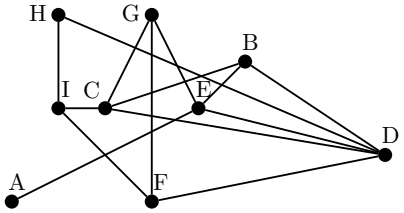
Chapter 3

Public-channel cryptography

3.1 The Perfect Code cryptography system

The Perfect Code cryptography system is based around an object called a *graph*. We are not talking about the graph of a function. A graph is essentially a collection of points (called vertices) connected by lines (called edges). For example, the graph below has 9 vertices and 16 edges.

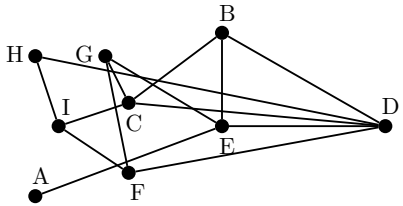


We have given the vertices labels so that we can easily refer to them. For example, notice that the vertex H is adjacent (connected by an edge to) vertex D, but not to vertex C.

The *degree* of a vertex is the number of other vertices it is adjacent to. So, for example, A has degree 1 in the above graph, while C has degree 4.

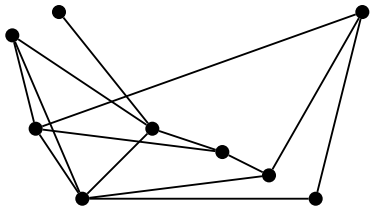
Ex. 3.1.1. What is the degree of vertex I?

Below is the same graph, drawn a slightly different way:



It is the same graph because vertices which were adjacent in the first graph are still adjacent, while vertices not adjacent in the first graph are not adjacent here either.

It can be surprisingly difficult to tell whether two graphs are actually the same. For example, the following graph is actually the same graph as the previous two already drawn. Can you figure out how the vertices match up???



Ex. 3.1.2. Assign the labels A through I to the vertices in this graph so that it matches the labeling given in the drawings above. Note for example that since A and E are adjacent in the drawings of the graph above, A and E must be adjacent in your labeling of the graph. On the other hand, since C and E are not adjacent in earlier drawings of the graph, they cannot be adjacent in your labeling of this graph. *Hint:* To get you started: note that in the earlier graphs, A was the only vertex of degree 1. Thus it is easy to tell which vertex must be A. And now it should be easy to figure out which vertex is E as well. . . .

A **perfect code** in a graph consists of a set of vertices with the following two properties:

1. None of the vertices in the set are adjacent to each other

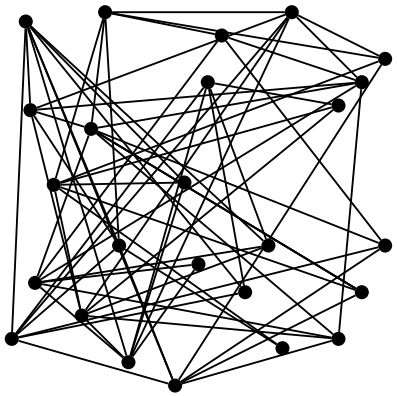
2. Every vertex from the graph which is not in the set is adjacent to *exactly one* vertex from the perfect code.

In the graph discussed above (including the graph from the exercise if labeled correctly!) the vertices E and I together form a perfect code, because they are not adjacent to each other, and every other vertex is adjacent to exactly one of them. (A is adjacent to E, B to E, C to I, D to E, F to I, G to E, and H to I.)

Graphs can have more than one perfect code, but it turns out that there are no more for this particular graph. For example, the vertices E, F, and H, are all not adjacent to each other, and all other vertices are adjacent to at least one of these. However, some other vertices are adjacent to more than one of these

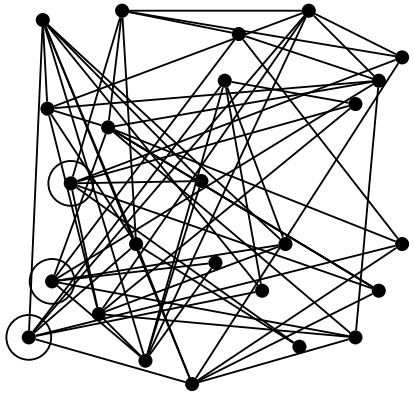
(*e.g.,* I is adjacent to both F and H). Thus, E, F and H do not form a perfect code.

An interesting observation is that there seems to be *no quick way* to find perfect codes in a graph. For example, consider the following graph:

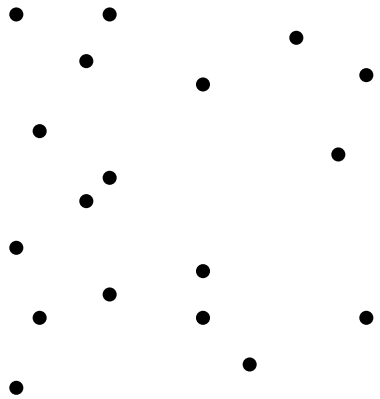


This graph has a perfect code, but it would be very difficult to find it. What would you do if asked to locate it? It seems you would essentially have to try all possible sets of vertices, checking each one against the perfect-code conditions. This would take a long time!

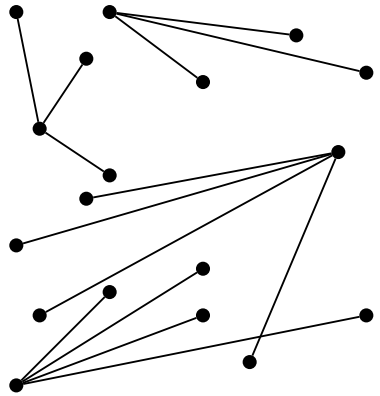
Incidentally, below is the same graph with a perfect code (consisting in this case of three vertices) circled. You can check that these vertices are not adjacent to each other, and that every other vertex in the graph is adjacent to exactly one of them.



In spite of the fact that it can be quite difficult to *find* a perfect code in a graph, it is quite easy to *make* a graph in which we know a perfect code. For example, suppose we begin with the following vertices:



We can now pick a few vertices arbitrarily, and connect each other vertex in the graph to one of the chosen vertices:



These vertices (four of them in this case) will be the perfect code. To finish off, we can add edges arbitrarily between some of the vertices *not in the perfect code* (Figure 3.1.)

Now we know where the perfect code is, but it wouldn't be obvious to someone who hadn't seen us making the graph.

If Alice wants to send Bob a message using the perfect code cryptography system, then Bob (the intended *recipient* of a message) must first create a graph in which he knows the location of a perfect code. He can reveal the graph to Alice (or anyone else, for that matter), but must keep secret the location of the perfect code. To keep things simple, let's assume Bob creates the graph discussed at the beginning of this section, shown again here:

This graph is Bob's public key: he can publish it for everyone to see. In particular, he makes it available to Alice. Alice can now send Bob a message consisting of a number. (She can either convert a text message into a number, of use the number she sends as the key for a symmetric cipher). Suppose that

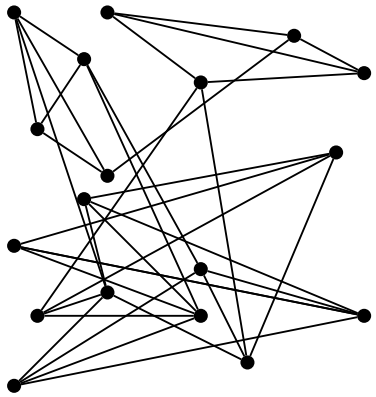


Figure 3.1: In the final graph, the perfect code cannot be easily found.

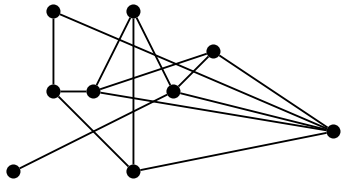


Figure 3.2: Bob creates a graph in which he knows the location of a perfect code (circled). He sends the graph to Alice, *without revealing the perfect code.*

Alice wants to send Bob the message '17'. She begins by 'spreading out' the number 17 across the vertices of the graph. By this we mean that she writes a number next to each vertex of the graph, so that the sum of all of the numbers is 17 (Figure 3.3).

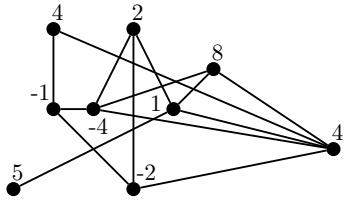


Figure 3.3: First Alice 'spreads out' the message 17 on the graph...

At this point, the message has still not been encrypted. Anybody intercepting the graph with these labels can recover the original message simply by adding all the numbers together.

To encrypt the message, Alice makes a new copy of the graph in which she labels each vertex with the sum of its label, together with the labels of all the vertices it is adjacent to. For example, the upper-left vertex is relabeld from 4 to $4+(-1)+4=7$, while the lower left vertex is relabeled from 5 to $5+1=6$. We call this operation the *clumping* operation. The complete relabeling is shown in Figure 3.4.

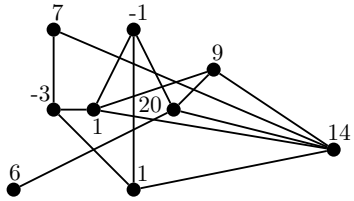


Figure 3.4: ... then Alice creates a new 'clumped' labeling of the graph, in which each vertex is labeled with the sum of its original label and the original labels of its adjacent vertices.

After completing this summing operation, the original message can no longer be recovered by simply summing the labels at all the vertices (now we would get 46, which is not the message!). Thus Alice sends this message to Bob over open channels to Bob, without worrying about an eavesdropper being able to deduce the message.

At this point, it is perhaps not obvious how Bob can decrypt the message. The key is that Bob knows a perfect code in the graph (this is Bob's "private key"). By adding the labels of the vertices in the perfect code, he can recover the original message (Figure 3.5).

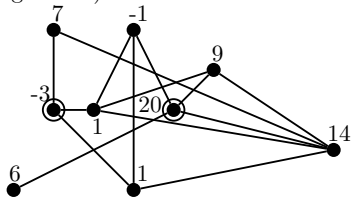


Figure 3.5: Bob decrypts the message by adding the labels at the perfect code vertices. (In this case, he recovers 17.)

Why does this recover the original message? Each label in the graph Bob receives is a sum of labels from vertices in the graph Alice first made when she 'spread out' her message. Since the perfect code is adjacent to every vertex in the graph exactly once, the labels at the perfect code now repesent the sum of *all* the vertices in the graph on which Alice 'spread out' her message; therefore, their sum is itself the message.

Ex. 3.1.3. You want to send Robert a message consisting of the number 30. Assume Robert sends you the graph from Figure 3.1 to use for the perfect-code

system. Give two labelings of this graph: the labeling produced by 'spreading out' your message on the graph, and the labeling which you would actually send to Robert. Check that Robert will recover the correct message.

It is worthwhile to emphasize just how amazing this situation appears to be. It implies that two people can communicate secret messages without sharing *any* secret information ahead of time at all! Even if one of the parties wishing to exchange messages knew nothing of the Perfect Code system, it could be cpmpletely explained in the presence of eavesdroppers, without the need for any secret keys to be exchanged.

It turns out that the Perfect Code system is not really secure: whlie it is true that an eavesdropper cannot count on being able to find a perfect code in an 'overheard' graph, and so decrypt the message in the same way the intended recipient will, an eavesdropper can solve a system of linear equations to deduce the 'spread-out' labeling which produced the transmitted 'clumped' labeling, thus allowing them to find the original message by simply summing the 'spread-out labels'.

Ex. 3.1.4. How can an eavesdropper who doesn't know the perfect code deduce the 'spread-out' labeling of graph from the 'clumped' labeling by solving a system of linear equations? In particular, what system of equations would an eavesdropper solve if they intercepted the graph in Figure 3.5? *Hint:* In general, they will solve a system of n equations in n variables, where n is the number of vertices in the graph.

The Perfect Code system was based on the fact that it can be very difficult to *find* perfect codes in a graph, in spite of the fact that it is easy to *make* graphs with known perfect codes. In spite of the fact that the Perfect Code system is not actually secure, it still hints at the possibility of a secure public-channel cryptosystem. In the next sections, we will learn about the RSA cryptosystem, which is a real and secure public-key system based on the fact that it can be hard to find the factorization of large numbers, even though it is easy to make a large number whose factorization is known (by simply multiplying together some prime numbers).

3.2 RSA warmup

Before we cover the RSA cryptosystem, we have a few more preliminaries to get out of the way. In this section, we'll begin by covering 'KidRSA', a toy public-key system that also uses modular arithmetic. We'll learn how to break KidRSA using the Euclidean algorithm, which will also be necessary for the use of the 'real' RSA system. Finally, we'll learn how to carry out fast modular exponentiation, a cornerstone of RSA encryption and decryption.

3.2.1 KidRSA

Like RSA, KidRSA is a public-key cryptography system where the public and private keys are pairs of integers. Like the Perfect Code system, the KidRSA system is not at all secure, and can in fact be broken very easily. One reason we cover KidRSA is that the method used to break the cipher will be an integral part of the RSA system.

Suppose Alice wants to send a message to Bob. As usual, Bob, as the recipient, has to all the legwork at the beginning by creating a public/private keypair. Bob does this as follows:

KidRSA key generation

1. Choose integers a, b, a', b'.
2. Set $M = ab - 1$.
3. Set $e = a'M + a$ and $d = b'M + b$.
4. Set $n = (ed - 1)/M$.
5. (e, n) is the public key, (d, n) private key.

Bob would now publically disclose the public key (e, n) to Alice, keeping the private key (d, n) secret. Encryption is simply by modular multiplication modulo n. For example, Alice encrypts a message $m \pmod n$ by computing $c = e \cdot m \pmod n$ (c stands for ciphertext)[1]. Bob will decrypt the ciphertext by computing $d \cdot c \pmod n$. This will recover the original message since d will be the multiplicative inverse of e modulo n. Referring to the key generation steps above, this is because $n = (ed - 1)/M$ is a divisor of $ed - 1$ (and so $ed \equiv 1 \pmod n$).

The 'point' of the key generation steps for KidRSA is just that they allow us to pick, in a systematic way, numbers e and and d for which we know some nontrivial divisor of $ed - 1$ (M, in this case), allowing us to choose n to be a divisor of $ed - 1$ ($(ed - 1)/M$ in this case), ensuring that $ed \equiv 1 \pmod n$ will hold. Put another way, the important thing is that in step 4 of the key generation, setting $n = (ed - 1)/M$ is guaranteed to give an integer!

Ex. 3.2.1. Show that, when the steps 1-3 of the KidRSA key generation steps are followed, M is always a divisor of $ed - 1$. (In other words, show that the n produced by the key generation steps is an integer).

Let's consider an example. Suppose Bob begins his key generation with the choices $a = 5, b = 3, a' = 7, b' = 5$. It will proceed as follows:

1. $a = 5, b = 3, a' = 7, b' = 5$
2. $M = 14$.
3. $e = 7 \cdot 14 + 5 = 103$ and $d = 5 \cdot 14 + 3 = 73$
4. $n = (103 \cdot 73 - 1)/14 = 537$.
5. $(103, 537)$ is the public key, $(73, 537)$ private key.

[1] Note that Alice's message has to be a positive number m which is less than n. In practice, this means that messages either have to be quite short, or need to be broken up

Bob would now publish $(103, 537)$. If Alice wants to encrypt the message '30', she would compute $103 \cdot 30 \pmod{537}$, which gives 405. Upon receiving the message 405, Bob would compute $405 \cdot 73 \pmod{537}$, which recovers 30.

Ex. 3.2.2. Carry out the KidRSA key generation procedure with the choices $a = 5, b = 4, a' = 3, b' = 6$. Indicate which key you publish, and which you keep secret.

Ex. 3.2.3. ...continuing the above problem. Someone has used your public key to send you the encrypted message 27. What is the decrypted message?

3.2.2 The Euclidean Algorithm

The KidRSA system is based on the fact that we can create pairs of numbers which are inverse to each other under multiplication to some modulus, even without a way to find the inverse of any particular given number to a given modulus. Thus, in the previous example, Bob was able to produce the numbers 103, 73, and 537 satisfying $103 \cdot 73 \equiv 1 \pmod{537}$ but it is not obvious how Alice, knowing only the numbers 103 and 537, could solve the congruence $103 \cdot x \equiv 1 \pmod{537}$ to recover the secret number 74. Recall that for the classical ciphers (*e.g.*, the affine cipher), we computed inverses simply by searching the modulo 26 multiplication table. If Alice wanted to use this same method to break KidRSA, each break would require producing columns of large multiplication tables.

There is, however, a very fast and efficient way to find modular inverses, called the Euclidean algorithm. We will first consider the simpler task of using it to compute gcd's.

At first glance, computing gcd's seems like a straightforward task: the gcd of 15 and 25 is 5, for example, since clearly this is the greatest divisor common to both the numbers. Things look less optimistic when numbers get bigger, however. Suppose, for example, that we want to compute $\gcd(707, 980)$. Running through all possible divisors 'by hand' now seems like an unpleasant task. However, a simple but elegant observation can save us all that trouble.

The important observation is simply that the gcd of 707 and 980 is the same as the gcd of 707 and $(980 - 707)$; since the gcd divides both 707 and 980, it divides $(980 - 707)$ as well. Thus we have reduced the problem of finding $\gcd(707, 980)$ to one of finding $\gcd(707, 273)$. Why stop now, though? We can use the subtraction trick again. In fact, 273 can be subtracted from 707 multiple times, reducing the problem to finding $\gcd(707 - 273 - 273, 273) = \gcd(161, 273)$. By subtracting 161 from 273, we have reduced the problem to finding $\gcd(161, 112)$. The next reduction gives $\gcd(49, 112)$. 49 can be subtracted twice from 112, giving $\gcd(49, 14)$[2]. 14 comes out 3 times from 49, leaving $\gcd(7, 14)$. Finally, we can subtract two 7's from 14, leaving $\gcd(7, 0)$, which is 7.

The procedure of subtracting to find the gcd used in the previous paragraph is called the *Euclidean algorithm*. It is often simplest to carry out the

[2]At this point, it should be clear what the gcd is, but it is worthwhile to see how the process ends.

calculations by arranging the numbers in a single-column table, of sorts. For example, for $\gcd(707, 980)$ we would begin by putting the bigger number above the smaller one:

980
707

As before, at each step, we subtract the last number from the one immediately preceding it—or, if it goes in more than once, we divide and take the remainder. Each time we add the result to the bottom of the column. For example, 980-707 was 273, so the column becomes

980
707
273

After we have finished the gcd calculations already carried out in the previous paragraph, the finished column would be

980
707
273
161
112
49
14
7
0

and the number immediately before 0 (7 in this case) is the gcd. Notice that it works because, as mentioned before, the subtractions don't change the gcd's, so that this column represents a chain of gcd's which are all equal: $\gcd(980, 707) = \gcd(707, 273) = \gcd(273, 161) = \cdots = \gcd(14, 7) = \gcd(7, 0)$.

Ex. 3.2.4. Find the following gcd's.

(a) $\gcd(573, 1042)$

(b) $\gcd(690, 1219)$

(c) $\gcd(695, 842)$

(d) $\gcd(335, 1407)$

3.2.3 The *extended* Euclidean algorithm

While finding gcd's is fun (and often important!), breaking KidRSA involves being able to find modular inverses. A slight extension makes the Euclidean algorithm up to this task.

We can't hope to find the inverse of 707 modulo 980, since it doesn't have one—we discovered already that they are both divisible by 7. Let's try to find the inverse of 707 modulo 979. We will carry out the Euclidean algorithm as before, but do some more "bookkeeping" on the side. We will introduce two more columns to our table, and intialize them with 1's and 0's as shown below:

	979	707
979	1	0
707	0	1

The two extra columns will keep track of how to write the numbers produced by the Euclidean algorithm as the sum (or difference) of multiples of 979 and 707. For example, the first row of this table should now be interpreted as saying that $979 = 1 \cdot 979 + 0 \cdot 707$, while the second means that $707 = 0 \cdot 979 + 1 \cdot 707$.

We now carry out the Euclidean algorithm exactly as before, except that *whatever we do to the numbers in the left column, we do to the right two columns as well.* For example, the next step of the Euclidean algorithm at this point is to subtract 707 from 979 and write the difference beneath. In this case, we will subtract each element of the second row from each element of the preceding row and write the result beneath. $979 - 707$ gives 272, $1 - 0$ gives 1, and $0 - 1$ gives -1, so the table becomes

	979	707
979	1	0
707	0	1
272	1	-1

This new row can be interpreted as saying that $272 = 1 \cdot 979 - 1 \cdot 707$ (and it's true!). At this point, the next step was to take the remainder after dividing 707 by 272. Since 272 goes twice into 707, this is the same as subtracting $2 \cdot 272$ from 707, so we will subtract 2 times each element of the last row from each element in the previous row. We get $707 - 2 \cdot 272 = 163$, $0 - 2 \cdot 1 = -2$, and $1 - 2 \cdot (-1) = 3$:

	979	707
979	1	0
707	0	1
272	1	-1
163	-2	3

The new row tells us that $163 = -2 \cdot 979 + 3 \cdot 707$. 163 goes once into 272, so to continue we subtract this row once from the previous row:

	979	707
979	1	0
707	0	1
272	1	-1
163	-2	3
109	3	-4

The new row means that $109 = \cdot 979 - 4 \cdot 707$. Next 109 goes once into 163, giving

	979	707
979	1	0
707	0	1
272	1	-1
163	-2	3
109	3	-4
54	-5	7

Now 54 goes twice into 109, so we subtract two times the final row from the previous row:

	979	707
979	1	0
707	0	1
272	1	-1
163	-2	3
109	3	-4
54	-5	7
1	13	-18

Since we got to 1, it means that 979 and 707 are relatively prime. Note now that this last row means that $1 = 13 \cdot 979 - 18 \cdot 707$. Rearranging things, this means that $(-18) \cdot 707 = 1 + (-13) \cdot 979$. In other words, $(-18) \cdot 707$ is 1 plus a multiple of 979. *So -18 is a multiplicative inverse of 707 modulo 979*!!

This is how the extended Euclidean algorithm gives us multiplicative inverses. After completing the extended table with inputs a and b, the final row (if the a and b are relatively prime) gives us an equation of the form $1 = xa + yb$, where 1, x, and y are the numbers from the last row. We see that x is the inverse of a modulo b, and y is the inverse of b modulo a. Assuming that b is less than a, it is usually y (the inverse of b modulo a) that we are interested in!

Note that, although the extended Euclidean algorithm often gives negative numbers as inverses (as in -18 in the above case), these are always equivalent to some positive number: for example, (-18) is an inverse of 707 modulo 979, but $-18 \equiv 961 \pmod{979}$, so we have that the inverse of 707 modulo 979 is 961.

Ex. 3.2.5. Find the following modular inverses—or, when there is none, give the greatest common divisor of the two numbers.

(a) $269^{-1} \pmod{552}$.

(b) $321^{-1} \pmod{529}$.

(c) $641^{-1} \pmod{1000}$.

(d) $105^{-1} \pmod{196}$.

3.3 Fast Modular Exponentiation

The purpose of this section is just to learn how to quickly compute things like

$$17^{288} \pmod{50}.$$

Although, at first glance, this problem would appear to take 288 multiplications, there is actually a much faster way to to carry out this problem.

Let's first consider a slightly different problem, however. Suppose we wanted to compute

$$17^{256} \pmod{50}.$$

Recall that $256 = 2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$. Therefore, we have that

$$17^{256} = 17^{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2} = 17^{2^{2^{2^{2^{2^{2^{2}}}}}}}$$

Thus $17^{256} \pmod{50}$ can be computed just by squaring 17 eight times:

$$\begin{array}{c} 17 \\ 39 \\ 21 \\ 41 \\ 31 \\ 11 \\ 21 \\ 41 \\ 31 \end{array}$$

Here the squarings are simply written in succession: $17^2 \equiv 39 \pmod{50}$, $39^2 \equiv 21 \pmod{50}$, and so on. The final result means that $17^{256} \equiv 31 \pmod{50}$. We calculated $17^{256} \pmod{50}$ with just 8 operations! And the fact that we reduce by the modulus after each step means we are never working with numbers that are particularly large.

What about the earlier problem of $17^{288} \pmod{50}$? Since 288 is not a power of 2, we can't just do the repeated squaring trick. However, notice that $288 = 256 + 32 = 2^8 + 25$. This implies that

$$17^{288} = 17^{2^{2^{2^{2^{2^{2^{2}}}}}}} \cdot 17^{2^{2^{2^{2}}}}$$

In other words, to compute $17^{288} \pmod{50}$, we can multiply the result of 8 squarings of 17 by the result of 5 squarings of 17 (modulo 15). Referring to the column above, 8 squarings of 17 modulo 50 gives 31, while 5 squarings gives 11. Thus

$$17^{288} \equiv 31 \cdot 11 \equiv 41 \pmod{50}.$$

Of course, *any* number can be written as the sum of some powers of 2 (this is how we represent a number in binary). Thus we can always essentially use this same trick. For example,

$$17^{91} = 17^{64} \cdot 17^{16} \cdot 17^{8} \cdot 17^{2} \cdot 17^{1}$$

Thus we can compute 17^{91} by computing

$$17^{91} = 17^{2^{2^{2^{2^{2}}}}} \cdot 17^{2^{2^{2^{2}}}} \cdot 17^{2^{2^{2}}} \cdot 17^{2} \cdot 17.$$

Again referring to the earlier column, this implies that

$$17^{91} \equiv 21 \cdot 31 \cdot 41 \cdot 39 \cdot 17 \equiv 33 \pmod{50}.$$

Ex. 3.3.1. For each of the following problems, rewrite the exponentiation as the product of powers all of whose exponents are powers of 2. Then use repeated squaring to solve the problem.

(a) $11^{80} \pmod{27}$

(b) $15^{43} \pmod{25}$

(c) $17^{165} \pmod{29}$

(d) $8^{250} \pmod{31}$

Fast modular exponentiation can be streamlined still a little bit more. We will demonstrate the streamlined procedure with the problem $17^{91} \pmod{50}$. It may not be obvious at first how the streamlined procedure is related to the method demonstrated above, but we will look at that in a moment!

The procedure is carried out in a 3 column table, with the first row initialized with the entries a, b, and 1, where we are computing $a^b \pmod{n}$. In this case, the first row of our table is

squarings	exponent	result
17	91	1

As indicated, the columns keep track of squarings of the base, the 'remaining exponent', and the 'result'. New rows of the table are added according to the following two cases:

Case 1: if the last exponent value is even, then the new exponent value is set as half the previous exponent value, and the new result value is replaced with the previous base times the previous result (reduced to the modulus—in this case 50). (The base value remains unchanged.)

Case 2: if the last exponent value is odd, then the new exponent value is set as 1 less than the previous exponent value, and the new base value is set as the square of the previous base value. (The result value remains unchanged).

With the table above, the exponent value of 91 means that we're in Case 2. Thus the next row of the table will have exponent value 90 and result value $17 \cdot 1 \equiv 17$, while the base remains unchanged at 17:

squarings	exponent	result
17	91	1
17	90	17

90 is even so we are now in Case 1: the exponent gets halved, the base squared, and the result remains unchanged:

squarings	exponent	result
17	91	1
17	90	17
39	45	17

45 is odd so we are in Case 2. We subtract 1 from the exponent, leave the base unchanged, and multiply the result by the base to get the new result:

squarings	exponent	result
17	91	1
17	90	17
39	45	17
39	44	13

44 is even so we are in Case 1. We halve 44, square 39, and leave 13 alone:

squarings	exponent	result
17	91	1
17	90	17
39	45	17
39	44	13
21	22	13

Case 1 again: we halve 22, square 21, and leave 13 alone:

squarings	exponent	result
17	91	1
17	90	17
39	45	17
39	44	13
21	22	13
41	11	13

Case 2: we subtract 1 from 11, and multiply 13 by 41:

squarings	exponent	result
17	91	1
17	90	17
39	45	17
39	44	13
21	22	13
41	11	13
41	10	33

Continuing in this fashion, the table completes to:

squarings	exponent	result
17	91	1
17	90	17
39	45	17
39	44	13
21	22	13
41	11	13
41	10	33
31	5	33
31	4	23
11	2	23
21	1	23
21	0	33

The answer (33 in this case) is found in the result value in the last column (exponent value 0).

Why does this streamlined procedure work?? It turns out that it is carrying out exactly the same 'repeated squarings' calculations we demonstrated earlier. The important things happen at the steps where the previous exponent was odd (notice that these are the only steps when the result is changed). Let's consider what happens at each of those steps (boldfaced in the table below):

squarings	exponent	result
17	91	1
17	**90**	**17**
39	45	17
39	**44**	**13**
21	22	13
41	11	13
41	**10**	**33**
31	5	33
31	**4**	**23**
11	2	23
21	1	23
21	**0**	**33**

At the first boldfaced step, the result changes from 1 to

$$17^1 = 17.$$

By the second boldfaced step, the base has been squared ($17^2 \equiv 39 \pmod{50}$), and this is multiplied by the result to give

$$17^2 \cdot 17^1 \equiv 13 \pmod{50}.$$

By the third boldfaced step, the base has been squared three times ($17^{2^{2^2}} \equiv 41 \pmod{50}$) and this is multiplied by the previous result to give

$$17^{2^{2^2}} \cdot 17^2 \cdot 17^1 \equiv 33 \pmod{50}.$$

Continuing in this manner, the fourth boldfaced step brings the result to

$$17^{2^{2^{2^2}}} \cdot 17^{2^{2^2}} \cdot 17^2 \cdot 17^1 \equiv 23 \pmod{50},$$

while the last brings it to

$$17^{2^{2^{2^{2^2}}}} \cdot 17^{2^{2^{2^2}}} \cdot 17^{2^{2^2}} \cdot 17^2 \cdot 17^1 \equiv 33 \pmod{50},$$

Notice that this last line is exactly the same calculation we did before considering the streamlined method. The streamlined method just 'automates' the procedure of splitting the exponent into powers of two, and reuses previous squaring calculations (for exmaple, when $17^{2^{2^{2^2}}}$ is calculated, it is done by squaring the previously calculated $17^{2^{2^2}}$, rather than starting over from 17).

Ex. 3.3.2. Solve each of the following problems using the streamlined method of repeated squarings. (Note that these are the same problems as from exercise 3.3.1.

(a) $11^{80} \pmod{27}$

(b) $15^{43} \pmod{25}$

(c) $17^{165} \pmod{29}$

(d) $8^{250} \pmod{31}$

Apart from the fact that it works, another absolutely important property of the fast-exponentiation methods are that they are fast enough to find even really gigantic modular exponents. To see why this is, let's imagine we use the streamlined exponentiation method to calculate $a^{57823153} \pmod{m}$, for some numbers a and m. We don't care what they are, actually, because we're not going to bother to actually do the calculation. We just want to see how many steps it would take. To see this, we can just fill in what would be the center column of the fast exponentiation table. Table 3.1 shows the resulting column of exponents, along with the operations performed at each step to give the next value.

Table 3.1 shows that there would be 41 steps required for the exponentiation. While it might not be fun to have to do these steps by hand, it is perhaps nevertheless remarkable that so few are required, considering the size of the exponent. The key to the efficiency of the algorithm lies in the fact that many of the steps involve cutting the exponent in half. In fact, looking at the table, it becomes clear that at least half of the steps innolve cutting the exponent in

exponent	
57823153	-1
57823152	/2
28911576	/2
14455788	/2
7227894	/2
3613947	-1
3613946	/2
1806973	-1
1806972	/2
903486	/2
451743	-1
451742	/2
225871	-1
225870	/2
112935	-1
112934	/2
56467	-1
56466	/2
28233	-1
28232	/2
14116	/2
7058	/2
3529	-1
3528	/2
1764	/2
882	/2
441	-1
440	/2
220	/2
110	/2
55	-1
54	/2
27	-1
26	/2
13	-1
12	/2
6	/2
3	-1
2	/2
1	-1
0	

Table 3.1: The exponent column from a fast modular exponentiation with exponent 57823153.

half, since it can never happen that we have an odd exponent twice in a row (since an odd number minus 1 is never odd!). This means that every two steps, the exponent has always been reduced by at least half. This means that every 8 steps, we cut the exponent down to at most $(\frac{1}{2})^4 = \frac{1}{16}$ of its former value. Since 16 is more than 10, this means every 8 steps, we definitely decrease the number of digits in our number by at least 1. Looking at Table 3.1, it seems that thenumber of digits decreases even faster than that; every 5 steps or so. That's because often times the exponent gets halved several times in a row.

All this means that even with really giant numbers, carrying out fast modular exponentiation would be feasible. For example, for a 200 digit exponent, modular exponentiation would certainly take less than $8 \cdot 200 = 1600$ steps, and would probably actually take closer to $5 \cdot 200 = 1000$ steps. These thousands of steps might be tedious to do by hand, but would take a computer a fraction of a second.

Ex. 3.3.3. Exactly how many steps would it take to compute $17823941^{82521821} \pmod{748283710527193}$ using fast modular exponentiation?

Ex. 3.3.4. Approximately how many steps would you expect it to take to compute

$$3^{5645684937625171293474564628199181726364763819191273764361819} \pmod{2385712349}$$

using fast modular exponentiation?

It turns out that the same kind of analysis done here shows that the Euclidean algorithm (and its extended version) from the previous sections also have similar efficiency properties. Consider the following calculation of the greatest common divisor of 57313 and 14233:

$$\begin{array}{r} 57313 \\ 34233 \\ 23080 \\ 11153 \\ 774 \\ 317 \\ 140 \\ 37 \\ 29 \\ 8 \\ 5 \\ 3 \\ 2 \\ 1 \\ 0 \end{array}$$

It seems clear that the numbers decrease very quickly. The important observation is that just like with fast exponentiation, *every two steps, the current value is decreased at least by half.* To see why this is the case, imagine that at some point in the table we have the two values:

$$\begin{array}{c} a \\ b \end{array}$$

If b is more than half of a, then the next element in the table will simply be $a - b$:

$$\begin{array}{c} a \\ b \\ a-b \end{array}$$

But since b was more than half of a, now $a - b$ is less than half of a, and so the value has indeed been decreased by half in at most 2 steps. Looking at the earlier calculation for $\gcd(57313, 14233)$, however, it is clear that it occasionally decreases even much more quickly than that.

3.4 RSA

In this section, we introduce the RSA public-key encryption algorithm. RSA was first described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman, and was the first publically known example of a secure, practical algorithm that could be used to encrypt secret messages without somehow establishing a secret key beforehand. Before the 1970's, nobody even had any reason to think that such a remarkable thing should even be possible.

As a public-key cryptography system, RSA calls on the intended recipient of an encrypted message to create a public and private key. He publishes the public key, which anyone can then use to encrypt messages to him, and keeps the private key secret.

In the case of the perfect code system, the ability to create a public/private keypair depended on the ease with which one can create graphs with known perfect codes, in spite of the difficulty associated with finding them. In the case of KidRSA, the ability to create a keypair depended on a method to create triples of numbers e, d, n such that $d = e^{-1} \pmod{n}$, in spite of the difficulty (we thought!) associated with finding modular inverses.

The RSA public/private keypair generation is based on the difficulty of factoring large integers. Given a number like 15, it's easy enough to determine that it factors as $15 = 3 \cdot 5$. When we try to factor a larger number, like 1,010,021, it becomes apparent that factoring the number 15 depended largely on our ability to simply try all possible divisors, since 15 was quite small. We don't know any good way to factor a larger number like 1,010,021—certainly, trying all possible factors would be a bit tedious by hand. In real-life computer implementations of RSA, the numbers which would have to be factored to break the system are hundreds of digits long, and would take even the worlds fastest computers thousands or millions of years to factor. Of course, we have already seen in the in the perfect code and KidRSA systems that the 'hard problem' public-key systems are based on tend to have an assymmetry to them (easy to do one way, hard to do the other). In the case of factoring, this assymetry comes from the fact that it is very easy to generate numbers for which we know the factors... simply

multiply some prime numbers together! Multiplication is very fast. In fact, even for hundred-digit numbers, this would in principle be possible to do by hand (although it might take you a couple days!). A computer can do such long multiplications in microseconds.

KidRSA turned out not to be secure because, well, finding modular inverses is easy after all, once we know the extended Euclidean algorithm! Although noone knows a way to easily find perfect codes in graphs, the perfect code system turned out not to be secure, because an attacker could circumvent the system by solving a system of equations, decrypting the message without ever finding out the perfect code. RSA is a cryptosystem where no one has figured out a way to break the underlying problem (by finding a way to quickly factor integers), and no one has found a way to circumvent the system by breaking it without factoring integers.

We'll discuss some of these issues later, but first let's go over the mechanics of RSA. Suppose Bob wants to receive RSA messages and so needs to generate a public/private keypair. He does this as follows.

RSA key generation

1. Choose distinct prime numbers p and q.
2. Set $n = pq$, and $\phi(n) = (p-1)(q-1)$.
3. Choose any e relatively prime to $\phi(n)$ and set $d = e^{-1} \pmod{\phi(n)}$
4. Set (e, n) is the public key, while (d, n) is the private key.

Unlike KidRSA, where encryption and decryption were done with multiplication, RSA encryption and decryption are done with exponentiation: given a message m, Alice computes the ciphertext as $c \equiv m^e \pmod{n}$. Bob decrypts the ciphertext by computing $c^d \pmod{n}$. *It should not be obvious at all at this point why this recovers the original message.* It does though, and we will examine why in section 3.6. For now, let's try an example.

Suppose Bob wants to create a private/public keypair, and chooses $p = 3$, and $q = 11$. Then $n = 33$, and $\phi(n) = 20$. If he chooses $e = 3$, then d is $3^{-1} \pmod{20}$, which is 7. Thus Bob publishes the public key $(3, 33)$. His private key is $(7, 33)$.

If now Alice wants to encrypt the message '5' using Bob's public key, she computes $5^3 \equiv 26 \pmod{33}$. When Bob receives the message 26, he decrypts by calcuating $26^7 \pmod{33}$, which will recover the plaintext 5.

Something seems quite strange about this, however. How can this be secure? Knowing that $n = 33$ and $e = 3$, can't Alice figure out that $\phi(n) = 20$ and then compute the inverse of $3 \pmod{20}$ to find d, just like Bob did? At first glance, this seems reasonable, but figuring out what $\phi(n)$ is from n requires *factoring* n (for example, we could tell that $\phi(33) = 20$ since we know that $3 \cdot 11 = 33$ and $2 \cdot 10 = 20$.) For large numbers, this just won't be feasible. But it does bring up an important point: apart from keeping d secret, Bob must keep $\phi(n)$ secret as well, and, by extension, the primes factors p and q of n, since these can easily be used to compute $\phi(n)$.

One final point to consider is whether RSA can actually be carried out efficiently. We've already said that it should be infeasible for someone to break

RSA by factoring large integers, but maybe even generating RSA keys and encrypting messages is already too cumbersome to be practical! Fortunately, this is not the case. Let's examine the steps of RSA key generation.

The first step, choosing prime numbers p and q, should actually seem quite problematic at this point. Remember that, in practice, the numbers that used for RSA are hundreds of digits long—in particular, they are much too long for us to be able to factor numbers of that size. It turns out that, in spite of the fact that we don't know ways of factoring really large numbers, there are nevertheless good ways of finding really large prime numbers. We won't cover that until Section 3.5.2, however. For now, we will just assume that we have access to some way of generating large prime numbers.

The second step of the process involves only subtraction and multiplication, so is easy enough.

In the third step, we need to be able to choose e relatively prime to $\phi(n)$, and then compute the inverse of e modulo $\phi(n)$. Both of these tasks can be done with the extended Euclidean algorithm: we can choose an e we hope will be relatively prime to $\phi(n)$, and carry out the extended Euclidean algorithm with e and $\phi(n)$. If the algorithm returns a gcd of 1, then the numbers are in fact relatively prime, and it also gives the inverse! If it gives a gcd greater than 1, we try a different value for e and run the algorithm again.

Finally, the actual encryption and decryption operations, consisting of modular exponentiations, can be done very efficiently using the streamlined method of fast modular exponentiation.

Ex. 3.4.1. Generate a public/private RSA keypair using the primes $p = 11$, $q = 23$. (You should use the extended Euclidean algorithm.) (Note: since you get to choose e, there is more than one possible answer to ths problem!)

Ex. 3.4.2. If you receive the message 124, encrypted using the public key you produced in the previous problem, what does the message decrypt to?

3.5 Primes

The RSA cryptosystem depends heavily on special properties of primes numbers to work. Prime numbers $(2, 3, 5, 7, 11, \dots)$ are positive numbers with exactly two positive divisors. For example, 2 is prime because its only positive divisors are 1 and 2. If we count negative divisors as well, primes have exactly 4 divisors (*e.g.*, -2, -1, 1, and 2).

Prime numbers have been studied for thousands of years, and yet there is still we much we don't know about them. Consider the fact that, among small numbers at least, it is easy to think of several examples of consecutive odd integers which are both prime: 3 and 5, 11 and 13, 17 and 19, 29 and 31, and so on. Even though people have studied the question for hundreds of years, Mathematicians still do not know whether such pairs occur infinitely often. This is known as the "Twin Prime Conjecture".

Ex. 3.5.1. Using a table of prime numbers, check what percentage of the prime numbers ≤ 100 occur in 'twins'. Also check the percentages for primes ≤ 200 and ≤ 300.

The mathematical thereom that there are infinitely many prime numbers goes back to Euclid:

Theorem 3.5.1. *There are infinitely many prime numbers.*

In other words, there is no 'biggest' prime number. We could try to satisfy ourselves of this fact by looking for really big prime numbers.[3]. But even after finding giant prime numbers, the possibility would remain that there might eventually be a 'biggest' one—just perhaps far too large for us to ever find.

To establish the truth of Theorem 3.5.1, then, we need to give a *proof*. A proof is a series of logical deductions showing that a statement holds. For example, here's a proof that the sum of two even numbers is even:

> An even number is a number that can be written as 2 times some other number. Thus if we have two even numbers a and b, we can write them as $a = 2c$ and $b = 2d$, and their sum as $a + b = 2c + 2d$. But then $a + b = 2(c + d)$, and so their sum is also 2 times some number, and so is even.

Euclid's theorem on the infinitude of primes is more substantial (and less obvious!) than the fact that even numbers sum to even numbers, so we should not be surprised that its proof will be more complicated than this.

We will prove Theorem 3.5.1 *by contradiction*. This means that we will assume that the theorem is false, and then arrive at a contradiction though logical steps. By arriving at a contradiction, we can conclude that the original assumption (that the theorem is false) must have been false, mean that the theorem must be true! The idea of the proof by contradiction is illustrated by the following reasoning by someone who wants to know whether or not it recently rained:

> Let's assume it recently rained. Then there must be puddles in the street. But I can see that there are no puddles in the street—this is a contradiction. Thus it must not have recently rained.

The kind of reasoning above is different from a mathematical proof, in that the successive statements don't really necessarily follow from one another. For example, it may be possible for it to rain without making puddles in some way that we haven't imagined[4]. In a mathematical proof, deductions are not just 'reasonable' like with the discussion on the rain, they absolutely follow without any assumptions. Comparing the rain 'proof' to the mathematical proof given above regarding the sum of even numbers, it becomes clear that there are actually lots of (reasonable) assumptions that are part of our everyday reasoning, but not part of mathematical proofs.

[3] [Enter current record here]

[4] maybe the street is covered with a giant tent, for example

Ex. 3.5.2. Identify another unjustified assumption made in the rain 'proof', and give a (possible fanciful) example of how it might be false.

Since we are proving Euclid's theorem by contradiction, we will start by assuming that there are only finitely many primes. This means that it would be possible (in theory[5]) to list them in order:

$$2, 3, 5, 7, 11, 13, 17, \ldots, p_L$$

where p_L is some 'last' prime number. We aren't assuming anything about how many primes there are; just that it is a finite number.

Imagine now that we define a number s by multiplying all of these prime numbers together and adding 1 to the result:

$$s = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot \cdots \cdot p_L + 1.$$

Now we can ask: which prime numbers is s divisible by? Well, since it is 1 more than a multiple of 2, it cannot be divisible by 2. And since it is 1 more than a multiple of 3, it cannot be divisible by 3. And since it is 1 more than a multiple of 5, it cannot be divisible by 5. And so on. All the way up to: it is 1 more than a multiple of p_L, so it cannot be divisible by p_L. But, currently, we are working under the assumption that these primes from 2 to p_L are all the prime numbers! This means that p_L has no prime factors, but this is impossible as any number must have at least one prime factor (possibly itself). Thus we arrived at a contradiction when we assumed Euclid's theorem was false, and so Euclid's theorem is true.

Ex. 3.5.3. As part of the proof above, we used the fact that every number has at least 1 prime divisor. Prove this statement. *Hints:* Remember that a prime number is just any number with 2 positive divisors. For any number s, let k denote the number of positive divisors of s. You would like to find a divisor of s with 2 positive divisors (if $k = 2$ then s is itself such a divisor and you are done). If k is bigger than 2, than consider some divisor s_1 of s other than 1 and itself; let k_1 be the number of positive divisors of s_1. Then $k_1 < k$ (why? you must explain this!) Continuing this argument, you can show that there must be some divisor of s for which the number of positive divisors is 2.

In the time since Euclid lived, mathematicians have proved many things about the prime numbers. Nevertheless, many things about the prime numbers (like the twin prime conjecture) remain unknown and mysterious.

Mathemtacians' fascination with prime numbers may stem in part from the contrast between their straightforward definition and the seemingly random 'behavior'. For example, in spite of the fact that prime numbers have a simple definition and satisfy many strict mathematical relations[6], there seems to be

[5] we might run out of paper in practice, for example

[6] for example, any number $1 \leq a < p$ always has an inverse modulo p if p is prime; also, Fermat's little Theorem, covered in the next section, is a good example.

no obvious pattern to which numbers are prime, and, for example, there are no known formulas to simply generate the prime numbers one after another.

This duality between their deterministic and random-like behaviors was remarked upon by Don Zagier in a lecture in 1975:

> There are two facts about the distribution of prime numbers of which I hope to convince you so overwhelmingly that they will be permanently engraved in your hearts. The first is that, despite their simple definition and role as the building blocks of the natural numbers, the prime numbers grow like weeds among the natural numbers, seeming to obey no other law than that of chance, and nobody can predict where the next one will sprout. The second fact is even more astonishing, for it states just the opposite: that the prime numbers exhibit stunning regularity, that there are laws governing their behavior, and that they obey these laws with almost military precision.

This fascination with the 'random-like' behavior of prime numbers has been around for centuries. In fact, one of the biggest open problems in mathematics, the Riemann hypothesis, is so captivating precisely because it would allow mathematicians to make a very precise statement about the random-like distribution of prime numbers. On the other hand, one of the most significant solved problems of the 150 years was the so-called 'prime number theorem', which states approximately how many prime numbers there are up to different points. If we define $\pi(n)$ to be the number of prime numbers $\leq n$ (so that, for example, $\pi(2) = 1$ and $\pi(8) = 4$), the prime number theorem states that

Theorem 3.5.2. *The number $\pi(n)$ of prime numbers $\leq n$ satisfies*

$$\pi(n) \sim \frac{n}{\ln(n)}.$$

The function $\ln n$ is the natural log—the logarthm to the base e. It is a very slow growing function. For example, the number called a googol (a 1 folowed by a hundred zeros) is far bigger than the number of particles in the entire universe. Nevertheless, its natural logarithm is only $230.2585\ldots$. Thus, in a certain sense, the prime number theorem tells us that even among really giant numbers, the primes still aren't too rare (*e.g.*, even among numbers around the size of 1 googol, around 1/230 numbers are prime). Note that the number $e = 2.718\ldots$ shows up in lots of unexpected places—for example, in the formula Se^p for the amount remaining in a savings account which contained an initial balance S at interest rate p compounded continuously. One mysterious aspect of the prime number theorem is that it implies a connection between prime numbers and the constant e.

The symbol $\sim$ in the theorem means that it is an approximation that gets arbitrarily accurate as n gets big: in other words, the percentage error of the approximation gets arbitrarily small as n gets big.

Ex. 3.5.4. For each of the following values of $\pi(n)$, calculate the approximation given by the prime number theorem and find the true value (using a table of primes), and find the percentage error in the estimate given by the prime number theorem: $\pi(25)$, $\pi(50)$, $\pi(100)$, $\pi(200)$, $\pi(400)$.

In the next section, we will prove an old and fundamental result about prime numbers called Fermat's little Theorem; this theorem will be used both to see why RSA works, *and* in the actual key-generation process of RSA, to find the large prime numbers necessary for the procedure.

3.5.1 Fermat's little Theorem

The subject of this section is a mathematical theorem called Fermat's little Theorem, or FℓT, for short. This theorem simply states that for any prime number p, and any number a which is not a multiple of p, a^{p-1} is always congruent to 1 modulo p

Theorem 3.5.3 (FℓT)**.** *For any prime number p and any number a relatively prime to p, we have*

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

This is, at first glance, a slightly mysterious fact. Let's try it out with a 'random' example. Suppose we compute $3^{12} \pmod{13}$. This can be done quickly with fast modular exponentiation:

squarings	exponent	result
3	12	1
9	6	1
3	3	1
3	2	3
9	1	3
9	0	1

and sure enough we get 3.

If the exponent/modulus is not prime, on the other hand, it doesn't necessarily work, as in the case of $3^{11} \pmod{12}$:

squarings	exponent	result
3	11	1
3	10	3
9	5	3
9	4	3
9	2	3
9	1	3
9	0	3

It[7] gives 3, not 1. On the other hand, it does sometimes happen when the exponent isn't prime, as is the case for $8^{20} \equiv 1 \pmod{21}$ (which you can check

[7] Note that this table looks a little funny, since it turns out that $9 \cdot 9 \equiv 9 \pmod{12}$, and $9 \cdot 3 \equiv 3 \pmod{12}$.

with fast modular exponentiation). The point of Fermat's little Theorem is that it tells us that the congruence $a^{p-1} \equiv 1 \pmod{p}$ *always* holds so long as p is a prime number.

The Proof

As a theorem, FℓT is a statement which is supposed to always hold true, no matter what, so long as its conditions are satisfied. As a result, we cannot justify FℓT by checking lots of examples—these could only serve to convince us that lots of instances of the congruence $a^{p-1} \equiv 1 \pmod{p}$ are true. If we only try examples, there always remains the possibility that there is some instance of the congruence that doesn't hold (possibly with absurdly large numbers) which we just haven't tried yet.

Thus, just like for Euclid's theorem in the previous section, we want to give a mathematical proof of Fermat's little theorem. The proof of Fermat's little theorem will be more difficult than the proof of Euclid's theorem.

To begin the proof, consider the list

$$a, 2a, 3a, \ldots, (p-1)a$$

of the first $(p-1)$ multiples of a. It turns out that *none* of these can be congruent to each other modulo p. To see why, imagine that two of them are actually congruent[8], *i.e.,* that we have $ka \equiv \ell a \pmod{p}$, where k and ℓ are distinct numbers between 1 and $p-1$. Since a is relatively prime to p, it has an inverse modulo p, and so we can multiply both sides of this congruence by it to give $k \equiv \ell \pmod{p}$. This is a contradiction, since k and ℓ are supposed to be different, so our assumption that $ka \equiv \ell a \pmod{p}$ must not actually be impossible, and so all of the multiples $a, 2a, 3a, \ldots, (p-1)a$, must really be different.

Since the numbers $a, 2a, 3, a \ldots (p-1)a$ are all congruent to different things modulo p, they must actually be congruent to each of the numbers $1, 2, 3, \ldots, (p-1)$ modulo p exactly once, just not in that order. That means that if we multiply them all together, we have:

$$a \cdot 2a \cdot 3a \cdots (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdots (p-1) \pmod{p}.$$

Written another way,

$$a^{p-1} \cdot (p-1)! \equiv (p-1)! \pmod{p}. \tag{3.1}$$

And now we're almost done! Observe that since $(p-1)!$ is not divisible by p, $(p-1)!$ must actually be relatively prime to p. Thus we can multiply both sides of the congruence in line (3.1) by the inverse of $(p-1)!$ (modulo p) to get the congruence

$$a^{p-1} \equiv 1 \pmod{p}. \tag{3.2}$$

[8] this part of the proof is a little proof by contradiction within the larger proof

Remember, this holds only under the assumption that a is not a multiple of p. If we multiply both sides of this congruence by a, we get the congruence from Fermat's little Theorem, which we have already seen also holds when a is *not* relatively prime to p.

Ex. 3.5.5. FℓT only tells us that $a^{p-1} \equiv 1 \pmod{p}$ for prime values of p, and for other values, it is not always true (for example, for $p = 8$, the theorem would be false since $3^7 \equiv 3 \pmod{8}$. Where in the proof did we use that p is a prime number? Identify the step in the proof that depended on p being a prime number.

3.5.2 Finding Primes

Fermat's little Theorem can actually be used to solve a major problem for us: finding the large primes necessary for RSA keypair generation. Why is finding such large primes such a problem? Suppose we have chosen a large number like

$$s = 421249166674228746791672110734681729275580381602196445017243909397$$

and would like to check that it is prime. What can we do? We need to make sure that it is not divisible by any (positive) numbers other than itself and 1. The simplest way to check this would be to just try dividing s by 2, 3, 4, 5, 6 *etc.* But there are $s-2$ numbers between 1 and s, so this requires dividing numbers almost s times! In this case, the number s is 66 digits long, and all the computers in the world working together couldn't do that many divisions in a trillion years (seriously).

You might have noticed, though, that it is unnecessary to try dividing by numbers like 4 and 6 since it is enough to check whether s has any prime factors. Also, if s has any divisor d other than 1 and itself, it turn out that it must have some divisor (other than 1) which is at most $\sqrt{n}$: if d is itself at most $\sqrt{n}$ then that divisor is d; if d is bigger than $\sqrt{n}$ than s/d (which is also an integer and a divisor of s) must be smaller than $\sqrt{n}$. This means that we only need to check for divisors up to $\sqrt{s}$ to decided whether or not s is prime; and as noted above, if we had some way of only trying division by prime numbers, then that would be enough.

However, since s is 66 digits long and so roughly 10^{65}, its square root is roughly $\sqrt{10^{65}} = (10^{65})^{1/2} = 10^{32.5} \approx 10^{32}$ Since the natural log of 10^{32} is $\ln(10^{32}) = 73.68\ldots$, the prime number theorem tells us that roughly $1/74$ (so well over 1%) of numbers up to $\sqrt{s}$ will be prime. This leaves roughly $10^{32}/74$, or more than 10^{30} divisions to do. The fastest supercomputer today could maybe do this many divisions in a few hundred thousand years or so—still too long to wait for Bob to make his RSA keys! And this was just for a 66 digit number. The numbers used for RSA can be hundreds of digits long.

Ex. 3.5.6. Assuming one only needs to divide by primes up to $\sqrt{n}$, use the same techniques as in the paragraphs above to estimate how many divisions it could take to determine whether a 100 digit number (so, around 10^{99}), is prime.

The motivation for the *Fermat primality test* is that Fermat's little theorem tells us a property that that all prime numbers have; namely, that they satisfy the congruence $a^{p-1} \equiv 1 \pmod{p}$ for any values of a from 1 to $p-1$ (since p is prime, all these values of a are relatively prime to p). Maybe then, if we pick a random number q and check for lots of values of a that $a^{q-1} \equiv 1 \pmod{q}$ (which we can do using fast exponentiation), then it is likely to be prime? Remember, the prime number theorem tells us that primes are not too uncommon[9], so a computer would not have to make too many guesses before stumbling upon even really large prime numbers.

To formalize this notion, consider the following process:

	given a possible prime $q\ldots$
1	pick a random $1 < a < q$.
3	Compute $a^q \pmod{q}$.
4	If $a^{q-1} \not\equiv 1 \pmod{q}$, then q is **not prime**.
5	If $a^{q-1} \equiv 1 \pmod{q}$, then q may be prime. (Go back to step 1?)

Look at step 3. Since FℓT tells us that $a^{q-1} \equiv 1 \pmod{q}$ if q is prime and a is between 1 and q, then if we find such an a for which this congruence does not hold, then q must not be prime! There are two ways this can happen: the first way is if a is not relatively prime to q. Since $1 < a < q$, this can't happen if q is prime, and so it means that q has to be composite.

The second way this can happen is if a *is* relatively prime to q, but the Fermat congruence still doesn't hold. Since this violates Fermat's little Theorem, this also shows that q isn't prime. Such a value a is called a 'Fermat witness', because it uses Fermat's little Theorem to 'witness' the fact that q is a composite number.

Step 4 is also interesting. In this case, we've tried an a and found that $a^{q-1} \equiv 1 \pmod{q}$. This case, unfortunately doesn't really tell us anything. q *may* be prime, but maybe not. We already pointed out in the last section that $8^{20} \equiv 1 \pmod{2}1$. And this happens in lots of other cases as well. For example, for $q = 341$, if we tried $a = 2$ we would compute $2^{340} \equiv 1 \pmod{341}$, and suspect that 341 is prime. Upon trying $a = 3$, however, we would compute $3^{341} \equiv 56 \pmod{341}$ and discover that it is not (some trial and error reveals that $341 = 11 \cdot 31$).

The question is, then: for how many a's do we need to compute $a^q \equiv a \pmod{q}$ before we can decide that q is prime? Are there any numbers q which are not prime, but for which $a^q \equiv a \pmod{q}$ holds for *all* a? In other words, are there any composite numbers that have no Fermat witnesses at all? It turns out that there are. The smallest such number is 561.

Ex. 3.5.7. Choose any number a between 2 and 560 and use fast exponentiation to check that $a^{560} \equiv 1 \pmod{561}$.

Fortunately, such numbers, called Carmichael numbers, are very rare. Until 1994 it was not even known if there were infinitely many of them. After 561, the next smallest is 1105, then 1729. As numbers get bigger, they get even rarer. In fact, there are so few large Carmichael numbers that if we randomly choose a hundred-digit number, there is less than a $\frac{1}{10^{32}}$ chance that it will be a Carmichael number.

[9] $\ln(10^{99}) = 227.95\ldots$, so roughly 1 out of every 228 hundred digits are prime

What about composite numbers which are not Carmichael numbers? For those we know that there is at least one Fermat witness—so, at least one a between 1 and p for which $a^p \not\equiv a \pmod{p}$. But that's not that useful, actually. If we have to try $p-2$ fast exponentiations to decide that p is probably prime, we might as well just do trial division (at least then we only have to check up to $\sqrt{p}$). It turns out, however, that any composite number which has at least one Fermat witness has lots and lots of them:

Theorem 3.5.4. *If a (composite) number q has at least one Fermat witness, then at least half the numbers relatively prime to q are Fermat witnesses for q.*

The proof of Theorem 3.5.4 is actually relatively straightforward. Assume as stated in the theorem that q is a composite number and has a relatively prime Fermat witness a. Recall that this means that a is a number which is relatively prime to q but for which $a^{q-1} \equiv s \pmod{a}$, where $s \not\equiv 1 \pmod{q}$. Now choose any number b which is relatively prime to q, but is not a Fermat witness for q (so, $b^{q-1} \equiv 1 \pmod{q}$). Then it turns out that $a \cdot b$ must be a Fermat witness to q.

To see why, first observe that since a and b are relatively prime to q, $a \cdot b$ is also relatively prime to q. Next, we need to check that $(a \cdot b)^{q-1}$ is not congruent to 1 modulo q. We have:

$$(a \cdot b)^{q-1} \equiv a^{q-1} b^{q-1} \equiv s \cdot 1 \equiv s \pmod{q},$$

and so $a \cdot b$ is a Fermat witness for q.

Assume now that we have a whole bunch of different numbers

$$b_1, b_2, b_3, \ldots b_k$$

between 1 and q all of which are *not* Fermat witnesses to q. What we said above implies that

$$a \cdot b_1, a \cdot b_2, a \cdot b_3, \ldots, a \cdot b_k$$

are all Fermat witnesses[10]. This means that there are at least as many numbers relatively prime to q which *are* Fermat witnesses as there are which are not. So the therem is true! □

Theorem 3.5.4 is very important for primality testing with Fermat's little Theorem, because it implies that if q is composite and we randomly choose an a between 1 and q, then if a is relatively prime to q we have at least a 50% chance that we will discover that q is composite by finding that $a^{q-1} \not\equiv 1 \pmod{q}$. On the other hand, it turns out that if a is *not* relatively prime to q, we have a

[10] Moreover, since a is relatively prime to q, these must all be different modulo q: if two were the same, we could multiply both by the inverse of a and find that to of the b_i's were the same, which would be a contradiction.

100% chance of finding this out! That's because if some number d divides both a and q, then it will divide both a^{q-1} and q, and so a^{q-1} will be congruent to a multiple of d modulo q.

Thus, in any case, we see that if q is composite, then there is at least a 50% chance that we will find it out (*i.e.,* that we will end up in step 3 in the Fermat primality test). This seems like its not really good enough: after all, we want to be *really sure* that numbers are prime before using them with RSA. The solution to this problem is simply to *repeat the primality test multiple times so long as it returns 'maybe prime'.* If we ever find a value which returns 'not prime', we choose a different possible prime value q and start over. But if the algorithm keeps returning 'maybe prime' for lots of different randomly chosen values of a, then this means there's an excellent chance that q is prime (unless it is a Carmichael number).

Consider, for example that, if q is composite, that there is at most a 25% chance that the algorithm will return 'maybe prime' twice in a row with different randomly chosen a's. If it is done 3 times, the probability is $.5^3 = .125 = 12.5\%$. After 10 times, the probability drops to $00.0976\ldots\%$, and after 20 times, it drops to $00.0000953\ldots\%$. In fact, if the test was repeated a few hundred times, the probability that it would incorrectly identify a number as prime would be less than the probability of randomly choosing a particular particle in the known universe.

To apply the Fermat primality test in practice, then, we simply repeat the test enough times that we reach a level of confidence we are comfortable with. For example. Suppose we are trying to choose a four digit prime number with the Fermat primality test, and are considering the number 3523. We have checked that it is not divisible by 2, 3, or 5 since these are easy to do without trying any divisons. The next step is to choose some random a's and compute $a^{3522} \pmod{3523}$. Table 3.2 shows the calculation for $a = 2174$.

Since the result is not 1, the test tells us that 3523 is definitely not prime. (It turns out that $3523 = 13 \cdot 271$.) On the other hand, if the test had returned 1, we would have repeated the test a few times until either we got a value different from 1 (revealing that our q was indeed composite) or we had established enough confidence that our q was actually prime.

Ex. 3.5.8. Test the number 1089 for primality using the Fermat primality test (show all your fast exponentiation calculations). Test until the algorithm returns 'not prime', or until it returns 'maybe prime' for 3 different values of a (whichever occurs first)[11].

3.6 RSA: why it works

When introducing RSA back in Section 3.4, we promised that we would eventually explain why RSA works. Let's recall the steps of RSA, shown below along

[11]For a number of this size not divisible by 2, 3, or 5, testing 3 times would be enough to impart a minimum confidence of roughly 80% that it is prime—not enough for real-life RSA implementations, but enough for this exercise.

squarings	exponent	result
2174	3522	1
1933	1761	1
1933	1760	1933
2109	880	1933
1855	440	1933
2577	220	1933
74	110	1933
1953	55	1933
1953	54	2016
2323	27	2016
2323	26	1101
2616	13	1101
2616	12	1925
1790	6	1925
1693	3	1925
1693	2	250
2050	1	250
1665	0	3523

Table 3.2: Calculating $2174^{3522} \pmod{3523}$.

with the methods they depend on:

RSA key generation

1	Choose distinct prime numbers p and q.	p and q are chosen randomly and tested for primality (repeatedly) with the Fermat primality test
2	Set $n = pq$, and $\phi(n) = (p-1)(q-1)$.	
3	Choose any e relatively prime to $\phi(n)$ and set $d = e^{-1} \pmod{\phi(n)}$	Extended Euclidean algorithm
4	Set (e, n) is the public key, while (d, n) is the private key.	Fast exponentiation will be used for encryption and decryption

Remember, encryption of a message m is calculated as $c = m^e \pmod{n}$, while decryption is accomplished by calculating $m = c^d \pmod{n}$. The problem for us now is understanding why decryption will give us back the original message. Why is it true that $(m^e)^d \equiv m \pmod{n}$?

Theorem 3.6.1 (The RSA theorem)**.** *If p and q are prime numbers, and $ed \equiv 1 \pmod{(p-1)(q-1)}$, then for any $0 \leq m < pq$, we have*

$$m^{ed} \equiv m \pmod{pq}$$

This fact, known as the RSA theorem, is closely related to Fermat's little theorem. Here we will *mostly* prove the RSA theorem: we will prove it in the case where m is not a multiple of p or q. The theorem still holds in this case, but the proof is slightly more complicated[12]

By Fermat's little Theorem for the prime number q, we have that

$$m^{(p-1)(q-1)} = (m^{p-1})^{q-1} \equiv 1 \pmod{q}, \tag{3.3}$$

and by Fermat's little Theorem for the prime number p that

$$m^{(q-1)(p-1)} = (m^{p-1})^{q-1} \equiv 1 \pmod{q} \tag{3.4}$$

Line (3.3) means that $m^{(p-1)(q-1)} = 1 + kq$ for some k; while line (3.4) means that $m^{(p-1)(q-1)} = 1+\ell p$ for some ℓ. Since these are equal, we have $1+kq = 1+\ell p$ and so $kq = \ell p$. Since p and q are prime, this is only possible if k is a multiple of p (and ℓ is a multiple of q), in which case we have that $m^{(p-1)(q-1)} = 1 + jpq$ for some j, and so

$$m^{(p-1)(q-1)} \equiv 1 \pmod{pq}. \tag{3.5}$$

Since $ed \equiv 1 \pmod{(p-1)(q-1)}$, we have that ed is 1 more than some multiple of $(p-1)(q-1)$; in other words, we have that $ed = 1 + k(p-1)(q-1)$ for some integer k, and so $m^{ed} = m \cdot m^{k(p-1)(q-1)} = m \cdot (m^{(p-1)(q-1)})^k$. Making the substitution from line (3.5), this gives that $m^{ed} = m \cdot 1^k = m$, as desired.

Ex. 3.6.1. How did this proof depend on the fact that m was not a multiple of p or q?

3.7 RSA + symmetric encryption

There's an issue that we've been glossing over for a while now. Messages to be encrypted with RSA must be numbers, and they must be at most n, the modulus included in the RSA keys. Of course, it is possible to convert text into a number, but even if n is hundreds of digits long (as it is in real-life computer implementations of RSA), this is still hardly big enough to be able to encode a single English sentence as a number modulo n. To get around this, one could imagine breaking a message into small parts, to be encrypted separately. However, if the message is large (perhaps it is a website, which may include images, for example) this would require lots of encryptions. Although the operations used for RSA (*e.g.*, modular exponentiation) can be done efficiently, they still take long enough that we don't want to have to perform them thousands of times when encrypting files.

[12]Note that, when p and q are really large numbers, the chance that m is a multiple of them is really tiny anyways!

The solution to this problem is to encrypt messages using a symmetric encryption method (*e.g.*, AES) and use RSA simply to transmit the keys used for the symmetric method. Thus while the message itself (which may be quite large) is encrypted with a block or stream cipher (both of which are much faster than RSA encryption), only the key (which is potentially much smaller than the message) need be encypted using RSA.

As an example, suppose we have published the RSA public key $(3, 55)$ and have a corresponding private key of $(27, 55)$. We might receive the following message:

> Dear [our name here]. The secret message is
> `QZODK BFUZS WQKEU EQMEK`
> Encryption is with the Caesar cipher. The key, encrypted with your RSA key, is 23.

Notwithstanding the fact that the Caesar cipher is very insecure, this message could be transmitted without any security at all—via email, or even on a billboard. That's because determining the key used for the encryption requires knowing our RSA private key [13]. In this case, since our key is $(27, 55)$, we recover the symmetric encryption key by calculating $23^{27} \pmod{55}$, which gives 12. Subtracting 12 from the Caesar-cipher text recovers the message `ENCRY PTING KEYSI SEASY` ("encrypting keys is easy").

Of course, in real life applications, the symmetric cipher used is not the Caesar cipher, but something secure like AES. Let's see how one could use BabyBlock in tandem with RSA.

Suppose we want to send the message `HURRY` to someone with whom we have not exchanged any secret keys. We can nevertheless encrypt the message using BabyBlock with a key of our choosing—say, 1010. The message `HURRY` becomes 001111010010001100011100 0 in binary, and is padded with three 0's to produce 001111010010001100011100 0000, so that the number of digits is a multiple of 4. BabyBlock encryption with the key 1010 results in a binary message of 11010011110011010111100100000, which converts to `.PG_FY`. Apart from transmitting this message, we want to transmit the key we used, but encrypted with RSA. Note that if we view the key 1010 as a binary number, it corresponds to the number 18. If the recipient has published an RSA public key of $(5, 91)$ we would calculate the encryption $18^5 \equiv 44 \pmod{91}$, and transmit the following to the agent:

> `.PG_FY`
> Encryption is with BabyBlock. The key, viewed as a binary number and encrypted with your RSA key, is 44.

If BabyBlock was substituted for a secure cipher like AES, and RSA was being done with suitably large numbers, such a message could be transmitted in the

[13]or breaking RSA, for example by factoring 55, which with such small numbers is also quite feasible!

open. To recover the message, the recipient will use their private key, which is $(29, 91)$ in this case, to decrypt the key. They calculate $44^{29} \pmod{91}$ and recover 18. By writing 18 in binary as 1010, they find the BabyBlock key which can be used to decrypt the message.

3.8 Digital Signatures

Public-key cryptography is a game-changer for modern communications. It means that people can securely conduct commerce over insecure networks like the internet. It means that people separated by large distances who never meet can nevertheless communicate in privacy.

Nevertheless, there is a separate problem concerning communication at large distances: the problem of authentication.

To introduce the problem, suppose the instructor of a cryptology course sends out homework assignments by email. There is probably no reason to encrypt such emails, since they aren't secret.

Suppose, however, that a student in the class—Albert, say—receives an email with a ridiculous assignment ("do all the problems in the book", for example). Assuming this is out of character for the instructor, Albert may doubt that the message was really sent by the instructor. Actually, it is not too difficult to 'spoof' emails so that they appear to have originated from someone other than the true sender. Thus Albert may suspect that a prankster has sent the email containing the bogus assignment, but has made it seem to originate from the instructor's address. Even if the email is encrypted, this offers no reassurance to Albert that the email was sent by his instructor, as this requires only Albert's public key, which may be public knowledge.

How can the instructor convince Albert that they are the sender of the email? In this case, since the two have met before, they probably have some shared information, so that the instructor could convince Albert by sending an email along the lines of:

> I am really your instructor; I remember 2 weeks ago that you were 20 minutes late and told me your dog ate your homework. And I really sent you that message containing the hard assignment.

But what if they didn't have any shared information? For example, what if the two had never actually met before?

Suppose, along these lines, that instead of being a student in a cryptology class, Albert is a consumer purchasing something from a website, and instead of worrying about whether the assignments he is receiving are authentic, he is worried about whether the website he is about to give his credit card number to is authentic. This authentication problem is dealt with by modern web browsers. They must verify that a website is being run by who it claims to be run by. How can this be accomplished?

It turns out, amazingly, that RSA provides a very simple solution to these authentication problems. In general, the setting in which *Digital Signatures*

are used concerns the situation in which a sender (Alice) wants to convince a receiver (Bob) that they are the true sender of a message (or a website, for example).

Recall that, when Alice wanted to *encrypt* a message to Bob, this required Bob to generate an RSA keypair. However, when Alice wants to *sign* a message (regardless of who it is being sent to), it is Alice who must generate a keypair. Once she has generated a public key (e, n) and a private key (d, n), Alice uses her *private* key to sign a message. Given a message m (some number $1 \leq m < n$), Alice computes the signature as $s \equiv m^d \pmod{n}$. She would transmit this signature along with the original message (which may be encrypted). To verify the message's signature, Bob will use Alice's public key by computing $m \equiv s^e \equiv m^{de} \pmod{n}$. Note that this recovers m for the same reason that RSA decryption recovered an encrypted message (because it turns out that $m^{de} = m^{ed} \equiv m \pmod{n}$). The fact that the signature checks out against the original message using Alice's public key should convince Bob that it was signed with Alice's private key, which only she should know.

Ex. 3.8.1. Assume you have created the keypair $(3, 55)$ (the public key) and $(27, 55)$ (the private key). You wish to produce a signature for the message '9'. Which key do you use? What is the signature?

Ex. 3.8.2. Assume now that you are the recipient of the message and signature from the previous exercise. Carry out the calculation used to verify the signature.

In practice, digital signatures would be completely unworkable without the use of *hash functions*, covered in the next section. Apart from the fact that hash functions are a practical necessity when using digital signatures, as we shall see, hash functions are fascinating in their own right: their security properties turn out to be startingly counterintuitive and seemingly paradoxical.

3.9 Hash functions

Recall that, in practice, RSA is too slow to be used to encrypt entire large files. We get around this problem by encrypting messages using a symmetric ciper (*e.g.,* AES in real life) and just using RSA to transmit the key.

Of course, we have the same problem when we want to sign a message. Suppose, for example, that Alice has told Bob that she will drive him to the airport if he buys her lunch. Being very untrustworthy, and worried that Alice might back out of their deal, Bob types up the following 'contract':

> I Alice agree to drive Bob to the airport this Saturday in exchange for the lunch he is buying me today, Wednesday.

Bob wants Alice to sign the message with her private RSA key and tell him the signature. That way, if she backs out on the deal, Bob can go to the

'authorities', show them that Alice's public key (which is public knownledge) recovers the message from the signature, showing that the contract must have in fact been signed by Alice with her public key.

The problem with this scenario is that it requires turning this message into a number whose size is at most the modulus of Alice's public key. Even with a message this short, there are more than 10^{140} different messages of this length, even if we ignore case, spacing, and punctuation. Thus turning this message into a number will result in a number with at least 140 digits; this is small enough (though not by much) to be encrypted with the giant RSA keys used in computer implementations of RSA, but certainly too long a number for us to work with here. (And, in general, most messages we will want to sign will be far too big even for computer-based implementations of RSA).

How can Alice 'sign' a reasonably sized number that will nevertheless authenticate the entire message as a genuine agreement by Alice? Recall that for encryption, we dealt with the size problem by using RSA to encrypt just the key for a symmetric cipher.... Does that work in this case?

Suppose for example that Alice encrypts the message with the Caesar cipher with they key 2, producing the following:

```
KCNKE GCITG GVQFT KXGDQ DVQVJ GCKTR QTVVJ KUUCV WTFCA KPGZE JCPIG
HQTVJ GNWPE JJGKU DWAKP IOGVQ FCAYG FPGUF CA
```

She could now sign the number 2 with her RSA private key, and tell the signature, the key, and the encrypted contract to Bob.

But Bob now can make it seem like Alice signed a totally different contract. For example, Bob could encrypt the message 'Alice must take Bob to the airport and buy his plane ticket' with the Caesar cipher using the key 2, and the signature would match the resulting encryptd message just as well as it matches the real one. And Alice could play the same game; if Alice did back out on their deal and refuse to take Bob to the airport, Alice could make her own bogus contract (*e.g.,* 'Bob has to buy Alice lunch every day for a month, and she doesn't have to take him to the airport') and claim that this is the contract that she *really* meant to sign.

All of this is just to say that signing keys is no good, since a key can be used to encrypt any message.

This is where hash functions come in. The purpose of a hash function is to take a (possibly quite long) message and return a 'hash' of the message, which is a sort of 'fingerprint' of the message. Alice, wishing to authenticate a message, would sign this hash value rather than the whole message.

Consider the following toy example of a hash function:

The ToyHash algorithm

Given a message, let v be the number of vowels (defined for ToyHash as a, e, i, o, or u), c the number of consonants, and w the number of words. The ToyHash value of the message is defined as

$$(v^2 + cv + w) \text{ MOD } 19$$

Let's return to the example of Alice's promise to Bob:

> I Alice agree to drive Bob to the airport this Saturday in exchange for the lunch he is buying me today, Wednesday.

This message has 37 vowels (not counting any y's), 55 consonants, and 22 words. Thus the ToyHash hash value of this message is

$$(37^2 + 55 \cdot 37 + 22) \text{ MOD } 19 = 6$$

Alice could now sign this hash value (6) with her RSA key and give Bob the result as a digital signature. If Bob tried to claim that Alice had actually signed a different contract, the different contract would likely have a different hash valiue and so would not match Alice's signature. On the other hand, ToyHash will never make a hash value larger than 19, ensuring that even with the small RSA keys we've been using to do calculations by hand, Alice will be able to sign the whole message with just one RSA operation.

Ex. 3.9.1. Assume you have created the keypair $(3, 55)$ (the public key) and $(27, 55)$ (the private key). You wish to produce a signature for the message 'Your instructions are to bombard the enemy with water balloons.' Compute the ToyHash value of this message and sign using RSA. (Which of your keys do you use for this purpose?)

Ex. 3.9.2. Briefly explain why it is necessary to use a hash function for the purpose of creating digital signatures, rather than using RSA to sign the message directly.

Not all hash functions are created equal, however, and it turns out that the ToyHash function is remarkably insecure. What does it mean to say that ToyHash is insecure? Roughly speaking, the problem is that it will be easy to make it seem like Alice signed a different contract than the one she intended to. Returning to the earlier example, recall that ToyHash produces a hash value of 6 on the message 'I Alice agree to drive Bob to the airport this Saturday in exchange for the lunch he is buying me today, Wednesday.' We suppose that Alice has signed this hash value using RSA and told Bob the result as a digital signature of the original message.

If Bob is clever, however, he might try to claim that Alice actually signed the following contract:

I Alice promise to buy Bob an airplane ticket to Hawaii this Saturday in exchange for buying me lunch today, Wednesday

This contract has 40 vowels, 57 consonants, and 21 words, giving a ToyHash value of $(40^2 + 57 \cdot 40 + 21)$ MOD 19) $= 6$, the same as the original contract! Bob will be able to make an untrue but apparently convincing claim that Alice signed this bogus contract, since her signature will match the hash value of this message.

The situation above, where two messages have the same hash value is called a *collision*. Collisions are a problem for hash functions, because each collision represents multiple messages which the hash function cannot distinguish between; for such pairs of messages, neither party can prove which message was actually signed. Finding collisions can involve a bit of luck, but can involve some simple strategies as well. For example, if Bob was looking to go to Cuba instead of Hawaii, he might have found that the bogus contract

> I Alice promise to buy Bob an airplane ticket to Havanna this Saturday in exchange for buying me lunch today, Wednesday

has a ToyHash value of 5 (it has 39 vowels, 59 consonants, and 21 words). By studying the formula $(v^2 + cv \cdot 40 + w)$ MOD 19) $= 6$, he might notice that he can increase the hash value by 1 if he can increase the number of words by 1 without changing the number of vowels of consonants. He can do this by inserting a space by 'air' and 'plane':

> I Alice promise to buy Bob an air plane ticket to Havanna this Saturday in exchange for buying me lunch today, Wednesday

The result has a ToyHash value of 6, and so forms a collision with a genuine contract. Although it is not really common or even correct to write 'air plane' as two separate words, the meaning is clear anyways, and contracts are not typically invalidated because of grammar mistakes!

Ex. 3.9.3. Assume Bob wants to go anywhere in Africa. Find a bogus contract that involves Alice paying his way to a city or country in Africa, whose ToyHash value matches that of the original genuine contract Alice actually signed.

Above we saw a major weakness of ToyHash; collisions are not at all uncommon, and, what's more, they are made easier to find by the fact that it can be easy to predict the impact on the hash value of certain kinds of changes to the message (for example, increasing the number of words). Let's begin by examining the cause of the first problem.

It certainly seems that collisions are going to be made common simply by the fact that ToyHash can only return 19 different hash values (0 through 18). This means that out of any 20 messages, we can be guaranteed that at least 2 share a hash value. Thus it seems that one obvious way to improve ToyHash is to simply increase the number of possible of hash values so that collisions don't

occur. But remember that the very purpose of the hash function is to reduce the size of the value we need sign with RSA. It turns out that to accomplish this at all, we need to accept that there will be collisions!

To see why this is the case, let's imagine that we typically deal with binary messages of around 10,000 digits ($\approx$ 10kb) and use a hash function which produces hash values 100 digits long. The number of possible messages in this situation is roughly $2^{10,000}$, while the number of possible hash values is roughly 2^{100}. There are far more possible messages than hash values, meaning that lots and lots of messages will get the same value!! In fact, there are roughly $2^{10,000}/2^{100} = 2^{9,900}$ times more messages than hash values, meaning that a 'typical' hash value will correspond to $2^{9,900}$ different messages!!! In fact, this same kind of reasoning shows that there will always be collisions if the hash function produces values which are smaller than the sizes of the messages.

Ex. 3.9.4. If we ignore spaces, case, and punctuation in text messages, there are 26^n possible messages that are n letters long (although most of them won't make much sense!) On average, approximately how many different 100-letter-long messages does a ToyHash value correspond to?

The situation described above makes it seems like it will be hopeless to come up with a good hash function. On the one hand, for a hash function to do its job, it must produce a hash value which will typically be small compared to the size of a message; on the other hand, accomplishing this very thing means that the hash function will have lots of collisions, which might be exploited by someone wishing to find bogus contracts which match a signature value they know for another contract.

In spite of this, there are secure hash functions. As already discussed, it is impossible to have a hash function which does its job without producing lots of collisions; the property of secure hash functions is not that such collisions are particularly rare, but simply that they are *hard to find*.

To see how this could be possible, let's again assume that typical messages in some situation consist of 10,000 binary digits, and we are using a hash function that produces hash values 100 digits long. We already pointed out that in such a situation, each hash value will correspond to $2^{9,900}$ messages. It is possible to imagine many hash functions for this situation for which it would be very easy to find a collision. Suppose, for example, that the hash function simply considered the message as a binary number, and reduced it modulo 2^{100}: the result would be a number which could be written as a binary number of at most 100 digits. Collisions are very easy to find in this case. In fact, given *any* message, a collision can be found by considering it as a binary number, and producing another message by adding to it 2^{100}. The problem with this hash function (a problem shared by ToyHash), is that it is easy to predict how the hash value will change with changes to the message.

Suppose, then, on the other hand, that the hash function was very complicated, in a way that made it nearly impossible to figure out how one could produce a message with a certain hash value without simply trying random messages and calculating their hash values. In this case, since there are 2^{100} hash

values, it could take $2^{100} + 1$ guesses before we found two messages with the same hash value[14]. A good hash function is designed to try to make this kind of wild guessing essentially the best strategy. It is perhaps slightly astonishing just how possible this is. SHA-1 is one hash function is widespread use in everyday internet communications; it outputs a 160-bit hash value. In spite of the fact that it has been studied extensively since its release in 1995, and in spite of the fact that for very small messages, 'gazillions' of collisions are guaranteed to exist by the very nature of a hash function, **as of 2009, no one knows even a single instance of a collision for the hash function SHA-1**. And SHA-1 is an old hash function that is being replaced by newer hash functions that produce bigger hash values and are expected to be even more secure. This is the kind-of-amazing thing about hash function security: in spite of the fact that all hash functions will have lots (and lots!) of collisions, it is possible to design a hash function for which we don't expect that anyone will ever be able to find even a single one!!

[14] In this scenario, there is actually a not-too-tricky technique to probably find a collision with just 2^{50} steps... but that's still a lot!