2.4 Details of the RSA Cryptosystem

In this section, we deal with some issues related to implementing the RSA cryptosystem: exponentiating large numbers, finding primes, and factoring.

Practical Aspects of Exponentiation mod n

Suppose you are going to raise a 100 digit number a to the 10^{120}th power modulo a 200 digit integer n. Note that the exponent is a 121 digit number.

Exercise 2.4-1 Propose an algorithm to compute $a^{10^{120}} \bmod n$, where a is a 100 digit number and n is a 200 digit number.

Exercise 2.4-2 What can we say about how long this algorithm would take on a computer that can do one infinite precision arithmetic operation in constant time?

Exercise 2.4-3 What can we say about how long this would take on a computer that can multiply integers in time proportional to the product of the number of digits in the two numbers, i.e. multiplying an x-digit number by a y-digit number takes roughly xy time?

Notice that if we form the sequence a, a^2, a^3, a^4, a^5, a^6, a^7, a^8, a^9, a^{10}, a^11 we are modeling the process of forming a^{11} by successively multiplying by a. If, on the other hand, we form the sequence a, a^2, a^4, a^8, a^{16}, a^{32}, a^{64}, a^{128}, a^{256}, a^{512}, a^{1024}, we are modeling the process of successive squaring, and in the same number of multiplications we are able to get a raised to a four digit number. Each time we square we double the exponent, so every ten steps or so we will add three to the number of digits of the exponent. Thus in a bit under 400 multiplications, we will get $a^{10^{120}}$. This suggests that our algorithm should be to square a some number of times until the result is almost $a^{10^{120}}$, and then multiply by some smaller powers of a until we get exactly what we want. More precisely, we square a and continue squaring the result until we get the largest $a^{2^{k_1}}$ such that 2^{k_1} is less than 10^{120}, then multiply $a^{2^{k_1}}$ by the largest $a^{2^{k_2}}$ such that $2^{k_1} + 2^{k_2}$ is less than 10^{120} and so on until we have

$$10^{120} = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_r}$$

for some integer r. (Can you connect this with the binary representation of 10^{120}?) Then we get

$$a^{10^{120}} = a^{2^{k_1}} a^{2^{k_2}} \cdots a^{2^{k_r}}.$$

Notice that all these powers of a have been computed in the process of discovering k_1. Thus it makes sense to save them as you compute them.

To be more concrete, let's see how to compute a^{43}. We may write $43 = 32 + 8 + 2 + 1$, and thus

$$a^{43} = a^{2^5} a^{2^3} a^{2^1} a^{2^0} \ . \tag{2.21}$$

So, we first compute $a^{2^0}, a^{2^1}, a^{2^2}, a^{2^3} a^{2^4}, a^{2^5}$, using 5 multiplications. Then we can compute a^{43}, via equation 2.21, using 3 additional multiplications. This saves a large number of multiplications.

On a machine that could do infinite precision arithmetic in constant time, we would need about $\log_2(10^{120})$ steps to compute all the powers a^{2^i}, and perhaps equally many steps to do the multiplications of the appropriate powers. At the end we could take the result mod n. Thus the length of time it would take to do these computations would be more or less $2\log_2(10^{120}) = 240\log_2 10$ times the time needed to do one operation. (Since $\log_2 10$ is about 3.33, it will take at most 800 times the amount of time for one operation to compute $a^{10^{120}}$.)

You may not be used to thinking about how large the numbers get when you are doing computation. Computers hold fairly large numbers (4-byte integers in the range roughly -2^{31} to 2^{31} are typical), and this suffices for most purposes. Because of the way computer hardware works, as long as numbers fit into one 4-byte integer, the time to do simple arithmetic operations doesn't depend on the value of the numbers involved. (A standard way to say this is to say that the time to do a simple arithmetic operation is constant.) However, when we talk about numbers that are much larger than 2^{31}, we have to take special care to implement our arithmetic operations correctly, and also we have to be aware that operations are slower.

Since $2^{10} = 1024$, we have that 2^{31} is twice as big as $2^{30} = (2^{10})^3 = (1024)^3$ and so is somewhat more than two billion, or $2 \cdot 10^9$. In particular, it is less than 10^{10}. Since 10^{120} is a one followed by 120 zeros, raising a positive integer other than one to the 10^{120}th power takes us completely out of the realm of the numbers that we are used to making exact computations with. For example, $10^{(10^{120})}$ has 119 more zeros following the 1 in the exponent than does 10^{10}.

It is accurate to assume that when multiplying large numbers, the time it takes is roughly proportional to the product of the number of digits in each. If we computed our 100 digit number to the 10^{120}th power, we would be computing a number with more than 10^{120} digits. We clearly do **not** want to be doing computation on such numbers, as our computer cannot even store such a number!

Fortunately, since the number we are computing will ultimately be taken modulo some 200 digit number, we can make all our computations modulo that number. (See Lemma 2.3.) By doing so, we ensure that the two numbers we are multiplying together have at most 200 digits, and so the time needed for the problem proposed in Exercise 2.4-1 would be a proportionality constant times 40,000 times $\log_2(10^{120})$ times the time needed for a basic operation plus the time needed to figure out which powers of a are multiplied together, which would be quite small in comparison.

Note that this algorithm, on 200 digit numbers, is as much as $40,000$ times slower than on simple integers. This is a noticeable effect and if you use or write an encryption program, you can see this effect when you run it. However, we can still typically do this calculation in less than a second, a small price to pay for secure communication.

How long does it take to use the RSA Algorithm?

Encoding and decoding messages according to the RSA algorithm requires many calculations. How long will all this arithmetic take? Let's assume for now that Bob has already chosen p, q, e, and d, and so he knows n as well. When Alice wants to send Bob the message x, she sends $x^e \bmod n$. By our analyses in Exercise 2.4-2 and Exercise 2.4-3 we see that this amount of time is more or less proportional to $\log_2 e$, which is itself proportional to the number of digits of e, though the first constant of proportionality depends on the method our computer uses to multiply numbers. Since e has no more than 200 digits, this should not be too time consuming for Alice if she has a reasonable computer. (On the other hand, if she wants to send a message consisting of many segments of 200 digits each, she might want to use the RSA system to send a key for another simpler (secret key) system, and then use that simpler system for the message.)

It takes Bob a similar amount of time to decode, as he has to take the message to the dth power, mod n.

We commented already that nobody knows a fast way to find x from $x^e \bmod n$. In fact, nobody knows that there isn't a fast way either, which means that it is possible that the RSA cryptosystem could be broken some time in the future. We also don't know whether extracting eth roots mod n is in the class of NP-complete problems, an important family of problems with the property that a reasonably fast solution of any one of them will lead to a reasonably fast solution of any of them. We do know that extracting eth roots is no harder than these problems, but it may be easier.

However, here someone is not restricted to extracting roots to discover x. Someone who knows n and knows that Bob is using the RSA system, could presumably factor n, discover p and q, use the extended GCD algorithm to compute d and then decode all of Bob's messages. However, nobody knows how to factor integers quickly either. Again, we don't know if factoring is NP-complete, but we do know that it is no harder than the NP-complete problems. Thus here is a second possible way around the RSA system. However, enough people have worked on the factoring problem, that most people are confident that it is in fact difficult, in which case the RSA system is safe, as long as we use keys that are long enough.

How hard is factoring?

Exercise 2.4-4 Factor 225,413. (The idea is to try to do this without resorting to computers, but if you give up by hand and calculator, using a computer is fine.)

With current technology, keys with roughly 100 digits are not that hard to crack. In other words, people can factor numbers that are roughly 100 digits long, using methods that are a little more sophisticated than the obvious approach of trying all possible divisors. However, when the numbers get long, say over 120 digits, they become very hard to factor. The record, as of the year 2000, for factoring is a roughly 155-digit number. To factor this number, thousands of computers around the world were used, and it took several months. So given the current technology, RSA with a 200 digit key seems to be very secure.

Finding large primes

There is one more issue to consider in implementing the RSA system for Bob. We said that Bob chooses two primes of about a hundred digits each. But how does he choose them? It follows from some celebrated work on the density of prime numbers that if we were to choose a number m at random, and check about $\log_e(m)$ numbers around m for primality, we would expect that one of these numbers was prime. Thus if we have a reasonably quick way to check whether a number is prime, we shouldn't have to guess too many numbers, even with a hundred or so digits, before we find one we can show is prime.

However, we have just mentioned that nobody knows a quick way to find any or all factors of a number. The standard way of proving a number is prime is by showing that it and 1 are its only factors. For the same reasons that factoring is hard, the simple approach to primality testing, test all possible divisors, is much too slow. If we did not have a faster way to check whether a number is prime, the RSA system would be useless.

In August of 2002, Agrawal, Kayal and Saxena announced an algorithm for testing whether an integer n is prime which they can show takes no more than the twelveth power of the number of digits of n to determine whether n is prime, and in practice seems to take significantly less time. While the algorithm requires more than the background we are able to provide in this book, its description and the proof that it works in the specified time uses only results that one might find in an undergraduate abstract algebra course and an undergraduate number theory course! The central theme of the algorithm is the use of a variation of Fermat's Little Theorem.

In 1976 Miller[8] was able to use Fermat's Little Theorem to show that if a conjecture called the "Extended Reiman Hypothesis" was true, then an algorithm he developed would determine whether a number n was prime in a time bounded above by a polynomial in the number of digits of n. In 1980 Rabin[9] modified Miller's method to get one that would determine in polynomial time whether a number was prime without the extra hypothesis, but with a probability of error that could be made as small a positive number as one might desire, but not zero. We describe the general idea behind all of these advances in the context of what people now call the Miller-Rabin primality test. As of the writing of this book, variations on this kind of algorithm are used to provide primes for cryptography.

We know, by Fermat's Little Theorem, that in Z_p with p prime, $x^{p-1} \bmod p = 1$ for every x between 1 and $p-1$. What about x^{m-1}, in Z_m, when m is not prime?

Exercise 2.4-5 Suppose x is a member of Z_m that has no multiplicative inverse. Is it possible that $x^{n-1} \bmod n = 1$?

We answer the question of the exercise in our next lemma.

Lemma 2.25 *Let m be a non-prime, and let x be a number in Z_m which has no multiplicative inverse. Then $x^{m-1} \bmod m \neq 1$.*

Proof: Assume, for the purpose of contradiction, that

$$x^{m-1} \bmod m = 1.$$

Then

$$x \cdot x^{m-2} \bmod m = 1.$$

But then $x^{m-2} \bmod m$ is the inverse of x in Z_m, which contradicts the fact that x has no multiplicative inverse. Thus it must be the case that $x^{m-1} \bmod m \neq 1$. ■

This distinction between primes and non-primes gives the idea for an algorithm. Suppose we have some number m, and are not sure whether it is prime or not. We can run the following algorithm:

[8] G.L. Miller. "Riemann's Hypothesis and tests for primality," J. Computer and Systems Science **13**, 1976, pp 300-317.

[9] M. O. Rabin. "Probabilistic algorithm for testing primality." Journal of Number Theory, **12**, 1980. pp 128-138.

```
PrimeTest(m)
choose a random number $x$, $2 \leq x \leq m-1$.
compute $y = x^{m-1} \bmod m$
if $(y = 1)$
    output ‘‘ $m$ might be prime’’
else
    output ‘‘$m$ is definitely not prime’’
```

Note the asymmetry here. If $y \neq 1$, then m is definitely not prime, and we are done. On the other hand, if $y = 1$, then the m might be prime, and we probably want to do some other calculations. In fact, we can repeat the algorithm Primetest(m) for t times, with a different random number x each time. If on any of the t runs, the algorithm outputs "m is definitely not prime", then the number m is definitely not prime, as we have an x for which $x^{m-1} \neq 1$. On the other hand, if on all t runs, the algorithm Primetest(m) outputs "m might be prime", then, with reasonable certainty, we can say that the number m is prime. This is actually an example of a *randomized algorithm*; we will be studying these in greater detail later in the course. For now, let's informally see how likely it is that we make a mistake.

We can see that the chance of making a mistake depends on, for a particular non-prime m, exactly how many numbers a have the property that $a^{m-1} = 1$. If the answer is that very few do, then our algorithm is very likely to give the correct answer. On the other hand, if the answer is most of them, then we are more likely to give an incorrect answer.

In Exercise 12 at the end of the section, you will show that the number of elements in Z_m without inverses is at least $\sqrt{m}$. In fact, even many numbers that do have inverses will also fail the test $x^{m-1} = 1$. For example, in Z_{12} only 1 passes the test while in Z_{15} only 1 and 14 pass the test. (Z_{12} really is not typical; can you explain why? See Problem 13 at the end of this section for a hint.)

In fact, the Miller-Rabin algorithm modifies the test slightly (in a way that we won't describe here[10]) so that for any non-prime m, at least half of the possible values we could choose for x will fail the modified test and hence will show that m is composite. As we will see when we learn about probability, this implies that if we repeat the test t times, and assert that an x which passes these t tests is prime, the probability of being wrong is actually 2^{-t}. So, if we repeat the test 10 times, we have only about a 1 in a thousand chance of making a mistake, and if we repeat it 100 times, we have only about a 1 in 2^{100} (a little less than one in a nonillion) chance of making a mistake!

Numbers we have chosen by this algorithm are sometimes called *pseudoprimes.* They are called this because they are very likely to be prime. In practice, pseudoprimes are used instead of primes in implementations of the RSA cryptosystem. The worst that can happen when a pseudoprime is not prime is that a message may be garbled; in this case we know that our pseudoprime is not really prime, and choose new pseudoprimes and ask our sender to send the message again. (Note that we do not change p and q with each use of the system; unless we were to receive a garbled message, we would have no reason to change them.)

A number theory theorem called the Prime Number Theorem tells us that if we check about $\log_e n$ numbers near n we can expect one of them to be prime. A d digit number is at least 10^{d-1}

[10] See, for example, Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, McGraw Hill/MIT Press, 2002

and less than 10^d, so its natural logarithm is between $(d-1)\log_e 10$ and $d\log_e 10$. if we want to find a d digit prime, we can take any d digit number and test about $d\log_e 10$ numbers near it for primality, and it is reasonable for us to expect that one of them will turn out to be prime. The number $\log_e 10$ is 2.3 to two decimal places. Thus it does not take a really large amount of time to find two prime numbers with a hundred (or so) digits each.

Important Concepts, Formulas, and Theorems

1. *Exponentiation.* To perform exponentiation mod n efficiently, we use repeated squaring, and take mods after each arithmetic operation.

2. *Security of RSA.* The security of RSA rests on the fact that no one has developed an efficient algorithm for factoring, or for finding x, given x^e mod n.

3. *Fermat's Little Theorem does not hold for composites.* Let m b e a non-prime, and let x be a number in Z_n which has no multiplicative inverse. Then $x^{m-1} \bmod m \neq 1$.

4. *Testing numbers for primality.* The randomized Miller-Rabin algorithm will tell you almost surely if a given number is prime.

5. *Finding prime numbers.* By applying the randomized Miller-Rabin to $d\ln 10$ (which is about $2.3d$) numbers with d digits, you can expect to find at least one that is prime.

Problems

1. What is 3^{1024} in Z_7? (This is a straightforward problem to do by hand.)

2. Suppose we have computed a^2, a^4, a^8, a^{16} and a^{32}. What is the most efficient way for us to compute a^{53}?

3. A gigabyte is one billion bytes; a terabyte is one trillion bytes. A byte is eight bits, each a zero or a 1. Since $2^{10} = 1024$, which is about 1000, we can store about three digits (any number between 0 and 999) in ten bits. About how many decimal digits could we store in a five gigabytes of memory? About how many decimal digits could we store in five terabytes of memory? How does this compare to the number 10^{120}? To do this problem it is reasonable to continue to assume that 1024 is about 1000.

4. Find all numbers a different from 1 and -1 (which is the same as 8) such that $a^8 \bmod 9 = 1$.

5. Use a spreadsheet, programmable calculator or computer to find all numbers a different from 1 and $-1 \bmod 33 = 32$ with $a^{32} \bmod 33 = 1$. (This problem is relatively straightforward to do with a spreadsheet that can compute mods and will let you "fill in" rows and columns with formulas. However you do have to know how to use the spreadsheet in this way to make it strightforward!)

6. How many digits does the 10^{120}th power of 10^{100} have?

7. If a is a 100 digit number, is the number of digits of $a^{10^{120}}$ closer to 10^{120} or 10^{240}? Is it a lot closer? Does the answer depend on what a actually is rather than the number of digits it has?

8. Explain what our outline of the solution to Exercise 2.4-1 has to do with the binary representation of 10^{120}.

9. Give careful pseudocode to compute $a^x \bmod n$. Make your algorithm as efficient as possible. You may use right shift n in your algorithm.

10. Suppose we want to compute $a^{e_1 e_2 \cdots e_m} \bmod n$. Discuss whether it makes sense to reduce the exponents mod n as we compute their product. In particular, what rule of exponents would allow us to do this, and do you think this rule of exponents makes sense?

11. Number theorists use $\varphi(n)$ to stand for the number of elements of Z_n that have inverses. Suppose we want to compute $a^{e_1 e_2 \cdots e_m} \bmod n$. Would it make sense for us to reduce our exponents mod $\varphi(n)$ as we compute their product? Why?

12. Show that if m is not prime, then at least $\sqrt{m}$ elements of Z_m do not have multiplicative inverses.

13. Show that in Z_{p+1}, where p is prime, only one element passes the primality test $x^{m-1} = 1$. (In this case, $m = p + 1$.)

14. Suppose for RSA, $p = 11$, $q = 19$, and $e = 7$. What is the value of d? Show how to encrypt the message 100, and then show how to decrypt the resulting message.

15. Suppose for applying RSA, $p = 11$, $q = 23$, and $e = 13$. What is the value of d? Show how to encrypt the message 100 and then how to decrypt the resulting message.

    Applying the extended GCD algorithm, or just by experimenting we see that $d = 17$. $n = pq = 253$. Then

$$\begin{aligned} 100^{13} \bmod 253 &= 10^{26} \bmod 253 = (-12)^8 \cdot 100 \bmod 253 \\ &= (100(12^4 \bmod 253)(12^4 \bmod 253)) \bmod 253 = 133. \end{aligned}$$

    To reverse the process,

$$133^{17} = (((133^4 \bmod 253)^4 \bmod 253) \cdot 133) \bmod 253 = 210 \cdot 133 \bmod 253 = 100.$$

16. A digital signature is a way to securely sign a document. That is, it is a way to put your "signature" on a document so that anyone reading it knows that it is you who have signed it, but no one else can "forge" your signature. The document itself may be public; it is your signature that we are trying to protect. Digital signatures are, in a way, the opposite of encryption, as if Bob wants to sign a message, he first applies his signature to it (think of this as encryption) and then the rest of the world can easily read it (think of this as decryption). Explain, in detail, how to achieve digital signatures, using ideas similar to those used for RSA. In particular, anyone who has the document and has your signature of the document (and knows your public key) should be able to determine that you signed it.