makings of easy algorithms.

3. *There is no known easy algorithm for finding the prime factors of an integer...***and we think there never will be.** Much of the security of existing public-key cryptosystems rests on the unproven assumption that factoring is a hard *problem,* and which means there is no easy algorithm. An important qualification here is that this assumption is based on conventional models of computation; it has been proved that if it is possible to build *quantum computers,* then factoring is easy.

A consequence of our three fundamental facts is a recipe for creating a secret that is safe forever: Flip a coin about a thousand times, writing down 1 for heads and 0 for tails. Take the integer whose binary representation you wrote down, and test it for primality (fact 2). If it is composite, test the next higher integer, and then the next, until a prime is found. Fact 1 says that this will only require a few hundred tests. Repeat the experiment to find a second prime. Multiply the two numbers together. *You can publish the product, but no one can ever find the factors*.

# 3 Euclid's Algorithm

$\mathbf{Z}$ denotes the set of integers; $\mathbf{Z}^+$ the set of positive integers. $m|n$ means $m$ *divides* $n$, in other words $n = c \cdot m$ for some integer $c$.

Given $n \in \mathbf{Z}$, $d \in \mathbf{Z}^+$, there exist unique $q, r \in \mathbf{Z}$ with $0 \leq r < d$ such that $n = qd + r$. $q$ and $r$ are the quotient and remainder in ordinary integer division. We also write $r = n \bmod d$.

For example,

$$14 = 4 \cdot 3 + 2, -14 = -5 \cdot 3 + 1.$$

Thus

$$14 \bmod 3 = 2, -14 \bmod 3 = 1.$$

Once we have the remainder, we can divide $r$ into $d$ to obtain a still smaller remainder, and continue the process until we get zero as a remainder.

For example

$$\begin{aligned} 68 &= 3 \cdot 19 + 11 \\ 19 &= 1 \cdot 11 + 8 \\ 11 &= 1 \cdot 8 + 3 \\ 8 &= 2 \cdot 3 + 2 \\ 3 &= 1 \cdot 2 + 1 \\ 2 &= 2 \cdot 1 + 0. \end{aligned}$$

This procedure is called *Euclid's Algorithm.* The last nonzero remainder produced in the procedure is called the *greatest common divisor* ($\gcd$) of the numbers you started with. So in the example, we get $\gcd(19, 68) = 1$. The $\gcd$ has the following properties:

$$\gcd(a, b)|a, \gcd(a, b)|b,$$

and for any integer $c$,

$$(c|a \quad \text{and} \quad c|b) \Rightarrow c|\gcd(a, b).$$

Here is a Python implementation of Euclid's Algorithm:

```
def euclid(num1,num2):
    while num2 != 0:
        (num1,num2)=(num2,num1 % num2)
    return num1
```

Observe that in any division in the process we have

$$n = qd + r \geq 1 \cdot d + r > 2r.$$

Thus if $r_i, r_{i+1}, r_{i+2}$ are three successive remainders in Euclid's algorithm

$$r_i > 2r_{i+2}.$$

This means that the total number of steps to termination of the algorithm is no more that $2 \log_2 n$, where $n$ is the larger of the numbers you start with. Since each division requires no more than $(\log_2 n)^2$ steps, this means *Euclid's algorithm is easy.*

Every successive remainder produced in the course of Euclid's algorithm has the following property: It can be written in the form $an + bm$, where $n$ and $m$ are the numbers you start with. The reason is this: Obviously we have

$$n = 0 \cdot m + 1 \cdot n, m = 1 \cdot m + 0 \cdot n.$$

If at some step in Euclid's algorithm we have

$$r_i = qr_{i+1} + r_{i+2},$$

with

$$r_i = am + bn, r_{i+1} = a'm + b'n,$$

then

$$r_{i+2} = (a - qa')m + (b - qb')n.$$

If we keep track of these coefficients at each step, we get the result. Here is how we apply this in our example above, with $n = 68, m = 19$: The first pairs of coefficients are $(0, 1)$ and $(1, 0)$. The successive steps give

$$q = 3, a = 0 - 3 \times 1 = -3, b = 1 - 3 \times 0 = 1, 11 = -3 \times 19 + 1 \times 68.$$

$$q = 1, a = 1 - 1 \times -3 = 4, b = 0 - 1 \times 1 = -1, 8 = 4 \times 19 - 68.$$

$$q = 1, a = -3 - 1 \times 4 = -7, b = 1 - 1 \times -1 = 2, 3 = -7 \times 19 + 2 \times 68.$$

$$q = 2, a = 4 - 2 \times (-7) = 18, b = -1 - 2 \times 2 = -5, 2 = 18 \times 19 - 5 \times 68$$

$$q = 1, a = -7 - 1 \times 18 = -25, b = 2 - 1 \times (-5) = 7, 1 = -25 \times 19 + 7 \times 68.$$

This is called the *Extended Euclid Algoarithm*. This procedure for computing the coefficients $a$ and $b$ adds a couple of extra multiplications and additions of $\log n$-bit numbers at each step, so we still have an *easy algorithm.* A Python implementation is shown below.

```
def extended_euclid(num1,num2):
    #given values x,y return
    #(gcd(x,y),r,s) where r*x+s*y=gcd(x,y)
        (a,b,aa,bb)=(0,1,1,0)
        while num2 !=0:
            (q,r)=divmod(num1,num2)
            (a,b,aa,bb)=(aa-q*a,bb-q*b,a,b)
            (num1,num2)=(num2,r)
        return (num1,(aa,bb))
```

# 4 Unique Factorization

If $\gcd(m, n) = 1$, then $m$ and $n$ have no common factors (other than the trivial factor 1). We then have $am + bn = 1$ for some integers $a, b$.

Suppose $m, n \in \mathbf{Z}$ and $p$ is prime, and suppose $p|mn$. Then $p|m$ or $p|n$. To see this, suppose $p \not| m$. Then $\gcd(p, m) = 1$–this is where we use the fact that $p$ is prime. Thus $1 = ap + bm$ for some $a, b$. So

$$n = (ap + bm)n = apn + bmn.$$

Since $p$ divides $mn$, $apn + bmn$ is divisible by $p$, so $p|n$.

This means that if we try to write an integer as the product of primes in two different ways

$$n = p_1 \cdots p_r = q_1 \cdots q_s,$$

then the two lists $p_1, p_2, \ldots,$ and $q_1, q_2, \ldots,$ must contain exactly the same primes: because if, say, $q_1$ did not appear among the $p$'s, we would have $q_1|p_1 \cdots p_r$, without $q_1$ dividing any of the $p_i$, contradicting our observation above. For the same reason, the number of times each prime appears in the two lists must be the same. For example if 3 appears 4 times among the $p_i$ and at least 5 times among the $q_j$, then we would get two prime factorizations for $p/3^4$, one of which contains 3 and the other of which does not, contradicting what we just showed.

So the factorization of an integer into primes is *unique* up to a rearrangement of the factors.

# 5 Modular Exponentiation and Modular Inverses

We write

$$a \equiv b \pmod{n}$$

if $a$ and $b$ leave the same remainder upon division by $n$. This is equivalent to saying $n|a - b$. An important simple fact here is that if

$$a_1 \equiv b_1 \pmod{n}$$

and

$$a_2 \equiv b_2 \pmod{n},$$

then

$$a_1 a_2 \equiv b_1 b_2, a_1 + a_2 \equiv b_1 + b_2 \pmod{n}.$$

Now suppose we have the task of computing something like

$$56789^{42376} \bmod 111427.$$

The equations above tell us that at each multiplication, we can take the remainder modulo 111427, so we never have to deal with numbers larger than this. On the other hand, that still makes 42376 multiplications to perform, so the complexity of this algorithm is governed by the *magnitude* of the exponent, so this method of modular exponentiation is *hard.*

However, there is a different method, involving far fewer multiplications and divisions. With fewer than $\log_2(42376) \approx 16$ divisions, we can compute the binary expansion of $42376$, essentially writing this number as the sum of powers of 2:

$$42376 = 2^{15} + 2^{13} + 2^{10} + 2^8 + 2^7 + 2^3.$$

We compute $56789^{2^k} \bmod 111427$ by squaring and reducing $k$ times, so we get a table of all of these values up to $k = 15$ with no more than 15 multiplications and divisions, and then do 5 more multiplications and divisions to find the product of $56789^{2^k} \bmod 111427$ for $k = 15, 13, 10, 8, 7, 3$. The whole *repeated squaring* algorithm requires no more than $2 \log_2 N$ multiplications and divisions of numbers of length $\log_2 N$ to compute

$$a^b \mod c,$$

where $N = \max(a, b, c)$. So *modular exponentiation is easy.*

Suppose you want to compute $a^{-1} \bmod n$, in other words, to solve the equatin

$$ax \equiv 1 \pmod{n}.$$

If $a$ and $n$ have a prime factor $p$ in common, then this is impossible, since $n - ax$ will be divisible by $p$. However if $\gcd(a, n) = 1$, then we can write

$$1 = ax + ny$$

for some $x, y$, and thus solve the equation. Furthermore, we saw above how to do this with the extended Euclid algorithm, so *computing modular inverses is easy.*

**Python note.** Python has built-in exact arithmetic with arbitrarily large integers, and built-in modular exponentiation. The computation below, which seems to give the answer instantly, would be impossible without an easy algorithm:

```
>>> pow(123456789,876547617868168,16763781991)
7608365131L
```

Python will not let you compute the modular inverse, if it exists, by setting the second argument of `pow` to -1. To find $a^{-1} \bmod n$, you need to apply the Extended Euclid Algorithm to $(a, n)$ to find $x, y$ such that $ax + ny = 1$. You then have $a^{-1} \bmod n = x \bmod n$.

# 6 Fermat's Theorem and Primality Testing

Let us choose a prime $p$ and compute the powers $a^m \bmod p$ for $1 < a < p, 1 \leq m < p$, We'll let Python do this for us, in the case $p = 13$:

```
>>> table=[[pow(m,n,13) for n in range(1,13)] for m in range(2,13)]
>>> for row in table:
print row


[2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1]
[3, 9, 1, 3, 9, 1, 3, 9, 1, 3, 9, 1]
[4, 3, 12, 9, 10, 1, 4, 3, 12, 9, 10, 1]
[5, 12, 8, 1, 5, 12, 8, 1, 5, 12, 8, 1]
[6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1]
[7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1]
[8, 12, 5, 1, 8, 12, 5, 1, 8, 12, 5, 1]
[9, 3, 1, 9, 3, 1, 9, 3, 1, 9, 3, 1]
[10, 9, 12, 3, 4, 1, 10, 9, 12, 3, 4, 1]
[11, 4, 5, 3, 7, 12, 2, 9, 8, 10, 6, 1]
[12, 1, 12, 1, 12, 1, 12, 1, 12, 1, 12, 1]
```

We observe the following properties: The powers of $a$ always cycle: for instance at $a = 3$ we get tthe cycle $3, 9, 1$ repeatedly, for $a = 4$, we get $4, 3, 12, 9, 10, 1$ repeatedly, and for $a = 2, 6, 7, 11$ we get a single long cycle of length 12. In all cases, the length of the cycle is a divisor of 12. This behavior holds in general (this is not too hard to prove): Whenever $1 \leq a < p$ and $p$ is prime, then

$$a^{p-1} \equiv 1 \pmod{p}.$$

This is called 'Fermat's Little Theorem'. This gives us a way of proving that a number is composite without actually producing a factorization. For example, let $n = 1091645783$. It is tough to factor this, and trial division would require

tens of thousands of divisions. But thanks to Fermat's theorem and easy modular exponentiation, and few dozen multiplications and divisions yield

$$2^{n-1} \bmod n = 608256757.$$

So $n$ is composite. This method yields no information about the factors of $n$.

Let's provide just a little more detail about this kind of primality test: It never gives a false negative—that is, it never labels a prime as composite. There are two ways that it can give a false positive: It may be that $n$ is composite, and $a^{n-1} \equiv 1 \pmod{n}$ for some choices of $a$, but not for others. In this case, it can be shown that $a^{n-1} \not\equiv 1$ for at least half the choices of $a$ (and probably many more) so if $n$ passes this test for a few dozen sampled values of $a$, it is almost certain to be prime. This is a *probabilistic* algorithm: there is a slight probability of error, but repeated sampling reduces the probability of error to a negligible amount.

The other way that the test can yield false positives is if $n$ is composite but $a^{n-1} \equiv 1 \pmod{n}$ for *every* $1 < a < n$. There are such integers $n$. They are called *Carmichael numbers*, and the Fermat test answers incorrectly for them, no matter how many values of $a$ are sampled. There are infinitely many Carmichael numbers, but they are extremely sparse, so that the chance of hitting one in a random search for primes $\approx 10^{100}$ is negligibly small. The problem can be eliminated entirely—a refinement of the Fermat test, called the *Miller-Rabin test*—also rejects Carmichael numbers with high probability.

The probabilistic feature of these algorithms can also be eliminated: In 2002, the Agrawal-Kayal-Saxena algortihm (AKS), an efficient algorithm that tests for primality with no possibility of error, was published. (Probabilistic tests remain more efficient, however.)

# 7 Chinese Remainder Theorem

Let us imagine that we have a pair of counters, one that counts modulo 9, (*i.e.* 0,1,2,3,4,5,6,7,8,0,1...) and the other modulo 5. We start both counters at 0, and at the press of a button, both counters advance 1. What configurations of the pair of counters will we see as we continue to press the button? Here is the result:

```
>>> [(i%9,i%5) for i in range(45)]
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4),
 (5, 0), (6, 1), (7, 2), (8, 3), (0, 4),
 (1, 0), (2, 1), (3, 2), (4, 3), (5, 4),
```

```
 (6, 0), (7, 1), (8, 2), (0, 3), (1, 4),
 (2, 0), (3, 1), (4, 2), (5, 3), (6, 4),
 (7, 0), (8, 1), (0, 2), (1, 3), (2, 4),
 (3, 0), (4, 1), (5, 2), (6, 3), (7, 4),
 (8, 0), (0, 1), (1, 2), (2, 3), (3, 4),
 (4, 0), (5, 1), (6, 2), (7, 3), (8, 4)]
```

The pair of counters cycles through all 45 possible configurations before returning to 0.

On the other hand, if we had used the moduli 4 and 6, we would get

```
  >>> [(i%4,i%6) for i in range(24)]
[(0, 0), (1, 1), (2, 2), (3, 3),
 (0, 4), (1, 5), (2, 0), (3, 1),
 (0, 2), (1, 3), (2, 4), (3, 5),
 (0, 0), (1, 1), (2, 2), (3, 3),
 (0, 4), (1, 5), (2, 0), (3, 1),
 (0, 2), (1, 3), (2, 4), (3, 5)]
```

The difference between the two counters is always even: Thus there are two complete cycles, with all the pairs in which the difference is divisible by 2 appearing twice, and the other twelve pairs (for instance, $(2, 5)$) not appearing at all.

The difference between the two situations occurs, as you might expect, because 9 and 5 are relatively prime, whereas 4 and 6 have the common factor 3. To see why we get the complete cycle of length 45 in the first case, without having to tabulate the entire list, observe that if a pair $(j, k)$ appears twice, then we have $i < i'$ such that

$$i \equiv i' \equiv j \pmod 9,$$
$$i \equiv i' \equiv k \pmod 5.$$

Thus $9|(i - i')$ and $5|(i - i')$. Because of unique factorization, all of the factors of 9 and all of the factors of 5 must appear in the prime factorization of $i - i'$, so $45|(i - i')$ This means that there can be no repeats of a pair among

$$(i \bmod 9, i \bmod 5),$$

for $0 \leq i < 45$. Since no pair can appear twice among these 45 pairs, every pair must occur exactly once.

The identical argument works for every pair of relatively prime moduli. Here is the general principle: If $m, n$ are relatively prime, and $0 \leq j < m$, $0 \leq k < n$, then there is a unique $0 \leq x < mn$ such that

$$x \equiv j \pmod{m},$$

$$x \equiv k \pmod{n}.$$

This fact is called the *Chinese Remainder Theorem.* Here is another way to see that the Chinese Remainder Theorem is true, as well as a way to compute the solution $x$ to such a pair of equations. Let $K = mn$. Because $m$ and $n$ are relatively prime, $n$ has a multiplicative inverse modulo $m$, and vice-versa. We then pick

$$x = (j \cdot n \cdot (n^{-1} \bmod m) + i \cdot m \cdot (m^{-1} \bmod n)) \bmod K.$$

To see that this works, observe that if we reduce modulo $m$, then the second term becomes 0 and the first term is congruent to $j \cdot 1 = j$ modulo $m$. Likewise, the second term is congruent to $k$ modulo $n$.

Here's an example, again with the moduli 9 and 5 that we used before. Suppose we want to solve

$$x \equiv 7 \pmod{9}$$

$$x \equiv 3 \pmod{5}.$$

We first need $5^{-1} \bmod 9$ and $9^{-1} \bmod 5$. If we had big numbers we would do this efficiently with the extended Euclid algorithm, but here the numbers are so small that we can figure it quickly with very little calculation:

$$5^{-1} \bmod 9 = 2, 9^{-1} \bmod 5 = 4.$$

We thus get

$$\begin{aligned} x &= (7 \times 5 \times 2 + 3 \times 9 \times 4) \bmod 45 \\ &= (70 + 108) \bmod 45 \\ &= (25 + 18) \\ &= 43. \end{aligned}$$

# 8 RSA

Here we present the RSA public-key cryptosystem as a number-theoretic algorithm; we'll talk about the cryptographic significance later. There are three separate algorithms. It is important to note every step along the way that each of these algorithms is *easy.*

## 8.1 The algorithms

### 8.1.1 Key generation

1. Bob generates two distinct large random prime numbers $p$ and $q$: he can pick random target values and test nearby numbers until he finds a prime. Because of easy primality testing and the density of primes, this is easy.

2. Bob computes $N = pq$ and $K = (p-1)(q-1)$. This is easy because multiplication is easy.

3. Bob searches for a (typically small) value $e$ such that $\gcd(e, K) = 1$. This is easy because $K < 10^{1000}$ cannot have a large number of distinct prime factors, so there will be some small prime that does not divide $K$. He publishes $(e, N)$ as his *public key.*

4. Bob computes $d = e^{-1} \bmod K$. This is easy with the extended Euclid algorithm. He keeps $(d, N)$ as his *private key.*

### 8.1.2 Encryption

Alice's plaintext message must be a positive integer $M < N$. She sends Bob the ciphertext

$$C = M^e \bmod N.$$

This is easy with the repeated squaring algorithm.

### 8.1.3 Decryption

Bob computes

$$C^d \bmod N$$

to recover the plaintext.

### 8.1.4 Proof of Correctness

$$C \equiv M^e \bmod N,$$

so

$$\begin{aligned} C^d &\equiv M^{ed} \\ &\equiv M^{t(p-1)(q-1)+1} \quad \text{for some } t \text{ since } d = e^{-1} \bmod (p-1)(q-1) \\ &\equiv (M^{t(q-1)})^{p-1} M \\ &\equiv M \pmod{p}. \end{aligned}$$

The last equality follows by Fermat's theorem if $p$ does not divide $M$. If $p$ does divide $M$ (which is extremely unlikely for big numbers and a random message $M$) then it follows because $M \equiv 0 \pmod{p}$.

By the identical argument

$$C^d \equiv M \pmod{q}.$$

By the Chinese Remainder Theorem there is exactly one solution to

$$x \equiv M \pmod{p}$$

$$x \equiv M \pmod{q}$$

that is less than $N$, so the only such solution must be $M$ itself. Therefore

$$C^d \equiv M \pmod{N},$$

so the decryption algorithm recovers the plaintext.

## 8.2 Worked Example

We work with artificially small parameters. Bob begins by generating $p = 79$, $q = 109$. We then get

$$N = 79 \times 109 = 8611$$

$$K = 78 \times 108 = 8424.$$

Note that $3|K$, so we cannot choose $e = 3$. However, $e = 5$ works.Observe also $5 \times 1685 = 8424 + 1$, so $e^{-1} \bmod K = 1685$. We thus have public key $(5, 8611)$ and private key $(1685, 8611)$.

Let's choose plaintext $M = 4000$. The following calculations show encryption, and then decryption.

```
>>> pow(4000,5,8611)
8469
>>> pow(8469,1685,8611)
4000
```