Crypto Book Working Notes

Midge Cozzens and Steven J. Miller

PRINCETON UNIVERSITY PRESS
PRINCETON AND OXFORD

Chapter One

Error Detecting and Correcting Codes

We've discussed at great lengths why it's important to encrypt information. If we have some information that we only want certain individuals to get, it would be foolish to shout it out, or write it out in English and leave it lying on the table for our friend to read. So, clearly, there are times we want to encrypt our message. We then transmit it to our friend, who will then decode it and act accordingly; however, what if the message is garbled in the transmissions? As everyone knows, nothing is perfect, even computers. We've all had experiences where a computer hangs up for no reason, or displays strange text. What happens if there is an error in transmitting our message? For example, we often use binary for our messages, and send strings of 0s and 1s. What happens if one of the 1s is accidentally transmitted as a 0 (or vice-versa)? This leads to the subject of error detection and, if possible, error correction.

While our motivating example dealt with an encoded message, the same issues arise in other situations too. Imagine we want to go to a website to watch some streaming video, maybe our favorite baseball team is playing. The video might not be encrypted, and a string of 0s and 1s are sent to our computer, which creates a movie based on these. If we're thousands of miles away and there are lots of intermediate sites, there could be many chances for digit errors; it would be terrible if there were so many of these that we could not reconstruct the picture.

These transmission errors can have devastating consequences. The following is one of our favorite scenes from *Indiana Jones and the Last Crusade*; the script below was downloaded from http://www.scifiscripts.com/scripts/Indiana3.txt.

> Indy's Speedboat bounces across the choppy waters heading in the direction of the docked steamship. Kazim and his men rush to two more speedboats tied to the dock. They chase after Indy. Indy grapples with the Turkish Agent. As Indy grips his arms, we see a gun in the Agent's hand. It fires. As Indy fights with the Turk, he becomes aware of the Speedboats behind him and two enormous Freighters ahead of him, joined together by two giant ropes. Indy, having gained the advantage, leans on top of the Turkish Agent.
>
> Indy (to Elsa): *Are you crazy?! You don't go between them!*
>
> Elsa can barely hear Indy over the noise of the motor.

> Elsa: *Go between them? Are you crazy?!*
>
> Indy finally delivers the punch that sends the Turkish Agent flying overboard. Turning, Indy sees that Elsa has committed the speedboat to a course between the two Freighters, now being pushed even closer together by a Tugboat.
>
> Indy: *I said go around!*
>
> Elsa: *You said go between them!*
>
> Indy: *I said don't go between them!*
>
> It's purely academic at this point since the hulls of the two Freighters loom up on either side of them like cavern walls.

Unlike poor Indi and Elsa, it is not purely academic for us, or for millions of people around the world. We haven't sent our messages yet, so we have time to think about these issues. Is there a way to send our messages so that, if there is a transmission error, the recipient at least knows an error happened. For example, Elsa didn't hear the phrase "*You don't*". If she knew that part of the message was missing, she could ask Indi to repeat himself. Being able to detect errors is, not surprisingly, called error detection.

While error detection is great, it's incomplete. Frequently it's either expensive to transmit a message again, or there may not be time. For example, in the situation above Indi is in a life-and-death struggle, and he really can't stop and talk again. The ideal situation is for our recipient to know that, not only was an error made in transmission, but to also be able to fix it and get the message that the sender meant to send. This is known as error correction. It's a lot harder than error detection, but amazingly there are really good algorithms to do this.

The purpose of this chapter is to describe some of the issues of error detection and correction. These are vast subjects, pursued by both academics and professionals. The importance of both are clear. We'll just scratch the surface of these topics, but we will get into enough detail to see some of the truly wonderful, amazing methods. To do the subject justice requires a little abstract algebra (group theory), linear algebra and probability; however, we can go a long ways with just some elementary observations. It's frequently the case that it is very hard for someone to have the flash of insight that leads to the discovery of one of these methods; however, it's often not that difficult to follow in their footsteps and learn their method.

1.1 MOTIVATING RIDDLES

Math riddles are more than a fun way for many of us to relax; they can also serve as a nice stepping stone to some advanced theory with interesting applications. This ia particularly true for the two riddles below. The solution to the first introduces a concept which is helpful in designing an error detection code, while the second riddle leads to an idea that can actually correct errors!

Both of these riddles are well-known throughout the mathematics community. We've had a little fun with the phrasing. Try and solve these before reading the answers. For the first, it's possible to get a less than optimal solution that is still better than trivial. Both involve hats that are either white or black; the connection to transmitting data is clear when we replace white with 1 and black with 0.

We'll first state each riddle, and then give some explanatory text about it. After that, we'll discuss a possible approach which is much worse than the best possible. If you want, you can of course skip the rest of this section and go straight to the the definitions of error correcting and error detecting codes. The two riddles below are merely meant to help motivate the material, to help teach you how to look in a clever way at some problems, and to have some important ideas simmering in your mind as you read on.

1.1.1 Statements.

Riddle 1: Imagine 100 mathematicians are standing in a line, wearing a black or white hat. Each mathematician can **only** see the color of the hats of the people in front of them. They close their eyes and you have to put a black or a white hat on each person. You can do this any way you wish; you can give everyone a white hat, or just throw them on randomly. The first person sees no hats, the second person sees just the hat color of the first person, the second sees that hat colors of the first two people, and so on until we reach the last person, who sees the hat colors of the 99 in front of her.

When you say go, the last person says either ‘white’ or ‘black’ but not both; immediately after she speaks the second to last person says either ‘white’ or ‘black’ but not both. This continues until the first person speaks, saying either ‘white’ or ‘black’ (but not both). After all 100 have spoken, we count how many said the color of their hat, and how many said the opposite color. For each person who said their color correctly, you give everyone in line one dollar; however, for each person who was wrong, everyone in line gives you one dollar.

Remember, these are not 100 ordinary people. These are 100 mathematicians, and they are allowed to talk to each other and decide on a strategy to maximize their expected winnings. They are thus extremely intelligent, and if there is a good idea they will find it! You get to put the hats on any way you wish, they get to look and the k^{th} person sees the hat colors of the $k-1$ people in front of them. What is the minimum amount they can guarantee themselves earning? Or, to put it another way, what is the smallest N such that, no matter what you do, at least N of the 100 mathematicians will correctly say their hat color!

Remember, we want to find out how many people we can guarantee say their hat color correctly. Thus, if ever someone is guessing, we have to assume they guess wrong. One possible strategy is for everyone to just say ‘white’; however, you know their strategy. If everyone is going to just say ‘white’, then all you have to do is give everyone a black hat. In this case, $N = 0$, which is really bad. If the hats were to be placed randomly, then yes, this strategy should lead to about half the people saying the correct color, but that is not this problem. In this problem, you can be malicious!

So the mathematicians have to be a bit more clever. Each person cannot be entirely devoted to finding their hat color – they have to somehow help each other. After a little thought, they come up with the following plan. All the even numbered people say the hat color of the person in front of them. By doing this, all the odd people know their hat color! For example, if person 99 has a white hat then person 100 says ‘white’; while she may be wrong, person 99 now says ‘white’ and is correct. If this is their strategy, you’ll of course make sure all the even people have the opposite hat color as the person in front of them; however, there is nothing you can do about the odd people. They will *always* get their hat color right, and thus with this strategy $N = 50$ (in other words, at least half the people will always be right).

So, here’s the question: can you do better than $N = 50$ (i.e., better than 50%)? Amazingly, yes! See how well you can do. Can you do $N = 66$? Or $N = 75$? Clearly the best possible is $N = 99$, as there is no way to ever help the last person.

Is that possible? Can we find a way so that 99 out of 100 are correct?

The next riddle is famous in not just mathematics, but also economics. It too is a hat problem, but unlike the last one (where you could put the hats down however you want), this time the hats are randomly assigned.

Riddle 2: Three mathematicians enter a room, and a white or black hat is placed on each person's head. Each person is equally likely to have a white or a black hat, and the hat assignment of one person has no affect on the hat assignment to anyone else. Each person can see the other people's hats but not their own.

No communication of any sort is allowed, except for an initial strategy session before the game begins. Once they have had a chance to look at the other hats, each person has to decide whether or not to say 'white', 'black' or remain silent. Everyone must speak (or remain silent) at the same time. If everyone who speaks says the color of their hat, then everyone gets one million dollars; however, if even one person who speaks says the wrong color for their hat, then everyone loses one million dollars. If no one speaks, then no money is won or lost.

What is the best strategy for the mathematicians? In other words, what is the largest value of p so that, if this game is played many, many times, then the mathematicians are expected to win p percent of the time, and lose $1 - p$ percent of the time.

As mentioned above, a major difference between the two riddles is that here each hat is randomly assigned; each person gets a white hat half the time and a black hat half the time. There is a very simple strategy that ensures that the mathematicians never lose: no one ever speaks! Unfortunately, with this strategy they also never win.

There is an easy way to make sure they win half the time. One person is told to always say 'white' and the other two are told to always remain silent. Half the time the person will be correct in saying white, and half the time they will be wrong. Thus, we can easily get $p = 1/2$.

Is it possible to do better? It seems absurd to think about getting a p greater than 1/2. After all, each person who speaks says either white or black, and they are equally likely to have a white or a black hat. Doesn't this mean that anyone who speaks will be right half the time and wrong half the time; further, if more people speak it's even worse, as they only win if everyone who speaks is right. It therefore seems impossible to come up with a better strategy, yet there is one, and this strategy will lead to error correcting codes!

1.1.2 Solutions.

Hopefully you and some of your friends had time to play with these riddles. If you haven't, this is your last chance to think about them before seeing the solutions!

First Riddle: There are several strategies for the first riddle, all of which do far better than just half are correct. We give just two. Let's first review the strategy that ensures that at least half are correct. We had all the even people say the hat color of the person in front of them. We used one piece of information to get one piece of information. Can we do better?

Let's examine what person 3 sees. In front of her are two people. There are four, and only four, possibilities: she sees WW, WB, BW, BB (where of course W means a white hat and B a black hat; the first letter denotes the hat of person 1 and the second letter the hat of person 2). Further, person two gets to see person one's hat; there are only two possibilities here: W or B. What we're going to do is have the third person say something which, when combined with what the second person sees, will allow first the second person and then the first person to deduce their hat color. Let's have the third person say 'white' if the two hats have the same color, and 'black' if the two hats have opposite colors. As soon as the third person says this, the first two people know whether or not they have the same or opposite colors. If they have the same, then the second person just says the color of the first person, and then the first person says that color as well; if their colors are opposite, then the second person says the opposite color of what he sees in front of him, and then the first person says the opposite color of the second person.

For example, if the third person sees BW she says 'black', as the hats are different colors. Seeing person 1 wearing a black hat, person two says 'white', and then person one says 'black'. In this way we can get make sure two out of every three people are correct. This means that we can take $N = 66$, or almost two-thirds of the people are guaranteed to say the correct color with this strategy.

Before reading on, see if you can improve our strategy. Can you get four-fifths? How far can you push this?

We now jump to the best strategy. It's absolutely amazing, but we can make sure that 99 out of 100 people are correct! How? The last person counts up how many white hats she sees, and how many black hats. The number of white hats plus the number of black hats must add up to 99, which is an odd number. Thus there has to be an odd number of white hats or an odd number of black hats, but not both. Here's the strategy: the last person says 'white' if there is an odd number of white hats, and 'black' if there is an odd number of black hats.

Why does this strategy work? Let's say the last person sees 73 white hats and 26 black hats. She therefore says 'white' as there is an odd number of white hats. What should the second to last person do? The only difference between what he sees and what the last person sees is that he cannot see his hat color. There are only two possibilities: he sees 72 white hats and 26 black hats, or he sees 73 white hats and 25 black hats. He knows that there is an odd number of white hats. If he sees 72 white hats, he knows that he must be wearing a white hat, as otherwise the last person would not have said white. Similarly, if he sees 73 white hats then he must be wearing a black hat, as otherwise the last person wouldn't have said there are an odd number of white hats.

The process continues. Each person keeps track of what has been said, and whether or not initially there was an odd number of white or black hats. Before speaking each person can see whether or not there are an odd number of white or black hats in front of them, and speak accordingly. Let's continue our example. We'll say there are 73 white hats, and for definiteness let's assume the 99th person has a white hat. Thus the last person says 'white' as there is an odd number of white hats. The second to last person now immediately says 'white', because he sees an even number of white hats (if he didn't have a white hat, the last person would have said 'black'). Now let's look at the 98th person. She knows that the last person saw an odd number of white hats. The second to last person said 'white'. This means that there must be an even number of white hats on the first 98 people. Why? If there were an odd number of white hats on the first 98 people, then the last person would see an even number of white hats (because the odd number from the first 98 plus the one white hat from the 99th person would add up to an even number, which contradicts the last person seeing an odd number). So, the 98th person knows there are an even number of white hats on the first 98 people. If she sees 71 white hats then she says 'white', otherwise she says 'black'.

The key concept in this strategy is that of **parity**. All we care about is whether or not there is an even or an odd number of white hats on the first 99 people. The last person transmits this information, and the other people use it wisely. In the next section we'll expand on the notion of parity and use it to create an error detecting code.

Second Riddle: There is actually a strategy that will work 75% of the time when the hats are randomly placed! Here it is: each person looks at the other two. If you see two hats of the same color, you say the opposite color; if you see two hats of opposite colors, you stay silent.

That's it! It's simple to state, but does it work, and if so, why? Let's tackle whether or not it works first. Imagine the three hat colors are WBW. Then the first person sees BW, the second sees WW and the third sees WB. Only the second person sees two hats of the same color. So only the second person speaks, saying 'black' (the opposite color); the other two people are silent. What if instead it was WWW? In this case, everyone sees two hats of the same color, so everyone speaks and says 'black', and everyone is wrong.

Table 1.1 looks at who speaks, and if they are correct or incorrect.

There are several remarkable facts that we can glean from this table. The first and most important, of course, is that the strategy is successful exactly three-fourths of the time. This is truly amazing. Each person has an equal chance of having a white or a black hat, yet somehow we manage to do better than 50%. How can this be?

The answer lies in the three columns saying whether or not each person is correct or incorrect. While the outcome column is very nice for our three people (saying they win 6 out of 8 times), it is the individual right-wrong columns that reveal what

			#1	#2	#3	#1	#2	#3	Outcome
W	W	W	black	black	black	wrong	wrong	wrong	lose
W	W	B			black			right	win
W	B	W		black			right		win
W	B	B	white			right			win
B	W	W	black			right			win
B	W	B		white			right		win
B	B	W			white			right	win
B	B	B	white	white	white	wrong	wrong	wrong	lose

Table 1.1 The various outcomes for our hat strategy. The first three columns are the hat colors of the three people, the next three columns are what each person says (if they remain silent, we leave it blank), the next three columns are whether or not a speaker is correct, and the final column is whether or not the players win or lose.

is really going on. Note each person is correct twice, incorrect twice, and silent four times. Thus, each person is only correctly saying their hat color half the time they speak (and only a quarter of the time overall). Notice, however, the sideways M pattern in who is correct and who is incorrect. We've somehow arranged it so that the wrong answers are piled up together, and the correct answers are widely separated. In other words, when we are wrong, boy are we wrong! All three people err. However, when we are right only one person is speaking. We thus take 6 correct and 6 incorrect answers and concentrate the incorrect answers together and spread out the correct answers.

The arguments above explain how it works, but it doesn't really say why it works. To understand the why, we have to delve a bit deeper, and it is this explanation that plays such a central role in error correcting codes. We have a space of eight possible hat assignments:

$$\{\text{WWW, WWB, WBW, WBB, BWW, BWB, BBW, BBB}\}.$$

Each assignment is equally likely; thus one-eighth of the time we have WWW, one-eighth of the time we have WWB, and so on. We partition our space into two disjoint subsets:

$$\{\textbf{WWW}, \text{ WWB, WBW, BWW}\}, \quad \{\text{WBB, BWB, BBW, } \textbf{BBB}\};$$

note we've flipped the fourth and fifth elements. What is so special about this partition is how the elements are related. In the first set, the second, third and fourth elements differ from the first element, which is WWW, in only one place. To put it another way, if we start with WWW we get any of the next three elements by changing *one and only one hat color*. Further, and this is the key point, the only way we can get something in the second set from WWW is to change *at least two hat colors*. We have a similar result for the second set. The first, second and third elements can be obtained from BBB by switching exactly one of their colors; further, nothing in the first set can be switched into BBB unless we change at least two hats.

This partitioning of the outcome space is at the heart of our solution. We have split the eight elements into two sets of four. The first set is either WWW or anything that can be made into WWW by switching exactly one hat. The second set is either BBB or anything that can be made into BBB by switching exactly one hat. Further, these two sets are disjoint – they have no elements in common, and thus they split our space into two equal groups. Later in this chapter we'll see how this partition can be used to build

1.2 DEFINITIONS AND SETUP

It's time to switch from our informal discussion to a more rigorous description of error detection and correction. To do so, we need a few definitions. We've repeatedly talked about codes in this chapter (error correcting codes, error detecting codes). What do we mean by this? A **code** is a collection of strings formed from a given **alphabet**. The alphabet might be the standard one for the English language, or it might be the binary set $\{0, 1\}$. If the alphabet has just two elements we refer to it as a **binary code**; if the alphabet has r elements we say we have an **r-ary code**. The elements of the code are called the **codewords**. If every codeword has the same length, we have a **fixed length code**.

In practice, here's what happens. We have some message or some data that we want to transmit. We choose a code, say C. We then use an encoding function to convert our data to a string of elements in the code C. We transmit that string, and then our compatriot on the other end converts the string back. We won't discuss in too much detail the process of encoding, as we've talk at great lengths about that in the cryptography chapters. Instead, here we're going to concentrate on the transmission of the encoded string. The issues of tantamount importance to us will be detecting transmission errors and, when possible, correcting them.

Let's look at some examples.

Example 1.2.1. *Take $A = \{a, b, c, \dots, x, y, z\}$ as our alphabet, and let C be the set of all finite strings of elements in A. Clearly C is a code; moreover, every English word is in C, as are nonsense words like qwerty and noitadnuof. It is not a fixed length code, as 'the' and 'bulldog' are both valid code words, and they have different lengths.*

Example 1.2.2. *Take $A = \{0, 1\}$ as our alphabet, and let C be the set of all strings of elements in A of length 4. We easily see that C is a code; further, it is a fixed length code, as each code word has length 4.*

Exercise 1.2.3. *Consider the two codes from the above examples.*

1. *For the code from Example 1.2.1, how many code words are there of length exactly 2? Of length at most two? If there are finitely many code words, how many code words are there?*

2. *For the code from Example 1.2.2, how many code words are there of length exactly 2? Of length at most two? If there are finitely many code words, how many code words are there?*

Exercise 1.2.4. *Consider the binary code*

$$\{0,\ 1,\ 10,\ 11,\ 100,\ 101,\ 110,\ 111\},$$

and say you receive the message 1101*. It's possible that this message is the codeword 110 followed by the codeword 1; it could also be the codeword 1 followed by the codeword 101, or 11 followed by 0 followed by 1. Could it be anything else, and if so, what?*

The last exercise illustrates a grave defect of some codes. In many cases, just because we receive a message does not mean we know what was sent! It's important to note that we are not talking about receiving an encrypted message and being unable to decrypt it. That's not the issue at all. The problem is we receive a collection of code words and we don't know how to parse them. In the last example, does 1101 mean the message 110 1, or does it mean 1 101, or perhaps even 11 0 1? There is unfortunately no way to know. Perhaps a more enlightening example would be to use letters and English words. Consider the string HITHERE. Should this be parsed as 'HI THERE' or 'HIT HERE'?

This problem cannot exist if all codewords have the same length. If we had a binary code where each codeword is of length 3, then the only way to parse 110101010 is as 110, 101, 010 (of course, 110, 101 and 010 should then be code words in our code!). It is because of this issue that we restrict our attention to fixed length codes.

Exercise 1.2.5. *Of course, a code can still have each message uniquely decipherable even if it isn't a fixed length code.*

1. *Imagine our code is the set*

$$\{1,\ 1001,\ 111000111000\}.$$

Is this code uniquely decipherable?

2. *Imagine our code is the set*

$$\{1,\ 10001,\ 111000111\}.$$

Is this code uniquely decipherable?

3. *Consider an r-ary code $C = \{c_1, c_2, \dots, c_n\}$ with the length of c_i equal to ℓ_i. McMillan proved that if the code C is uniquely decipherable, then*

$$\sum_{i=1}^{n} \frac{1}{r^{\ell_i}} \ \leq\ 1;$$

unfortunately, this sum can be finite without the code being uniquely decipherable. Consider the code C of all binary words of length at most 10 with an even number of 1s. Thus 100100011 is in C, as is 00100001*, but not* 1000000011*. Show C cannot be uniquely decipherable.* Hint: remember that 0001 is a code word of length 4, and is different from the code word 1.

From here on, we'll mostly concentrate on fixed length binary codes with alphabet $\{0, 1\}$. We need one last definition. The **Hamming distance** between two binary strings (both of the same length, say n) is the number of places where the two strings differ. We could denote this distance function by d_{Hamming}, but as we won't use any other distance functions, let's keep the notation simple and just write d for the distance function. The Hamming distance is always an integer, of course.

For example, imagine $n = 10$ and our two strings are 0011100101 and 0011100001. These two strings are almost identical; the only difference between them is in the eighth position. Thus we would write

$$d(0011100101, 0011100001) \ = \ 1.$$

A very important example is the binary code

$$\{111,\ 110,\ 101,\ 011,\ 100,\ 010,\ 001,\ 000\}.$$

We have

$$\begin{aligned} d(111, 111) \ &= \ 0 \\ d(111, 110) \ &= \ d(111, 101) \ = \ d(111, 011) \ = \ 1 \\ d(111, 100) \ &= \ d(111, 010) \ = \ d(111, 001) \ = \ 2 \\ d(111, 000) \ &= \ 3. \end{aligned} \tag{1.1}$$

If you view the two code words as points in n-dimensional space, the Hamming distance measures how far apart they are, given that we can only walk parallel to the coordinate axes. Thus $d((0, 0), (1, 1)) = 2$; we can either take the path

$$(0, 0) \ \longrightarrow \ (0, 1) \ \longrightarrow \ (1, 1)$$

or the path

$$(0, 0) \ \longrightarrow \ (1, 0) \ \longrightarrow \ (1, 1).$$

This is different than the normal distance between the two points, which is found using the Pythagorean theorem (in this case, it would be $\sqrt{(1 - 0)^2 + (1 - 0)^2} = \sqrt{2}$).

The **minimum distance of a code** C is the smallest distance between two distinct code words in C. We denote this by $d(C)$, and we may write it as

$$d(C) \ = \ \min_{\substack{w_1 \neq w_2 \\ w_1, w_2 \in C}} \ d(w_1, w_2).$$

If we didn't force the two code words to be distinct, then the minimum distance of any code would be zero, as $d(w, w) = 0$; thus this is a necessary constraint. The **maximum distance of a code** C is the maximum distance between two distinct code words.

Exercise 1.2.6. *Let C be the binary code of words of length 5 with an odd number of 1s. What is the minimum distance of the code?* Hint: the number of code words is $\binom{5}{1} + \binom{5}{3} + \binom{5}{5} = 5 + 10 + 1 \ = \ 16$. This is a small enough number to be manageable; in other words, you could write down all the different code words, but then you would have to look at all pairs of distinct code words, and there are $\binom{16}{2} = 120$ different pairs! Fortunately you don't have to investigate all of these pairs; you just have to find the minimum. Try proving that the minimum cannot equal 1.

Exercise 1.2.7. *Let C be the binary code of words of length 10 with exactly eight 1s. What is the minimum distance of the code? What is the maximum distance?* Hint: there are $\binom{10}{8} = 10!/8!2! = 45$ code words in C; you clearly want a faster way then enumerating all of these!

Exercise 1.2.8. *Let C be the binary code of words of length 10. What is the minimum distance of the code?*

We end this section with another definition that we'll need. We say a code C is **k-error detecting** if the following holds: no matter what code word of C we choose, if we change at most k digits then the resulting string is not a code word of C. If C is k-error detecting but not $(k+1)$-error detecting, then we say C is **exactly k-error detecting**. If a code cannot detect any errors, it is 0-error detecting.

Example 1.2.9. *Let's consider the binary code*

$$\{00,\ 01,\ 10,\ 11\}.$$

This code unfortunately cannot detect any errors. Why? We have a fixed length code (of length 2). There are four possible code words of length 2, and all of them are in our code. If we change any digit of any code word, we end up with another code word.

The above example shows us that, in order to be k-error detecting (for some $k \geq 1$), it is necessary that our code is only a subset of all possible words. If we have a binary code of fixed length n, then there are 2^n possible code words. Clearly if every word is in our code then we cannot detect any errors. The question becomes how many words can we include and have a 1-error detecting code, and which words can these be? We can then of course ask the same question for a 2-error detecting code, and a 3-error detecting code, and so on.

Exercise 1.2.10. *Let's consider the binary code*

$$\{00,\ 11\}.$$

Show this is a 1-error detecting code but not a 2-error detecting code, and thus is exactly a 1-error detecting code.

Exercise 1.2.11. *Consider the binary code*

$$\{000,\ 111\}.$$

Show this code is exactly 2-error detecting.

1.3 EXAMPLES OF ERROR DETECTING CODES

Remember we have two problems. The first is to construct an error detecting code, and the second is to construct an error correcting code. In this section we tackle the first problem, and give a variety of error detecting codes. Remember, our goal here is *not* to figure out what message was sent, but rather to determine whether or not there was a transmission error. This is a much easier problem. Most of the codes we'll see here are exactly 1-error detecting. It's not unreasonable to emphasize these codes. Hopefully the probability of making an error is small; if that is the case, it's unlikely we'll have two or more errors in our message.

In this section we'll briefly describe several different types of error detecting codes, and then discuss them in greater detail in the following section, both historically and mathematically. Another reason for this approach is that it often helps to see new material multiple times, and thus we'll give two slightly different presentations. Here we'll mostly state the various codes, while in the next section we'll dwell on them a bit more.

Example 1.3.1. *Let's imagine our code is*

$$C \; = \; \{1000,\ 0100,\ 0010,\ 0010,\ 1110,\ 1101,\ 1011,\ 0111\}; \qquad (1.2)$$

this is a fixed length binary code (the length is 4), made up of words with an odd number of 1s. The minimum distance of this code is 2. This can be seen by brute force computation, though that is unenlightening. A better way is to note that if we take any two words with exactly one 1 then they differ in two places, and similarly if we take two words with exactly three 1s. What happens if we take one of each? If the 1 from the code word with exactly one 1 aligns with one of the 1s from the word with exactly three 1s, then the 0 of the word with three 1s aligns with the 0, but the other two 1s do not, and thus the separation is 2; a similar argument gives 2 when the 1 does not align.

Imagine we receive the message 0101. *We were expecting a code word from C; it is impossible that this was the message. Why? Our message has to be one of the eight words in* (1.2)*; however, all of the code words have an odd number of 1s, and* 0101 *has an even number of 1s. There* must *have been an error!*

This is terrific – we know there was a mistake, and we can ask for the message to be resent. We unfortunately cannot tell where the error is; perhaps the message was supposed to be 0100, or perhaps it was meant to be 1101. We can't tell, but we do know an error was made.

Exercise 1.3.2. *Consider the code from* (1.2). *Imagine you receive the message*

$$1110110101001011.$$

Could this have been the intended message? What if you receive

$$1110101100110100.$$

Could that have been the intended message?

There are many other error detecting codes. Here is another.

Example 1.3.3 (Parity code)**.** *Let C is the binary code of all words of length 4 such that the sum of the digits of a code word is divisible by 2 (alternatively, we say the sum of the digits is congruent to 0 modulo 2). There are thus $2^4 = 16$ binary code words of length 4; half of these have digits summing to zero modulo 2. Why? We can choose the first three digits any way we wish, and then the last digit is forced* *on us. For example, if we have 101 then the final digit must be a 0, while if we have 010 the final digit must be a 1. There are* $2^3 = 8$ *ways to choose the first three digits, so there are 8 code words. Thus*

$$C = \{0000,\ 0011,\ 0101,\ 1001,\ 1100,\ 1010,\ 0110,\ 1111\}. \qquad (1.3)$$

If we receive the message 1011*, we know there was a transmission error as the digits do not sum to zero modulo 2 (i.e., there is not an even number of 1s). Going through all the cases, we see if we take any code word and change exactly one digit then it is no longer a code word. Thus, the code is 1-error detecting. It is not 2-error detecting. To see this, we can change 0011 to 0101 by changing the middle two digits, and this is a valid code word in* C*.*

Exercise 1.3.4. *Consider the code from* (1.3)*. Imagine you receive the message*

$$0011011011110111.$$

Could this have been the intended message? What about

$$11001010101010100011?$$

These two examples give us two error detecting codes; however, they seem very similar. Both involve codes with 8 words, and both are exactly 1-error detecting. Let's look at a really different example.

Example 1.3.5 (Fixed number code)**.** *We take our binary code* C *to be all binary words of length 5 with exactly two 1s:*

$$C = \{00011,\ 00101,\ 01001,\ 10001,\ 00110,\ 01010,\ 10010,\ 01100,\ 10100,\ 11000\}.$$

Note the minimum distance of this code, like the other two examples, is 2. If there ie exactly one error then we will detect it, as there will then be either three 1s or just one 1. If there are exactly two errors, we may or may not detect it. If the message was meant to be 00011 but we receive 01111, we know there was an error; however, if we receive 00101 then we would not realize an error occurred. This code was used by Bell Telephone Laboratories in the 1940s (see page 8 of [Th]), as the ten different code words could be set in correspondence with the ten different digits, and thus this gave a way of programming in the decimal system.

Exercise 1.3.6. *Using the code from Example 1.3.5, how many different messages of length 5 can be sent? If there is exactly one error, how many messages could be received?*

We have two candidates for an exactly 1-error detecting code, the code from Example 1.3.3 and the code from Example 1.3.5. Which is better? It's a little hard to compare the two codes. The first has code words of length 4, the second has code words of length 5. Both can detect exactly one error, but is it better to be able to detect one error in four digits or one error in five? We're comparing apples and oranges, and that's always a bit messy. We need to find a way to judge how good each code is.

One very good metric is to look at how much information each code allows us to convey. Let's look first at the code from Example 1.3.3. The code words there have length 4, with the fourth digit determined by the first three. Thus we have three free digits, and can effectively send any of 8 words: 000, 001, 010, 100, 101, 110, 011, 111. If we were to look at a string of length 20, we could send 5 code words. As each code word can be one of eight possibilities, this means we can send $8^5 = 32,768$ messages. What about the code from Example 1.3.5? There we have 10 possible code words of length 5. So, if we were to send a message of length 20, we could send 4 code words, giving us a total of $10^4 = 10,000$ messages.

It should now be clear why we looked at messages of length 20. This is the least common multiple of 4 (the length of the code words of the first code) and 5 (the length of the code words of the second code). The first code allows us to transmit almost three times as much information as the second. From this perspective, the first code is far superior. So, if all we care about is the amount of information transmitted, the first code is better; however, clearly this is not the only thing we care about. We are also concerned with detecting errors. In blocks of length 4, the first code can detect one error, while the second code can detect one error in blocks of length 5.

Comparing these two codes turned out to be easier than we thought – in each of the two natural metrics, the first code was clearly superior.

We end with one final example of an error detecting code.

Example 1.3.7 (Repetition code)**.** *This is the 'Tell me twice' code, where we take binary code of fixed length and replace each word with its double. For example, if we started off with the code*

$$\{00,\ 01,\ 10,\ 11\}$$

the new code becomes

$$\{0000,\ 0101,\ 1010,\ 1111\}.$$

It's very easy to detect exactly one error; if there is just one error, then the first two digits is not the same pair as the last two digits. Thus, this is a 1-error detecting code. It isn't a 2-error detecting code as 1010 could be the intended message, or it could have been 0000 corrupted in two places.

In Example 1.3.7, we have a four digit code with exactly four code words. Note this is much worse than Example 1.3.3. There we also had a four digit code, but we had 8 code words. Both codes can detect one error in a block of length 4, but the code from Example 1.3.3 can transmit twice as many messages per block of four.

1.4 COMPARISON OF CODES

Much of this section is inspired by the informative monograph by Thompson [Th]. In addition to describing the math, he discusses the history of the subject in great and often entertaining detail, which is a result of having interviewed many of the principal players.

Let's revisit some of the codes we discussed in the last section. To facilitate comparisons, we introduce one more definition. Imagine we have a binary code of fixed length n. We say the code is an (n, r) **binary block code** if the first r digits are free message digits and the last $n - r$ are check digits. What this means is that the first r digits can be either 0 or 1 without any constraints; there are 2^r possibilities here. After we have specified the first r digits, however, the last $n - r$ are forced on us by our choice of code. These are the check digits, and allow us to detect errors.

Perhaps the simplest example is the $(2n, n)$ binary block code that repeats a code word of length n; this is the Repetition Code of Example 1.3.7. What this means is that we consider the 2^n words of length n, and our code words are each of these words doubled. For example, if $n = 2$ we have four words of length $2^2 = 4$: 00, 01, 10, 11; our code words are 0000, 0101, 1010, 1111. This code easily detects one error; if only one digit is altered, the first half of the received code word does not match the second half of the code word. Unfortunately, this code cannot tell us what the error is. Moreover, it is very wasteful. *Half* of the digits are devoted to error checking. This is prohibitively expensive. Surely we can do better.

Let's return to the Parity Code of Example 1.3.3. In the language of this section, these are $(n + 1, n)$ binary block codes: we get to choose the first n digits freely, and then the final digit is forced upon us by the constraint that the sum of the digits is even (equivalently, there are an even number of 1s, or the sum of the digits is zero modulo 2). As discussed in the last section, it's easy to detect an error here. If $n = 2$ then we have the four code words 000, 011, 101, 110. Note that for this code, $\frac{n}{n+1}$ percent of the digits are devoted to transmitting the message, and only $\frac{1}{n+1}$ to error detection. Note the significantly greater efficiency here; as n gets very large almost all of the message is devoted to transmitting information, which is in stark contrast to the Repetition Code.

1.5 ERROR CORRECTING CODES

We now move to the exciting, long awaited topic of error correction. We'll describe some simple and easily implemented codes that not only detect errors, but actually correct them as well!

The simplest is an expanded form of the Repetition Code of Example 1.3.7. Instead of repeating the message once, let's repeat it twice. In other words, if we want to send one of 00, 01, 10, 11, then our code is

$$\{000000,\ 010101,\ 101010,\ 111111\}.$$

The idea behind this code is that, if mistakes are rare, the probability of two mistakes is very unlikely, and the majority opinion is probably correct. For example, imagine we receive the code word 101110. This is not in our code, and thus we know there has been a transmission error. If there can be only one digit error, then the original message must have been 101010; it is the only code word whose distance is at most 1 unit from 101110. All three of the blocks say the first digit of the message is 1; however, two of the blocks say the second digit is a 0 while one block says it is a 1. Going by the majority, we declare the fourth received digit to be in error, and the intended message to be 101010.

We've done it – we have a code that not only allows us to detect but also correct errors. Unfortunately, it is quite expensive. How costly is it? Well, in our fixed code of length 6 in the above example, only two digits were used for the message, and the other 4 were used for detection and correction. That means only one-third of the message is actually conveying information. Can we do better?

Let's return to the second riddle from §1.1. The key observation in our solution of the hat problem is that of the eight binary words of length 3, each word is either 000, 111, or differs in exactly one digit from one of 000 and 111. Let's consider a code based on this. We transmit either 000 or 111. Our received message is either one of these, or differs from one of these in exactly one digit. If our received message is not 000 or 111, we take whichever of these two is exactly one digit away. Note that this method is the same as repeat it twice.

While this actual code turns out to be a disappointment (it's the same as just the first digit is free, or again an efficiency of one-third), the idea of using the Hamming distance to partition the set of possible code words into disjoint subsets is very useful. There are entire courses devoted to this subject; we cannot begin to do it justice in a section or two, but we can at least give a flavor of what can be done. In this section we'll content ourselves with describing a very powerful error correcting code, the **Hamming** $(7, 4)$ **code**. Our exposition will be entirely unmotivated – we'll simply state what it is and see why it works. In the next section, we'll discuss how Hamming and others found this code.

The general idea is easy to state. Imagine we have a binary fixed length code C (of length n) whose minimum distance is d. This means our code words all have the same length (namely n), and are just strings of 0s and 1s. Further, given any two distinct code words, they differ in at least d places. Imagine our code C has the following wonderful property: each of the 2^n possible words is within a distance of $\lfloor \frac{d-1}{2} \rfloor$ of an element of C. Then our code C can detect and correct $\lfloor \frac{d-1}{2} \rfloor$ errors!

Where does the $\lfloor \frac{d-1}{2} \rfloor$ come from? For definiteness, imagine $d = 4$ and $n = 7$, and that we have a code C such that any two distinct code words in C differ by 4. Further, assume that each of the $2^7 = 128$ possible words of length 7 is a distance of at most $\lfloor \frac{d-1}{2} \rfloor$ from at least one of our code words. Could some word w be this close to two different code words? The answer is no because the distance function is transitive. Namely, if c_1 and c_2 are the two code words, then

$$d(c_1, c_2) \ \le\ d(c_1, w) + d(c_2, w) \ \le\ 2\lfloor \frac{d-1}{2} \rfloor \ \le\ d - 1;$$

however, we know the distance between any two distinct code words is at least d, and thus we have a contradiction. Thus each word is at most $\lfloor \frac{d-1}{2} \rfloor$ units from exactly one code word. This allows us to detect and correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors.

Exercise 1.5.1. *Check that the Hamming distance is transitive by comparing* $d(100110,$ $101101)$ *and* $d(100110, 100111) + d(100111, 101101)$. *Of course this isn't a proof of transitivity, but it is a good check.*

Exercise 1.5.2. *Complete the argument above by showing the Hamming distance* *is transitive.* Hint: first show it is enough to prove this in the special case when our words have length one!

We are now ready to give Hamming (7,4) code. The code has 7 binary digits, 4 are free message digits and 3 are check digits. The message digits are the third, fifth, sixth and seventh, and the check digits are the remaining three. The code C is the following 16 words:

> 1111111, 0010110, 1010101, 0111100, 0110011, 1011010, 0011001,
> 1110000, 0001111, 1100110, 0100101, 1001100, 1000011, 0101010,
> 1101001, 0000000.

A tedious calculation (there are better ways) shows that the minimum and the maximum distance of the code is 4; equivalently, this means each code word differs from any other code word in exactly 4 places!

Exercise 1.5.3. *There are 120 distances to check, as there are* $\binom{16}{2} = 120$ *ways to choose two out of 16 words when order does not count. Show* $d(0110011, 0100101) = 4$ *and* $d(1000011, 1101001) = 4$*. If you want to write a program to check all of these, the pseudocode looks like:*

```
Let Ham($i$) denote the $i^{th}$ of the 16 code words.
min = 7; max = 0.
For $i = 2$ to 7,
    For $j = 1$ to $i - 1$,
        Let $d = d($Ham($i$), Ham($j$)$)$,
        If $d <$ min then min = $d$.
        If $d >$ max then max = $d$.
Print min. Print max.
```

The Hamming $(7, 4)$ code has 16 code words. Each code word is exactly 4 units from any other code word. Thus $n = 7$ and $d = 4$, so $\lfloor \frac{d-1}{2} \rfloor = \lfloor \frac{3}{2} \rfloor = 1$, which means we can detect and correct one error. Another way of stating this is that each of the $2^7 = 128$ binary words of length 7 is at most 1 unit from a code word. To see this, note that each code word has seven neighbors that differ from it in just one digit; further, two different code words cannot share a neighbor that differs in just one place as this violates the code words being separated by 4. How many words have we accounted for? There are the original 16, and each generates seven more, for a total of $16 + 16 \cdot 7 = 128$. We have thus accounted for all of the 128 possible words!

We have therefore shown that the Hamming $(7, 4)$ code can correct one error. How efficient is it? How much information does it transmit? As we have 4 message digits and 3 check digits out of 7, four-sevenths or about 57% of the digits convey information. This is far superior to the Repetition Code, where only one-third of the digits were transmitting information. In fact, over half of our digits convey information, something that is of course impossible with a repetition code.

The Hamming $(7, 4)$ code has a lot of nice features. Our message is digits 3, 5, 6 and 7. We'll discuss in the next section why we take the check digits 1, 2 and

4 to be the values they are; for now, let's just accept these. There are three parts to using an error detecting code. The first is we have to encode our message, we then transmit our message, and then the recipient tries to decode the message. Our code tells us how each message is encoded. For example, while 1111 becomes 1111111, we have 1011 is sent to 0110011. We then transmit our message, and any single error can be detected and corrected. To decode our message, is it enough to just drop the first, second and fourth digits? Unfortunately, no; this works if there are no transmission errors, but of course the entire point is in correcting errors! Fortunately the decoding isn't too bad. So long as there is at most one error, only one of our 16 code words will be within 1 unit of the received message; whatever code word is this close is the decryption. Thus the Hamming $(7, 4)$ code has an extremely simple decryption routine; it's so easy we don't have to worry at all about any difficulties in implementing it.

Our purpose is to just show that efficient, easily implementable error correcting codes exist, and not to write a treatise on the subject. To do so would involve a lot more mathematics, involving group theory, sphere packings and lattices. One can build codes that can correct 2 errors, or 3, or 4 and so on. Stepping back, though, what we have just shown is already quite stunning: we can construct an easy to implement code that can detect and correct an error while still having more than half of its digits devoted to our message! Recasting this in terms of the Indiana Jones scene described earlier, it would be wonderful if Elsa could realize she misheard him and add the missing phrase 'You don't'.

1.6 MORE ON THE HAMMING $(7, 4)$ CODE

This section is heavily influenced by the Thompson's monograph [Th]; we heartily recommend it to anyone who would like to know a bit more about the characters involved in these stories, though be warned that the mathematics discussed is quite deep.

Before describing how Hamming arrived at this code, it's interesting to note why he was looking for it. In the 1940s Hamming was working at Bell Laboratories. He had access to computer time, but only on the weekends. The way it worked was his program was fed into the computer, which ran it until an error was found. At the first sign of an error, the program halted and the computer moved on to the next person's task. For two straight weekends, errors developed and Hamming was left with nothing. Frustrated, he wondered why couldn't the computer be taught to not just detect errors, but also correct them.

Example 1.6.1 (Square code)**.** *Hamming had a series of papers developing error correcting techniques. One fun example is a $(9, 4)$ binary block code. Consider a four digit number in binary, say $b_1b_2b_3b_4$. We write the number in a square:*

$$\begin{array}{cc} b_1 & b_2 \\ b_3 & b_4 \end{array}$$

and extend it to

$$\begin{array}{ccc} b_1 & b_2 & b_5 \\ b_3 & b_4 & b_6 \\ b_7 & b_8 & b_9 \end{array}$$

as follows: b_5 *equals* b_1+b_2 *modulo 2,* b_6 *equals* b_3+b_4 *modulo 2,* b_7 *equals* b_1+b_3 *modulo 2,* b_8 *equals* b_2+b+4 *modulo 2, and finally* b_9 *is* $b_1+b_2+b_3+b_4$ *modulo 2. We thus have four message digits and five check digits. It's a nice exercise to show that this can detect and correct any single error (in fact, we don't even need* b_9*). This code conveys more information than the repeating block code. While the 'tell me three times' or the 'majority rules' repeating code had one-third of its digits conveying information, here four-ninths or about 44% of the digits convey information (if we take the improved version where we remove the superfluous* b_9*, then half the digits are transmitting information). While this is worse than the Hamming* $(7,4)$ *code, it is a marked improvement. Note that this method is a natural outgrowth of parity checks.*

Exercise 1.6.2. *Given the message* 1011*, encode it using the algorithm above.*

Exercise 1.6.3. *List the 16 different messages in the code from Example 1.6.1.*

Exercise 1.6.4. *Assume you and your friend are using the method from Example 1.6.1, and you receive the message 101111000. You quickly realize this is not what your friend meant to send; assuming there was only one error in the transmission, what was the intended message?*

We now describe the derivation of the Hamming $(7,4)$ code. We have 7 digits at our disposal, and the goal is to devote four of them to our message and three of them to checks. We will use parity checks, which is very common when working with 0s and 1s. We're going to ignore the geometry and the distance metric, and concentrate on how the check digits are used. We have 16 possible messages, or equivalently 16 numbers.

We place our four digit message into slots 3, 5, 6 and 7 of our seven digit message. We now have to assign values to the first, second and fourth digits. Let's write our seven digit message as $d_1d_2d_3d_4d_5d_6d_7$, where we know d_3, d_5, d_6 and d_7 and need to determine d_1, d_2 and d_4. We set

$$\begin{aligned} d_1 &= d_3+d_5+d_7 \bmod 2 \\ d_2 &= d_3+d_6+d_7 \bmod 2 \\ d_4 &= d_5+d_6+d_7 \bmod 2. \end{aligned} \tag{1.4}$$

Exercise 1.6.5. *Assume we are using the Hamming* $(7,4)$ *code, and we receive the message 0011001, which is not one of our sixteen code words. What message was meant? Which parity checks fail?*

Why does this method work? We have three parity checks. The first involves (d_3, d_5, d_7), the second (d_3, d_6, d_7) and the last (d_5, d_6, d_7). We assume there is at most one error. If the message is transmitted correctly, then (1.4) is satisfied. If there is an error, then at least one of the three equations i (1.4) cannot hold. We now explore all the various possibilities. If all three parity checks fail, then d_7 was in error.

If only the first fails, then d_1 (one of our check digits!) is wrong. To see this, note that if the second two equations hold, then $d_2, d_3, \ldots, d_7$ must all be correct, and therefore the cause of the error must be d_1.

If the first two equations fail, then the fact that the last holds means d_4, d_5, d_6 and d_7 are correct. The only element in common between the first two equations is d_3, which therefore must be in error.

We end with a few words on how one can find these parity checks; these arguments can easily be skipped. We have 16 code words; we list them, and give their equivalent decimal number.

0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000 (8)
1001 (9)
1010 (10)
1011 (11)
1100 (12)
1101 (13)
1110 (14)
1111 (15)
10000 (16)

We can split these 16 numbers into three sets, depending on the value of their ones digit, their twos digit (remember we're working in binary, or base 2; thus 10 is really the same as 2), and their fours digit (100 is really 4). The numbers that have a 1 as their ones digit are 1, 3, 5, 7, 9, 11, 13 and 15, those with a 1 as their twos digit are 2, 3, 6, 7, 10, 11, 14 and 15, and those with a 1 as their fours digit are 4, 5, 6, 7, 12, 13, 14 and 15. Note each check has exactly eight of our sixteen numbers associated to it, and each of the sixteen numbers appears in zero, one, two or three of the checks.

1.7 ADDITIONAL RIDDLE

HAVEN'T INCORPORATED THIS YET, BUT THE HAMMING (7,4) CODE CAN BE MOTIVATED BY THE FOLLOWING RIDDLE. I'LL ADD THIS SOON.

The King of Freedonia has caught 10 spies from the Kingdom of Sylvania who attempted to poison his wine. The king keeps 1000 bottles of wine in his cellar. The spies managed to poison exactly one bottle but were caught before they could poison any more. The poison is a very special one that is deadly even at one-trillionth the dilution of the poisoned wine bottle and takes EXACTLY 24 hours to kill the victim, producing no symptoms before death.

The trouble is that the King doesn't know which bottle has been poisoned and the wine is needed for the Royal Ball in exactly 24 hour's time! Since the punishment for attempted regicide is death, the king decides to feed some of the wine to the spies. The King informs his wine steward that if he mixes wine from appropriate bottles for each spy, he will be able to identify the poisoned bottle by the time the ball starts and kill at most 10 of the spies. Further, each spy drinks only once.

How does he do it, and can one guarantee that at most 9 (or even at most 8) are killed?

Chapter Two

Primality Testing and Factorization

NOTE: THIS INTRO WAS ORIGINALLY WRITTEN FOR SOMETHING ELSE, AND MIGHT NEED TO BE MOVED ELSEWHERE IN THE BOOK.

It's June 6, 1944, and you are an allied commander during D-Day, when Allied forces launch the greatest amphibious landings the world had seen in an attempt to liberate Europe. You lead your troops ashore, and find the disposition of German forces is not what headquarters predicted; if you can alert command in time, you can have your reinforcements land at strategic positions. Of course, if the Germans learn where future forces will attack, they'll try and shift their reserve troops to thwart those landings.

This example illustrates two of the key issues in cryptography. First, we need to be able to encode and decode messages quickly, ideally in real time. Battlefield conditions change minute by minute, if not faster; if it takes too long to send and receive secure communications, the information or requests could be outdated. Second, the code must be secure. The level of security needed depends on the situation. For communications before D-Day that describe the landings, these codes must be very secure, requiring at a minimum weeks to decrypt. The reason is that an enormous part of D-Day is the surprise, specifically when and where it will occur. For communications during the battle, however, far less security is needed. There it's fine to use a code that can be broken in several hours, but not sooner, for battlefield requests. For example, if troops radio headquarters at noon that future waves should land at a specified location, it suffices for the code to be secure until those landings have taken place and been observed by the enemy.

We have discussed many different encryption and decryption schemes. Many of these use prime numbers. For example, in RSA we choose two large primes p and q and make public their product. This example illustrates two complementary issues for us: the need for efficient algorithms for primality testing and for factorization. We need to *find* prime numbers in order to use RSA, which takes two primes as an input. Additionally, the security of RSA is related to the assumed difficulty of *factoring* numbers into primes.

Obviously these two problems are related. If we can efficiently factor any number N then we get for free a primality test (simply see whether or not any factor is strictly between 1 and N); the converse, however, need not be true. That is to say, below we will see examples of algorithms that will say whether or a number N is prime, but if N is composite give us *no* information on its factors! At first this seems absurd. Every integer is either a prime or a product of primes; thus if our number isn't prime surely we must know a divisor! Sadly, this is not the case. It turns out that there are certain properties that prime numbers satisfy that composite numbers do not, and thus we can learn that a number is composite without actually knowing any factors!

We have been in this situation before. Let's revisit Euclid's proof of the infinitude of primes. We assumed for the sake of a contradiction that there are only finitely many primes, say 2, 3, 5, 7, 11, $\dots$, p_n. We then considered $2 \cdot 3 \cdot 5 \cdots 7 \cdots 11 \cdots p_n + 1$, and noted that this number cannot be divisible by any prime on our list, as each leaves a remainder of 1. Thus our new number is either prime or else it is divisible by a new prime not in our list. While we have shown there exist infinitely many primes, our proof does not give an infinite list of primes. The situation is similar for many primality tests, where we can learn a number is composite without knowing any of its factors.

In this chapter we'll explore how long it takes to do various mathematical tasks, concentrating on the importance of efficiency for cryptographic applications.

2.1 BRUTE FORCE APPROACH

WHEN DO WE WANT TO ADD SOMETHING ABOUT THE FUNDAMENTAL THEOREM OF ARITHMETIC, WHICH ASSERTS THAT THERE IS ALWAYS A FACTORIZATION, AND MOREOVER THAT IT IS UNIQUE?

Our first method is probably the oldest known way to factor a number: brute force. While many problems in cryptography and other sciences can be solved by brute force, frequently the amount of time required is so large that such methods are essentially useless. Prime numbers play a central role in many cryptographic systems. The security of RSA, for example, rests on the assumption that it is difficulty to *quickly* factorize large numbers.

The key word, of course, is *quickly*, as there is a trivial algorithm to factor any number. To find the prime factorization of N all we need do is try every number at most N. For example, consider $N = 1776$. We first try 2, which is a factor as

$$1776 \ = \ 888 \cdot 2.$$

We can now conclude that 1776 is not prime, though we do not have its complete factorization. To complete the factorization of 1776, we need only factor 888 and then add in the factor of 2 we have just found. Note that, once *one* factor is found, we have a new factorization problem but with a smaller number. We find that 2 divides 888, so 2^2 divides our original number 1776. Continuing in this manner, we see that 2^4 divides 1776 ($1776 = 16 \cdot 111$) but 2^5 does not, as

$$\frac{1776}{2^5} \ = \ 111\frac{1}{2}.$$

Thus

$$1776 \ = \ 2^4 \cdot 111.$$

We have removed all the powers of 2 from our number, and now proceed to factor 111. Note that this problem is easier than the original problem of factoring 1776 for two reasons. First, our number is smaller, and second, we know that 2 cannot be a factor, as we have removed all powers of 2. We thus try the next number, 3, and find

$$111 \; = \; 3 \cdot 37.$$

As 3 does not divide 37, we have found all the powers of 3 that divide 111, and thus all the powers that divide 1776. Our factorization has improved to

$$1776 \; = \; 2^4 \cdot 3 \cdot 37.$$

We now try to factor 37. We do not need to try 2 or 3 as a factor, as we have already removed all of these. What should our next attempt be? There are two ways to proceed. One possibility is to try 4. The reason it is natural to try 4 is that it is the next integer after 3; however, clearly 4 cannot be a factor as 4 is just 2^2, and we have already removed all powers of 2. Thus the next number we try should be the next prime after 3, which is 5. Trying 5, we see that it is not a factor of 37 as

$$\frac{37}{5} \; = \; 7\frac{2}{5}.$$

Continuing in this manner, we keep trying all the prime numbers up to 37, and find that none of them work. Thus 37 is prime, and the complete factorization of 1776 into products of prime powers is

$$1776 \; = \; 2^4 \cdot 3 \cdot 37.$$

Exercise 2.1.1. *Using the brute force algorithm, determine whether or not 1701 is prime. If it is not prime, factor it.*

From a theoretical point of view, the brute force algorithm is perfect. It is guaranteed to work. It will tell if a number is prime, and if it is composite it will provide a complete factorization. Unfortunately, it is essentially worthless for real world applications due to its incredibly long run-time. To see this, consider the case when N is prime. If we use the naive approach of trying all numbers, we have N numbers to try; if we instead only try prime numbers as potential factors, by the Prime Number Theorem we have essentially $N/\ln N$ numbers to check. For example, if $N = 10^{100}$ then $\ln 10^{100}$ is about 230 (remember that we are taking the natural logarithm). This means that there are about $10^{100}/230$ primes to check, or more than 10^{97} candidate factors!

Can the brute force approach be salvaged? With a little thought, we can improve our algorithm enormously. We've already seen that instead of checking every number, clearly we need only look at prime numbers up to N. A much better observation is to note that it also suffices to just test primes up to $\sqrt{N}$. The reason is that if $N = ab$ then either a or b must be at most $\sqrt{N}$. If both a and b were larger than $\sqrt{N}$ then the product would exceed N:

$$\begin{aligned} a \; &> \; \sqrt{N} \\ b \; &> \; \sqrt{N} \\ a \cdot b \; &> \; \sqrt{N} \cdot \sqrt{N} \; = \; N. \end{aligned}$$

Thus for a given number N, either it is prime *or* it has a prime factor at most N.

For example, consider $N = 24611$. As $\sqrt{N} \approx 156.879$, this means that either N is prime *or* it has a prime factor at most 156. Checking all primes up to 156 (the largest such prime is 151), we see that none of them divide 24611. Thus 24611 is prime, and we can prove this by confining our search to primes at most 156. There are only 36 such primes, which is far fewer than all the primes at most 24611 (the largest is 24593, and there are 2726 such primes).

While it is an enormous savings to only have to check candidates up to $\sqrt{N}$, is it enough of a savings so that brute force becomes useful in practice? Let's analyze the problem in greater detail. Consider the situation of RSA, where an eavesdropper needs to factor a large number N which is the product of two primes. Let's say the primes p and q are around 10^{200} so that N is around 10^{400}. How long would it take us to try all possible divisors by brute force? The universe is approximately 13 billion years old, and is believed to contain less than 10^{90} subatomic items. Let's overestimate a lot, and say the universe is 1 trillion years old and there are 10^{100} subatomic items, each of which is a supercomputer devoted entirely to our needs and capable of checking 10^{40} prime numbers per second (note this is *magnitudes* beyond what the current best computers are capable of doing). How long would it take us to check the 10^{200} potential prime divisors of N? We first translate our age into seconds:

$$\text{1 trillion years} \ = \ 10^{12} \text{ years} \cdot \frac{365.25 \text{ days}}{1 \text{ year}} \cdot \frac{24 \text{ hours}}{1 \text{ day}} \cdot \frac{3600 \text{ seconds}}{1\text{hour}} \ < \ 10^{20}.$$

The number of seconds needed is

$$\frac{10^{200}}{10^{100} \cdot 10^{40}} \ = \ 10^{60};$$

the universe hasn't even existed for 10^{20} seconds, let alone 10^{60}!

The point of the above computation is to drive home the point that, just because we have an algorithm to compute a desired quantity, it does not mean that algorithm is useful. In our investigations in cryptography, we will encounter two variants of this problem. The first is a search for efficient algorithms, so we do not have to wait significantly longer than the universe has existed for our answer! The second is the opposite; for security purposes, we *want* problems whose answers take an incredibly long time to solve, as these should be secure. Of course, it is often necessary for us to solve these problems as well. We will thus be led to studying trapdoor functions and problems. These are problems where the answer can be obtained significantly faster with an extra piece of information that is not publicly available. For instance, let's return to the factorization problem, and let's assume that the password is the largest prime factor of N, where $N = pq$ is the product of two rimes. We've seen that if N is of the order 10^{400} then it takes too long to compute the prime divisors by brute force; however, if we are given p, then we can immediately determine q from the simple division $q = N/p$.

Exercise 2.1.2. *In the example above, we assumed the amount of time it took to test whether or not a given n divided N was independent of n. In practice this is of course absurd.* ***add stuff on bit operations.***

One of the greatest dangers in cryptography, of course, is that a problem which is believed to be difficult might in fact be easy. For example, just because no one knows a fast way to factorize large numbers does not mean no such method exists, it just means we do not know how to do it. It's possible someone has discovered a fast way to factor large numbers, and is keeping it secret precisely for these reasons! In cryptography, the fear is that there might be a hidden symmetry or structure that an attacker could exploit, and the underlying problem might not be as difficult as we thought. Let's revisit our factorization problem. Recall the goal is to find the factors of N, with N a product of two primes. If we don't know either prime, we saw that even if we had incredible computer resources at our disposal, the life of the universe isn't enough time to make a dent in the problem. If we know the smaller of the two factors, the problem is almost instantaneous. What if we are foolish, and accidentally choose the two primes to be equal? In this case, the problem is again trivial. Even though we don't know p, we can quickly discover it by noting that $\sqrt{N}$ is an integer. (There are lots of good ways to take square-roots; we could always approximate $\sqrt{N}$, and then test the two integers immediately above and below.)

We now discuss various algorithms for primality testing and factorization. Some of these work only for numbers of certain, special form; others work for any input. While many of these algorithms are significantly faster than brute force, to date there is no known, fast way to factor a number, though there are known, fast ways to determine if a number is prime.

2.2 FERMAT'S FACTORING METHOD

As remarked above, one of the central difficulties of cryptography is that much of it is based on the belief that certain problems are hard and thus cannot be solved without extra information that is hidden from an attacker. How do we know these problems are hard? We know because no one has solved them yet! Of course, this sounds like circular logic. There could be an elementary approach that has just been missed for years, and the problem might actually be easy after all. In fact, we'll see an example of this in §2.3.

We now describe some interesting approaches to factorizing numbers. The first is Fermat's method. We might as well assume our integer N is odd, as it is very easy to check whether or not it is divisible by 2. Further, we can assume that N is not a perfect square, as it is very easy to test if $N = n^2$ for some n. To do this, we simply approximate the square-root of N and then check the integers immediately below and above to see if either squares to N.

We therefore content ourselves with trying to factor odd N that are not perfect squares. Imagine that we can write N as the difference of two squares, so

$$N \;=\; x^2 - y^2.$$

If, somehow, we can find such an x and a y, then we can factor N as

$$N \;=\; (x-y)(x+y);$$

so long as $x - y$ and $x + y$ are neither 1 nor N we have factored N as a product of two non-trivial terms. For example,

$$2007 \;=\; 116^2 - 107^2,$$

so

$$2007 \;=\; (116+107)(116-107) \;=\; 223 \cdot 9.$$

For a more interesting example, consider

$$12069 \;=\; 115^2 - 34^2 \;=\; 149 \cdot 81,$$

or even more amazingly

$$\begin{aligned} 123456789987654321 &= 355218855^2 - 52188552^2 \\ &= 407407407 \cdot 303030303; \end{aligned}$$

we couldn't use 12345678987654321 as an example because of the entertaining fact that it equals 111111111^2.

Several questions immediately present themselves.

- If N is an odd, composite number, can N be written as a difference of two squares?
- If N can be written as a difference of two squares, how do we find these squares?

We start with the first question. It turns out that it is always possible to write an odd, square-free N as a difference of two squares:

$$\left(\frac{N+1}{2}\right)^2 - \left(\frac{N-1}{2}\right)^2 \;=\; \frac{N^2+2N+1}{4} - \frac{N^2-2N+1}{4} \;=\; N.$$

How did we arrive at this? This comes from what happens when our factorization is trivial. Explicitly, if

$$N \;=\; x^2 - y^2 \;=\; (x-y)(x+y),$$

let's consider what happens when $x - y = 1$. As the product is N, this forces $x + y$ to equal N. Adding these two equations give

$$\begin{aligned} x + y &= N \\ +\,\underline{x - y} &= \underline{1\quad} \\ 2x + 0 &= N + 1, \end{aligned}$$

or $x = \frac{N+1}{2}$. If instead we subtract these two equations we find $2y = N - 1$ or $y = \frac{N-1}{2}$. Thus the difference of squares is really

$$x^2 - y^2 \;=\; \left(\frac{N+1}{2}\right)^2 - \left(\frac{N-1}{2}\right)^2 \;=\; \frac{N^2+2N+1}{4} - \frac{1-2N+N^2}{4} \;=\; N.$$

We see that our first question was poorly phrased. It is not enough to write N as a difference of two squares; we want the factors $x-y$ and $x+y$ to be neither 1 nor N.

Clearly this cannot be possible if N is prime. What if N is composite, say $N = rs$ (where of course r and s are odd as N is odd)? We then need $(x-y)(x+y) = rs$. If we let $x - y = r$ and $x + y = s$, then adding the two gives $2x = r + s$ or $x = \frac{r+s}{2}$; note x will be an integer since r and s are odd so their sum is even. Similarly, subtracting the two equations gives $2y = s - r$ so $y = \frac{s-r}{2}$. In other words, if $N = rs$ then we have

$$N \;=\; \left(\frac{r+s}{2}\right)^2 - \left(\frac{s-r}{2}\right)^2.$$

Our discussion above gives us some cause for hope. We have shown that if N is composite then we can write it as a difference of two squares in a non-trivial way, and conversely if we can write it as a difference of two squares then we can obtain a factorization.

The next question is whether or not we can *find* a difference of two squares equaling N. At first glance, there is a very obvious reason to worry about the practicality of Fermat's method: what if $N = x^2 - y^2$ but x and y are much larger than N? If this were the case then it might take a long time to find the factors. We clearly need some control over how large x and y can be in order to bound how long Fermat's algorithm will take. This is readily done by noting again that $N = x^2 - y^2 = (x-y)(x+y)$. For a non-trivial factorization, $x - y \geq 2$ and thus $x + y \leq N/2$ (as $x + y = \frac{N}{x-y}$). In other words, x and y are at most $N/2$. Compared to our original attempt to factor by brute force, this is the same order of magnitude of how many numbers we must check; however, it is much worse than our refined brute force approach. There we saw it sufficed to check all numbers up to $\sqrt{N}$, whereas here we are still checking on the order of N numbers.

We can now explicitly state Fermat's method to factor N.

Fermat's Method

1. If N is even, then 2 is a factor and apply Fermat's method to $N/2$.
2. If N is a perfect square, then apply Fermat's method to $\sqrt{N}$.
3. For N an odd integer which is not a perfect square, let s equal the smallest integer greater than $\sqrt{N}$. Let $x = s, s+1, \dots, \frac{N-3}{2}$. If for any of these choices of x we have $x^2 - N$ is a square (say y^2), then N is composite and factors as $N = (x-y)(x+y)$; if for each of these x's the number $x^2 - N$ is not a square, then N is prime.

While we know that it suffices to check x and y that are at most $N/2$, if we had to check all such integers then Fermat's method would be too slow to be of practical use. Unfortunately, there are times when we would have to check all such numbers, for example, if N were prime. Fortunately, there is a large class of numbers where Fermat's method works quite well, namely those N which are the product of two primes each of which is near $\sqrt{N}$.

For example, consider $N = 327,653$. N is odd, and as $\sqrt{N}$ is approximately 572.41 we see N is not a perfect square. Letting s be the smallest integer at least $\sqrt{N}$, we have $s = 573$. Thus we must see if any of $s^2 - N$, $(s+1)^2 - N, \dots$ are perfect squares. We are in luck, as the very first case is

$$s^2 - N \ = \ 573^2 - 327653 \ = \ 328329 - 327653 \ = \ 676 \ = \ 26^2.$$

Rearranging gives

$$573^2 - 26^2 \ = \ (573-26)(573+26) \ = \ 547 \cdot 599.$$

For a more interesting example, consider $N = 223,822,733$. A straightforward computation shows that N is odd and not a perfect square. As $\sqrt{N} \approx 14960.7$, $s = 14961$. Thus we must check to see if any of $s^2 - N$, $(s+1)^2 - N$, and so on are perfect squares. We have

$$\begin{array}{rcrcr}
14961^2 - N & = & 8788 & \approx & 93.74^2 \\
14962^2 - N & = & 38711 & \approx & 196.75^2 \\
14963^2 - N & = & 68636 & \approx & 261.99^2 \\
14964^2 - N & = & 98563 & \approx & 313.95^2 \\
14965^2 - N & = & 128492 & \approx & 358.46^2 \\
14966^2 - N & = & 158423 & \approx & 398.02^2 \\
14967^2 - N & = & 188356 & = & 434^2.
\end{array}$$

Thus

$$N \ = \ 14967^2 - 434^2 \ = \ (14967+434)(14967-434) \ = \ 15401 \cdot 14533;$$

thought it is not obvious at all, the two factors above happen to be prime, and we have thus factorized $N = 223,822,733$. While Fermat's method does not work well for all numbers, it is amazing how well it captures the factors of N whenever we can write N as rs with r, s of size $\sqrt{N}$. We were able to show a nine digit number is composite by just doing 7 loops through the algorithm.

2.3 AGRAWAL–KAYAL–SAXENA PRIMALITY TEST

There are many algorithms used to factor integers or, if we set our goals lower, to simply determine if a number is prime. We need a way to compare these algorithms; however, it is not immediately clear how to compare different methods. For example, what if most of the time one method runs 10 times faster than another, but for some special numbers the method breaks down and the program never terminates? Which algorithm is 'faster'? Note that if the algorithm doesn't terminate for some inputs, then its average run time will be infinite!

Let's consider one method that takes 10 seconds for each input, and another which takes 1 second for all inputs that are not five more than a multiple of 10^{100}, where for these numbers the run-time equals 10^{1000} seconds. The average run-time of the first method is just 10 seconds, while for the second it is

$$\frac{(10^{100}-1)\cdot 1 + 1 \cdot 10^{1000}}{10^{100}} \ = \ 1 - 10^{-100} + 10^{10},$$

or approximately 10^{10} seconds (which is more than 300 years!). Which algorithm is better? Even though the first has a better average run-time, most people would prefer the second algorithm as *most* of the time it is ten times faster. Of course, this is a very artificial example in that we know precisely which inputs are bad for the second algorithm. Knowing this, we would clearly use the first algorithm for such special numbers and our second algorithm otherwise. In fact, we can generalize this idea and look at a combined, new method defined as follows:

- Step 1: Run the second method on our input. If it terminates in a second, we are done; if not, proceed to Step 2.
- Step 2: Run the first method on our input.

By mixing the two methods, we are able to take advantage of each in the situations where they run well. If the second method worked on our input (which happens for $10^{100} - 1$ out of every 10^{100} numbers) then our run-time is just 1 second; if we had to go to Step 2 then the total run-time is 11 seconds (one second for Step 1 and then 10 seconds for Step 2). Hence the average run-time is now

$$\frac{(10^{100} - 1) \cdot 1 + 1 \cdot 11}{10^{100}} \ = \ 1 + 10^{-99},$$

or just a tad over 1 second.

This illustrates a common feature in factorization. We can try certain methods that work well for restricted inputs, and if they do not work we can then proceed to a slower algorithm that works well for more inputs.

For a long time, one of the biggest problems in cryptography was whether or not there was a provably fast method to determine if a number N is prime. By 'fast' it was meant an algorithm that runs in time proportional to the number of digits of N to a fixed power. Note that $\log_{10} N$ is a good measure of how many decimal digits N has. For example, $\log_{10} 10^8 = 8$, $\log_{10} 10^9 = 9$, and if $10^8 < N < 10^9$ then $8 < \log_{10} N < 9$. People were thus searching for an algorithm which would tell whether or not an integer N was prime whose run-time was bounded by a polynomial in $\log_{10} N$. To get a sense of what this means, assume the run-time to test the primality of N is $4(\log_{10} N)^7 - 2\pi(\log_{10} N)^2 + 11$. If we now consider $N_2 = N^2$, then the run-time is increased by approximately a factor of $2^7 = 128$, while if we take $N_3 = N^3$ the run-time is approximately $3^7 = 2187$ times that for N. For example, if we take $N = 123456789$ then the run-time for N is about $9.08343 \cdot 10^6$ seconds and for N^2 it is approximately $1.16273 \cdot 10^9$ seconds, which is an increase in run-time of about 128.005.

Before 2002, there were no algorithms which were provably fast to determine whether or not a number was prime. There were numerous algorithms, but they were either probabilistic (and thus for some inputs it could take a long time to terminate) or rested on well-believed but unproven conjectures such as the Riemann Hypothesis. The situation dramatically changed in August, 2002, when Professor

Manindra Agrawal and his students Nitin Saxena and Neeraj Kayal at the Indian Institute of Technology Kanpur announced a provably fast, deterministic algorithm to determine whether or not an integer is prime. This was a phenomenal result, and a major theoretical breakthrough in a centuries old problem.

It is worth noting that, in practice, this algorithm has not replaced other methods. The reason is that there are a large class of methods that are probabilistic; they will always return the right answer, but on some inputs the run-time could be enormous. As these methods are faster for most inputs, it makes sense to try one of these other algorithms first, and only if the program is taking a long time to switch to this new algorithm.

We describe their algorithm, called the AKS primality test, below. Not surprisingly, their announcement sent waves through scientific circles, and many people looked at their method and proof. This massive feedback has led to refinements and variants; we describe essentially the original formulation.

Before we can state their algorithm, we need to define three concepts and state one result from combinatorics. The first is the Euler totient function, the second is modular arithmetic, and the third is the order of an element, and the result concerns values of the binomial coefficients $\binom{n}{k}$.

We start with the simplest, the Euler totient function. Denoted φ, we define it by setting $\varphi(n)$ equal to the number of integers in $\{1, 2, \dots, n\}$ that are relatively prime to n. In other words, we are counting how many of these integers do not share any prime factors with n. We thus have $\varphi(1) = 1$, $\varphi(2) = 1$, $\varphi(3) = 2$ $\varphi(4) = 2$ and $\varphi(12) = 4$. For example, if $n = 12$ then we have the numbers

$$\{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12\}.$$

Which of these are relatively prime to 12? We lose 2, 3, 4, 6, 8, 10 and 12, and are thus left with

$$\{1,\ 5,\ 7,\ 11\},$$

and hence $\varphi(12) = 4$.

The totient function is very well understood; for example, if p is prime then $\varphi(p) = p - 1$. To see this, consider the set

$$\{1,\ 2,\ 3,\ \dots,\ p-2,\ p-1,\ p\}.$$

We want to know how many of these numbers do not share a proper divisor with p. Since p is prime, the only numbers that divide p are 1 and p; thus p cannot divide 1, 2, ..., $p-1$ but does divide p. We therefore see that exactly $p-1$ of the p numbers are relatively prime to p, so $\varphi(p) = p - 1$.

While $\varphi(n)$ is the number of integers in $\{1, \dots, n\}$ that are relatively prime to n, frequently it matters *which* integers are relatively prime. These integers have special properties relative to n, and thus merit a name. We write $(\mathbb{Z}/n\mathbb{Z})^*$ for the subset of integers of $\{1, \dots, n\}$ that are relatively prime to n. For example,

$$(\mathbb{Z}/12\mathbb{Z})^* \ = \ \{1,\ 5,\ 7,\ 11\}$$

and

$$(\mathbb{Z}/p\mathbb{Z})^* \ = \ \{1,\ 2,\ \dots,\ p-1\}$$

for any prime p.

Exercise 2.3.1. *Compute $\varphi(15)$, $\varphi(21)$, $\varphi(33)$ and $\varphi(35)$.*

Exercise 2.3.2. *Notice each of the numbers in the previous problem is the product of two primes. Find a pattern between $\varphi(pq)$, $\varphi(p)$ and $\varphi(q)$ for the four numbers of the previous problem? Based on your success, conjecture a formula for $\varphi(n)$ when n is the product of two primes. Prove your claim.*

Exercise 2.3.3. *Compute $\varphi(4)$, $\varphi(9)$, $\varphi(25)$ and $\varphi(49)$.*

Exercise 2.3.4. *Based on the results from the previous exercise, guess a formula for $\varphi(p^2)$ where p is a prime. Prove your result.*

Exercise 2.3.5. *Find a formula for $\varphi(p^3)$ for p prime by trying out various p. Prove your claim.*

The next needed input is modular arithmetic. We say $x \equiv y \bmod n$ if $x - y$ is a multiple of n. Thus $15 \equiv 3 \bmod 12$ as $15 - 3 = 1 \cdot 12$, and $-24 \equiv 4 \bmod 7$ as $-24 - 4 = 4 \cdot 7$. We can define addition and multiplication for elements of $(\mathbb{Z}/n\mathbb{Z})^*$ by performing normal addition and multiplication, and then looking at the result modulo n.

For example, $10 + 5$ equals 3 modulo 12. At first glance this seems absurd, as 10+5 is 15; however, if we look at it modulo 12 we are saying we are only concerned with the answer after removing as many multiples of 12 as we can. Similarly $10 \cdot 5$ equals 2 modulo 12. These operations are usually called modular arithmetic, but they are often called clock arithmetic in homage to the fact that most of us learned this years ago when we were taught how to tell time. For example, if it is 10 o'clock now then in 5 hours it will be 3 o'clock; this does not seem strange, and in fact doesn't even merit a pause anymore. What we are doing above is just the natural generalization, where instead of having a clock with 12 hours we now have a clock with n hours.

We can generalize these concepts from integers to polynomials. We say $f(x) \equiv g(x) \bmod m(x)$ if there is a polynomial $h(x)$ such that $f(x) - g(x) = h(x)m(x)$. For example,

$$3x^2 + 7x + 4 \ \equiv \ x^2 + 2x + 1 \bmod x + 1$$

as

$$(3x^2 + 7x + 4) - (x^2 + 2x + 1) \ = \ (2x + 3)(x + 1).$$

We can even combine polynomial congruence and modular congruence, and we say $f(x) \equiv g(x) \bmod (n, m(x))$ if there is an $h(x)$ such that $f(x) - g(x) - h(x)m(x) \equiv 0 \bmod n$. Obviously the more congruences we have, the harder it is. As an example, we show $9x^2 - 3x + 1 \equiv x \bmod (7, x + 1)$. We have

$$(9x^2 - 3x + 1) - x \ = \ 9x^2 - 4x + 1.$$

Because we will eventually look at everything modulo 7, we write $9x^2$ as $2x^2 + 7x^2$ and $-4x$ as $3x - 7x$. We thus have

$$(9x^2 - 3x + 1) - x \ = \ 9x^2 - 4x + 1 \ = \ (2x^2 + 3x + 1) + (7x^2 - 7x).$$

We can factor $2x^2 + 3x + 1$ as $(2x+1)(x+1)$; note $x+1$ is our modulus $m(x)$! We have thus shown

$$(9x^2 - 3x + 1) - x \;=\; (2x+1)(x+1) + 7(x^2 - x),$$

or

$$(9x^2 - 3x + 1) - x - (2x+1)(x+1) \;\equiv\; 0 \bmod 7.$$

Exercise 2.3.6. *Is* $x^2 + 2x + 1 \equiv 2x + 1 \bmod x + 1$*?*

Exercise 2.3.7. *Is* $x^2 + 7x + 5 \equiv x^2 + 3x + 1 \bmod x + 1$*?*

Exercise 2.3.8. *Is* $x^2 + x + 2 \equiv x^2 + 1 \bmod (3, x+1)$*?*

The final concept we need in order to state the AKS primality test is the order of an element in $(\mathbb{Z}/n\mathbb{Z})^*$. It turns out that $(\mathbb{Z}/n\mathbb{Z})^*$ is a group under multiplication modulo n; one consequence of this is that if $x \in (\mathbb{Z}/n\mathbb{Z})^*$ then there is an integer k such that $x^k \equiv 1 \bmod n$. For example, if $n = 12$ then we have seen $(\mathbb{Z}/n\mathbb{Z})^* = \{1, 5, 7, 11\}$, and we have

$$1^2, 5^2, 7^2, 11^2 \;\equiv\; 1 \bmod 12.$$

For another example, consider

$$(\mathbb{Z}/7\mathbb{Z})^* \;=\; \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}.$$

A little calculation shows that each element is equivalent to 1 when raised to an appropriate power. The case of 1 is obvious; for the others, we have $2^4 = 8 \equiv 1 \bmod 7$, $3^6 = 727 \equiv 1 \bmod 7$, $4^2 = 8 \equiv 1 \bmod 7$, $5^6 = 15625 \equiv 1 \bmod 7$ and $6^2 = 36 \equiv 1 \bmod 7$. We denote the order of x modulo n by $\mathrm{ord}_n(x)$.

Exercise 2.3.9. *Find the orders modulo 9 for* $(\mathbb{Z}/9\mathbb{Z})^*$ *and the orders modulo 11 for* $(\mathbb{Z}/11\mathbb{Z})^*$*. Note that if* $y \equiv -x \bmod n$ *then the orders of* x *and* y *are related; if* $\mathrm{ord}(x)$ *is even then* $\mathrm{ord}(y) = \mathrm{ord}(x)$*, while if* $\mathrm{ord}(x)$ *is even then* $\mathrm{ord}(y) = 2\mathrm{ord}(x)$*. This observation can save a lot of time in computing orders!*

The needed combinatorial result is the following. Recall that binomial coefficient $\binom{n}{k}$ is defined by

$$\binom{n}{k} \;=\; \frac{n!}{k!(n-k)!}$$

when n is a positive integer and $0 \le k \le n$ is an integer; by convention we set $\binom{n}{0} = 0$. There is a nice combinatorial interpretation to these numbers; they are the number of ways of choosing k objects from n objects when order does not matter. (Thus we should view $\binom{n}{0}$ as saying that, mathematically, there is but one way to do nothing!) The key result for us is that

$$\binom{n}{k} \equiv 0 \bmod n \text{ for all } k \in \{1, \dots, n-1\} \quad \text{if and only if } n \text{ is prime.}$$

Note that we must exclude $k = 0$ and $k = n$, as $\binom{n}{0} = \binom{n}{n} = 1$ for all n.

Let's check the claim by looking at some examples. If we take $n = 4$ then since 4 is composite we expect at least one binomial coefficient not to be equivalent to 0 modulo 4. We have

$$\binom{4}{1} = 4, \quad \binom{4}{2} = 6, \quad \binom{4}{3} = 4;$$

while the first and last are congruent to zero modulo 4, the middle is not (it is congruent to 2). If instead we take the prime $n = 5$ then we have

$$\binom{5}{1} = 5, \quad \binom{5}{2} = 10, \quad \binom{5}{3} = 10, \quad \binom{5}{4} = 5;$$

note all of these are equivalent to 0 modulo 5.

Exercise 2.3.10. *Verify the claim for $n = 6$ and $n = 7$. Note that it suffices to just look at the binomial coefficients with $k \le n/2$ as $\binom{n}{k} = \binom{n}{n-k}$.*

We can now state the AKS primality test.

AKS primality test

1. Test to see if N is a perfect k^{th} power. If it is, then N is composite and stop, else proceed to Step 2.
2. Find the smallest prime r such that the order of N modulo r is greater than $\ln^2 N$; in other words, for all $r' < r$ if k is the smallest integer such that $N^k \equiv 1 \bmod r'$ then $k \le \ln^2 N$.
3. If any of the numbers in $\{2, 3, \dots, r\}$ have a non-trivial common factor with N (this means they share a divisor between 2 and $N-1$) then N is composite at stop, else proceed to Step 4.
4. If $N \le r$ then N is prime and stop, else proceed to Step 5.
5. For each positive integer a that is at most $\sqrt{\varphi(N)} \ln N$, check and see if $(x + a)^N \equiv x^N + a \bmod (x^r - 1, N)$. If there is such an a such that the equivalence fails, then N is composite; if the equivalence holds for all such a then N is prime.

Remark: Note that if the AKS primality test terminates in either Step 1 or 3 then not only do we learn that N is composite, but we also find a factor. Sadly, this is not the case if the program ends in Step 5.

There are numerous expositions describing both why the AKS primality test works, as well as why it works quickly. We refer the interested reader to the paper, posted at

`http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf` as well as **ADD REFS** for these details, and content ourselves with giving a rough analysis of some of the steps and then doing some illustrative examples.

Let's consider the first step; how difficult is it to determine if N is a perfect k^{th} power? First off, we should figure out how large k might be. Let's say $N = n^k$ for some n and k. Clearly the larger n is the smaller k is; thus k is largest when n is smallest. The smallest we may take n to be is 2, and thus the largest k can be, which we'll denote $k_{\max}$, must satisfy $2^{k_{\max}} \leq N$. If we take logarithms base b of both sides we find

$$\log_b 2^k_{\max} \ \leq \ \log_b N.$$

We now use the power rule, which says $\log_b x^y = y \log_b x$ and find

$$k_{\max} \log_b 2 \ \leq \ \log_b N.$$

This equation is simplest when we take the base b to be 2, as it then reduces to

$$k_{\max} \ \leq \ \log_2 N.$$

The above bound on $k_{\max}$ tells us that Step 1 will be fast. The number of k to check is at most $\log_2 N$, which is at most $4 \log_{10} N$. In other words, the number of k to check is bounded by a polynomial in the number of digits of N, which is our criterion for 'fast'.

Exercise 2.3.11. *Prove* $\log_2 N \leq 4 \log_{10} N$. Hint: *Use the Change of Base formula for logarithms, which states* $\log_b x / \log_b c = \log_c x$ *and note that* $\log_2 10 \leq 4$.

Step 2 could take a long time for two reasons. One reason is that it might be hard to compute the order of N modulo r, and the second is that we might need to take r large before we find an r such that the order of N modulo r is at least $\ln^2 N$. Fortunately results form number theory tell us that we can find an r without going to high (r is at most a constant times $\ln n$ to a fixed power at most 5), and thus r is small enough that computing the orders won't take too long.

As r is not too large, Step 3 is fairly fast. We just have to run the Euclidean Algorithm to find the greatest common divisor of n and $a \leq r$, and the Euclidean Algorithm is very fast. Step 4 is the simplest of all to analyze: it's just a simple comparison, which takes no time.

We are thus left with Step 5. For most large numbers, almost all of the run-time is due to this step. It is not pleasant to implement polynomial modular arithmetic, though this can be done in environments ranging from Java to Mathematica.

Let's look at some representative examples. For our first test, consider $N = 21$. It's a little absurd to use the AKS primality test here, as we can see N is composite and equal to 3 times 7. Going through the algorithm, we see N is not a perfect k^{th} power. As $\ln^2 21 \approx 9.26$, we need to find the smallest prime r such that the order of N modulo r is at least 10. The smallest such r is 19, and the multiplicative order is 18. In other words, $21^{18} \equiv 1 \bmod 19$, and no smaller power of 21 is equivalent to 1 modulo 19. If we had tried to take $r = 17$, we would have found 21 has multiplicative order of 4, as $21^4 = 194481 \equiv 1 \bmod 17$. We now move to Step 3 and look at the greatest common divisors of 21 and all $a \leq 19$; we are in luck as we very quickly discover that 21 and 3 have a common factor, and thus 21 is composite.

Let's do one more example. Consider now $N = 20413$. A quick check shows that N is not a perfect k^{th} power. We have $\ln^2 N \approx 98.4$, so we must find a prime r such that the multiplicative order of N modulo r is at least 99. The smallest such r is $r = 101$, and the order of N is 100. We now look at the greatest common divisors of N with $a \in \{2, \dots, 101\}$, and unfortunately all of these numbers are relatively prime to N. We thus continue to Step 4. As $N > r$, we move on to Step 5. We now have to deal with the polynomial congruences. Fortunately, this example isn't too bad; taking $a = 1$ shows that the congruence fails and thus N is composite! Explicitly, we have

$$(x+1)^{20413} \ \not\equiv\ x^{20413} + 1 \bmod (20413, x^{101} - 1).$$

To see the failure in its full glory, we write $(x+1)^{20413} - (x^{20413} + 1) \bmod (20413, x^{101} - 1)$ below:

$$\begin{aligned} &18358 + 13974x + 12056x^2 + 7124x^3 + 19263x^4 + 16714x^5 + 16714x^6 \\ &+19263x^7 + 7124x^8 + 12056x^9 + 13974x^{10} + 18358x^{11} + 11645x^{12} \\ &+2603x^{13} + 19830x^{14} + 19591x^{15} + \cdots + 19830x^{98} + 2603x^{99} + 11645x^{100}, \end{aligned}$$

which clearly is not zero!

Index

Bibliography

[Th] T. M. Thompson, *From error-correcting codes through sphere packings to simple groups*. The Carus Mathematical Monographs, Number 21, the Mathematical Association of America, 1983.