CSCI 270 Reading Supplement: Number Theory & Cryptography

1 Number Theory

1.1 Modular Exponentiation

Suppose we want to calculate 3^{644} **mod** 645. One way we might imagine doing this is to compute 3^{644}, divide by 645, and keep the remainder. It turns out there are better ways.

To do this, I am going to compute 3^{2^x} for all powers of two $0 \leq x \leq \lfloor \log 644 \rfloor$. In each case, I need only keep the remainder when divided by 645. Furthermore, I use the previous row in the table and square its result to get the next one. For example, when computing 3^8, I start with 3^4, square the result, and keep only the result after dividing by 645 : in this case, $3^8 \equiv 111 \pmod{645}$. Then, when computing 3^{16}, I start with 111 (the result from 3^8) and square that.

x	3^x **mod** 645
1	3
2	$3^2 = 9$
4	$9^2 = 81$
8	$81^2 = 6561 = 10 \cdot 645 + 111$
16	$111^2 = 12321 = 19 \cdot 645 + 66$
32	$66^2 = 4356 = 6 \cdot 645 + 486$
64	$486^2 = 236196 = 366 \cdot 645 + 126$
128	$126^2 = 15876 = 24 \cdot 645 + 396$
256	$396^2 = 156816 \to 81$

Before proceeding, ask yourself what the value of 3^{512} is going to be in this modulus. Note that it is going to be the result of squaring 81 and taking the remainder mod 645.

If you answered 111, you're right. You shouldn't have needed to do a computation to do this: $3^4 \equiv 81$ also, and after we went through the computation for that, we found $3^8 \equiv 111$.

How do we use this to compute 3^{644}? You might notice that 644 isn't a power of two. Fortunately, like all positive integers, it can be represented as the sum of powers of two: in this case, $644 = 512 + 128 + 4$. Therefore, $3^{644} = 3^{512} \cdot 3^{128} \cdot 3^4$.

```
ModularExponentiation($b, n = (a_{k-1}a_{k-2}\ldots a_1 a_0), m$)
Input: $b$, the base ; $n$, the exponent written in binary; $m$, the modulus.
Output: The remainder when $b^n$ is divided by $m$.
  $x \leftarrow 1$
  power $= b$ % $m$
  for $i = 0 \to k-1$ do
    if $a_i = 1$ then
      $x \leftarrow (x \cdot$ power $)$ % $m$
    power = power $\cdot$ power
  return $x$
```

How long does it take to compute modular exponentiation? We create $\mathcal{O}(\log n)$ rows of the table, each of which take a constant amount of computation. We then combine $\mathcal{O}(\log n)$ rows to form the result; each combination takes a constant amount of computation as well. These constants assume that b and m are of size $\mathcal{O}(1)$; they can be re-evaluated if that is not true. For example, in the RSA algorithm, the modulus m will be a few hundred bits long (much larger than a C++ **long**).

1.2 Euclid's Algorithm

Sometimes you want to calculate the gcd (greatest common divisor) of two numbers. In sixth grade, we did this by factoring both numbers and multiplying the common factors.

But finding the prime factorization is too slow for large numbers. For that, we have the **Euclidean Algorithm**.

It's based on the idea in the following **example**: Suppose we want to calculate $\gcd(91, 287)$.

Any common divisor of 91 and 287 must also divide $287 - 91$ and $287 - 2 \cdot 91$ and so on. So we observe that $287 = 91 \cdot 3 + 14$ ... and thus, any common divisor of 287 and 91 is also shared by 14.

That is, $\gcd(91, 287) = \gcd(91, 14)$.

The same thought process gets me to $91 = 14 \cdot 6 + 7$, so $\gcd(91, 14) = \gcd(14, 7)$.

And I can solve that easily; $14 = 7 \cdot 2$ so $7 = \gcd(14, 7) = \gcd(91, 14) = \gcd(91, 287)$.

More mathematically, this procedure says that if $a = bq + r$ (for integers a, b, q, r), then $\gcd(a, b) = \gcd(b, r)$. And if you'd like directions to follow, here's an algorithm:

```
EuclideanAlgorithm (a,b:  positive integers)
  $x \leftarrow a$
  $y \leftarrow b$
  while $y \neq 0$ do
    $r \leftarrow x\%y$
    $x \leftarrow y$
    $y \leftarrow r$
  return $x$
```

I know many students aren't yet in love with recursion, but the recursive formulation[1] is much more clear: take the gcd of the smaller of a, b with the remainder when the larger is divided by the smaller.

```
EuclideanAlgorithmRecursive (a,b:  positive integers)
  if $a < b$ then
    return EuclideanAlgorithmRecursive(b,a)
  return EuclideanAlgorithmRecursive(b, a % b)
```

Question 1. What is $\gcd(414, 662)$?

1.3 Finding Primes

1.3.1 The Grade Six Algorithm and the Sieve

How did you find prime numbers in sixth grade? One mechanism we can use is by checking the possible prime factors of a number. Observe that if n is a composite integer, it has a prime divisor less than or equal to $\sqrt{n}$.

Question 2. Is 101 prime? Prove your answer.

To solve this, we can check every prime number up to $\sqrt{101} \approx 10.1$ and confirm that none divide 101. Note

[1]That's all recursion is: how to solve a big problem where lots of the work is just solving a smaller version of the same problem. Obligatory "see also, recursion."

you should write each and that it doesn't divide 101. Writing "101 is prime because it is only divisible by itself and 1" is a circular argument.

We can use a more algorithmic approach; suppose you wanted to find all primes not exceeding some number. There's a method for this, known as the **Sieve of Eratosthenes**.

To find all primes not exceeding x, note that any such prime must have a prime divisor $\leq \sqrt{x}$. List out the numbers $2 \ldots x$. Remove all numbers a multiple of the first number, so we have all odd numbers between 3 and x. Do it again, so all multiples of 3 are removed. Then all multiples of 5 are removed, followed by 7. Repeat until the smallest number is larger than $\sqrt{x}$.

For example, let's find all primes less than or equal to 49, using the sieve:

	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49

How long does this approach take? To do that, we need to apply the sieve to find all primes whose value is at most $\sqrt{n}$. This takes $\mathcal{O}(n)$ because applying the sieve up to size x takes $\mathcal{O}(x^2)$[2]. Plugging in $x = \sqrt{n}$ yields our running time.

$\mathcal{O}(n)$ seems efficient, but keep in mind this means it is linear *in the value of* n not the number of bits needed to represent it. A simple C++ unsigned int can have value up to around 2^{32}, despite being only 32 bits in length. In CSCI 170 and 104L, we told you polynomial really meant "polynomial in the size of the input," but we may not have clarified much beyond that. Later in this section, we will see algorithms whose running time is proportional to *the number of bits needed to represent a number*: so, for a 30-bit number, it's something times 30 instead of something times 2^{30}: quite the difference!

1.3.2 Fermat's Little Theorem

Fermat's Little Theorem states that if p is prime and a is an integer not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$; furthermore, for every integer a we have $a^p \equiv a \pmod{p}$.

For example, $6^{22} \equiv 1 \pmod{23}$ – note that 23 is a prime number. Similarly, $73^{100} \equiv 1 \pmod{101}$.

Question 3. $2^{1234566} \equiv 899557 \pmod{1234567}$. Is 1234567 a prime number?

Question 4. Compute $2^{35} \bmod 7$. Remember that 7 is a prime number.

Question 5. For all a such that $1 \leq a \leq 560$, $a^{560} \equiv 1 \pmod{561}$. Is 561 prime?

Unfortunately, it is not. While Fermat's Little Theorem can be used to definitively show that a number *is not* prime, it cannot be used to show that a number *is prime.*

For any number n, if n is prime and a meets the criteria above, then $a^{n-1} \equiv 1 \pmod{n}$. Therefore, if a meets the criteria but $a^{n-1} \not\equiv 1 \pmod{n}$, then we know that n is not prime. However, this is not an if and only if, and some numbers (such as 561) can "pretend" to be prime. This means that there is some non-zero probability that we get a wrong answer when testing if a number is prime. So we need a better test.

[2]For each of the x values we will cross off $\mathcal{O}$(x) squares

As for efficiency: note that for any choice of a, this takes $\mathcal{O}(\log n)$ iterations to determine if a number is either certainly composite or possibly prime. This is much faster than the grade six algorithm! It just doesn't find it for certain...

1.3.3 Rabin-Miller Test

To extend this, I am going to add one more fact about prime numbers to your growing education. I am not going to prove this fact here.

Fact: If n is prime, then the only solutions (mod n) to $x^2 \equiv 1 \pmod{n}$ are 1 and $n-1$. That is, the only square-roots of 1 (mod n) are 1 and -1 ($n-1$ is essentially -1 mod n).

Note that this *is not* true for many composites. In fact, if n is neither prime nor a power of a prime, then there are nontrivial square roots of 1 mod n.

For example, 15 is composite and solutions for $x^2 \equiv 1 \pmod{15}$ are $x \equiv 1, 4, 14$. The first and last are 1 and $n-1$, while 4 is neither. I leave it to the reader to verify that $4^2 \equiv 1 \pmod{15}$. Note that this means that, mod 15, $\sqrt{1} = 1, 4, 11, 15$.

With that in mind, we now face how to use this knowledge. We could, when testing if a number n is prime, compute $x^2 \pmod{n}$ for many values of x and see if we happen to find a square root of 1 along the way. But this adds significant computation. Instead, I am going to compute several along the way.

Suppose I want to test if some value p is prime or not. I can use Fermat's Little Theorem and an appropriate value of a and compute $a^{p-1} \pmod{p}$. I am instead going to do this in a slightly different fashion than the direct modular exponentiation algorithm from earlier.

If p is even and it isn't 2, I immediately know it isn't prime. Once I have conducted that test, let's consider any odd whole number p. Because $p-1$ is even, I can divide it by two at least once, possibly more times, until I end up with an odd value. So I can write $p-1 = 2^k q$, for some odd whole number q and positive integer k (which corresponds to how many times I divided by two until ending up with an odd number).

I can rewrite $a^{p-1} \pmod{p}$ as $a^{2^k q} \pmod{p}$. To compute this, I find $a^q \pmod{p}$ and then square the result k times. This takes $\mathcal{O}(k \log q)$ operations because we square the previous value k times, and for each value we reach, the modular exponentiation algorithm takes $\mathcal{O}(\log q)$. Note that k is at most the number of bits needed to represent the value p ($\mathcal{O}(\log p)$), and q has value at most p. Therefore, this is $\mathcal{O}(\log p)$ operations: a polynomial function of the *size of the input* (as opposed to the value of the input). Much faster than the grade six algorithm.

Now we have computed $a^{p-1} \pmod{p}$ by computing $a^q \pmod{p}$ and then squaring the result k times. If the result of this operation isn't 1, we know that p is composite (by Fermat's Little Theorem) and stop. If the result is 1, then p might be prime. But also, even if this was the case for all $a \in [1, n-1]$, p might a Carmichael number that is merely "pretending" to be prime.

The Rabin-Miller test allows us to check another indicator for p being composite. As we evaluate a^{p-1} by starting with $a^q \pmod{p}$ and squaring it repeatedly, we can check the value of $a^{2^i q} \pmod{p}$ for all $i \in [0, k]$. Remember that $a^{2^k q} \pmod{p} \equiv 1$. If $a^q \pmod{p} \not\equiv 1$, then at some point there is a crossover where $a^{2^i q}$ $\pmod{p} \equiv 1$ but $a^{2^{i-1} q} \pmod{p} \not\equiv 1$. If $a^{2^{i-1} q} \pmod{p} \not\equiv -1$, then we've found a value $x = a^{2^{i-1} q}$ such that $x^2 = a^{2^i q} \equiv 1 \pmod{n}$ but $x \pmod{n}$ is not 1 or -1. This violates the fact about primes at the beginning of this section, so we know that p is composite.

If I give you a value of a such that a potential prime number p does not reveal itself as composite with the Rabin-Miller test, you might wonder how likely it is that p is in fact prime. It turns out that there are no “Carmichael-type numbers” for the Rabin-Miller Test; if p is composite, at least 75% of values[3] for a between 1 and $p-1$ will cause p to reveal itself as composite. So, if we test one value of a and it doesn't reveal that p is a composite, the probability of p being composite is 0.25. If we test m values of a, and none of them reveal p as composite, then the probability of it being composite is 0.25^m. Very quickly, we can be fairly confident that p is prime, and we can always increase our confidence by testing more values of a.[4]

As an example of a Carmichael number that reveals itself through the Rabin-Miller test, let's examine the Carmichael number $p = 561$. We'll run the Rabin-Miller test with $a = 2$. We can write $p-1 = 560$ as $2^4 \cdot 35$.

- $2^{35} \equiv 263 \pmod{561}$
- $2^{2\cdot 35} \equiv 263^2 \equiv 166 \pmod{561}$
- $2^{4\cdot 35} \equiv 166^2 \equiv 67 \pmod{561}$
- $2^{8\cdot 35} \equiv 67^2 \equiv 1 \pmod{561}$

Note that setting $x = 2^{4\cdot 35}$, $x^2 \equiv 1 (\text{mod } 561)$ and x is something other than 1 or -1 (mod p). The Rabin-Miller test has revealed that p is a composite.

1.4 Euler's Totient Function

The function $\Phi(n)$ is defined as the number of positive integers less than or equal to n that are *relative prime* to n. For example, $\Phi(10) = 4$ because 10 is relatively prime to $1, 3, 7, 9$.

Question 6. What is $\Phi(6)?\Phi(7)?\Phi(5)?$

For $\Phi(6)$, we can see that $2, 3, 4$ share a common factor with 6, so $\Phi(6) = 2 = |\{1, 5\}|$.

5 and 7 share no common factors with any smaller positive integers, so their Φ values are 4 and 6 respectively.

Now that we understand what the function is, let's explore some properties of the function.

Question 7. Let p be any prime number. What is $\Phi(p)$?

$\Phi(p) = p - 1$: for all integers $1 \ldots p-1$, we know that p shares no common factors with it.

Question 8. Let p be any prime number and k any positive integer. What is $\Phi(p^k)$?

Imagine writing all the positive integers up to p^k in many rows of p values each. On the first row, we write $1, 2, \ldots p$, the second row gets $p+1, p+2, \ldots 2p$, etc. This is a total of p^{k-1} rows. Now we cross off multiples of p, as these are the only numbers that share a common factor with p^k. This crosses off a total of p^{k-1} numbers. The size of what is left is $p^k - p^{k-1}$, and that's the value of $\Phi(p^k)$.

A related interesting property of this function is that if $\gcd(m, n) = 1$, then $\Phi(m \cdot n) = \Phi(m) \cdot \Phi(n)$.

Question 9. What is $\Phi(42)$?

[3] We haven't given you any proof of this fact in CSCI 270. This goes well into number theory to get the answer. A proof that at least 50% of values of a will work for Carmichael numbers is available in [CLRS], §31.8, and only one test is obviously needed for non-Carmichael composites.

[4] You might be wondering why we don't just test at least 75% of the values of a between 1 and $p-1$ so we can be sure that p is prime. This would cause the algorithm to be $\mathcal{O}(p \log p)$ which is worse than our sieve algorithm.

Note that 6 and 7 are relatively prime. Therefore, $\Phi(42) = \Phi(6) \cdot \Phi(7) = 2 \cdot 6 = 12$. You can verify this yourself if you would like, such as by checking how many values $1, 2, \dots 41$ are divisible by 2, 3, or 7.

2 Public-Key Cryptography

Instead of relying on multiple people keeping a secret, public-key cryptography relies on one person keeping a secret. If I want people to be able to send me messages using RSA, I post two values e and n (and tell people that these are my RSA public key). In this section, we will start by showing how to send a secret message to someone given their public key and then how to decode it once you receive such a message encoded with your public key.

Then we will show how to select keys so that (a) people can easily send you messages and (b) only you can read such messages.

2.1 RSA Encryption

RSA encryption is used to send secure messages. If I want to send a message, I need to know the recipient's public key. Given their public key e, n and a message I want to send M, I plug these values into the following equation (modular exponentiation will be useful for this[5]) and send the encrypted message C.

- $C = M^e \bmod n$

2.2 RSA Decryption

RSA decryption is used to read secure messages. If I want to read a message, I need to know my private key. Given my private key d, n and an encrypted message C, I plug these values into the following equation (again, modular exponentiation will be useful) and extract the decrypted message M.

- $M = C^d \bmod n$

2.3 Generating Keys

If I want to set up this system to receive messages securely, I need to know how to generate public and private keys. I need to pick values of d, e, and n so that my private key will properly decrypt messages sent by the public keys. Since I know I want $C = M^e \bmod n$ and $M = C^d \bmod n$, I need d and e to be inverses of one another (mod n). This means that for any x, $x^{de} \equiv x \pmod{n}$. The following steps show why this is a requirement:

- $M = C^d \bmod n$ and $C = M^e \bmod n$
- $M = (M^e \bmod n)^d \bmod n$
    - ∗ remember from the modular exponentiation section that $b^x \bmod m = (b \bmod m)^x \bmod m$
- $M = M^{ed} \bmod n$

[5]see https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation for a very clear example of modular exponentiation

- equivalently, $M^{de} \equiv M \pmod{n}$
- this needs to be true for any message $M = x$

It turns out that if $ed \equiv 1 \pmod{\Phi(n)}$, then for any x, $x^{de} \equiv x \pmod{n}$ and d and e are inverses (mod n).

When selecting n, I want a value that is difficult to factor. Otherwise, knowing n, it would be easy to solve for $\Phi(n)$ because $\Phi(x \cdot y \cdot z) = \Phi(x) \cdot \Phi(y) \cdot \Phi(z)$. Someone could intercept an encrypted message and, knowing only the public key e, n, use Euclid's algorithm to find d. If they have d, then they can decrypt the message. Here is how they could find d:

- they know $ed \equiv 1 \pmod{\Phi}$, ($\Phi(n)$ is shortened to Φ here)
- $ed + v\Phi = 1$, for some v (by the definition of mod)
- they can find values of d and v that satisfy this by using Euclid's algorithm to find gcd(e, Φ). To show how this works, let's use $e = 49, \Phi = 38$.
  - $(e) = 1 \cdot (\Phi) + 11 \quad \rightarrow \quad 11 = e - 1 \cdot \Phi$
  - $(\Phi) = 3 \cdot (e - 1 \cdot \Phi) + 5 \quad \rightarrow \quad 5 = \Phi - 3 \cdot (e - 1 \cdot \Phi) = -3 \cdot e + 4 \cdot \Phi$
  - $(e - 1 \cdot \Phi) = 2 \cdot (-3 \cdot e + 4 \cdot \Phi) + 1 \quad \rightarrow \quad 1 = e - 1 \cdot \Phi - 2 \cdot (-3 \cdot e + 4 \cdot \Phi) = 7 \cdot e - 9 \cdot \Phi$
  - $d = 7, v = -9$ solves this example. We only care about the value for d.

If I make n the product of two large prime numbers (which I know how to do using the Rabin-Miller algorithm), factoring it will be very difficult. So, $n = p * q$ where p and q are large primes.

- $\Phi(n) = \Phi(p \cdot q)$
- $\Phi(n) = \Phi(p) \cdot \Phi(q)$ (because p and q are relatively prime)
- $\Phi(n) = (p - 1)(q - 1)$ (because p and q are prime)

I select a value for e and choose values for p and q such that e is relatively prime to $\Phi(n) = (p - 1)(q - 1)$. Now I can use Euclid's Algorithm to find a value for d so that $ed \equiv 1 \pmod{\Phi(n)}$ (see the approach above to crack my message knowing $\Phi(n)$).

3 Additional References

This chapter of CSCI 270 is a combination of algorithms as a matter of theory, algorithms as a matter of practice, and number theory. In addition to the reading sections of your preferred CSCI 270 textbook, students who find this particularly interesting can learn more from the following sources, from which I borrowed a few examples for this document. For readability purposes they aren't listed explicitly above; this is not permission to do the same for your Writing 340 paper.

1. *A Friendly Introduction to Number Theory* by Joseph Silverman. This textbook was written to be a Number Theory textbook for a math class offered as a "general ed" class for non-STEM students. Chapters are short and clear; students interested in the math behind this section of CSCI 270 are encouraged to read this.
2. *Network Security: Private Communication in a Public World* by Charlie Kaufman, Radia Perlman, and Mike Speciner. This book covers a lot of cryptography (among other topics) as they are used for computer and network security purposes.
3. CSCI 476, offered most Fall semesters, is a fun tech elective choice for students who want to know more about cryptography.