

Crypto Book Working Notes

Midge Cozzens, Steven J. Miller (sjm1@williams.edu) and Wesley Pegden

January 4, 2011

Contents

- 1 Error Detecting and Correcting Codes** **3**
- 1.1 Motivating Riddles 5
 - 1.1.1 Statements. 5
 - 1.1.2 Solutions. 7
- 1.2 Definitions and Setup 11
- 1.3 Examples of error detecting codes 14
- 1.4 Comparison of Codes 17
- 1.5 Error correcting codes 18
- 1.6 More on the Hamming $(7, 4)$ code 21
- 1.7 Additional Riddle 24

Chapter 1

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 is 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 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 effect 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 him 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 he 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 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,

	#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
B W W	black			right			win
W B B	white			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.

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:

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

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:

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

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 an error correcting code.

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

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

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

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 than 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.

Example 1.2.11. *Consider the binary code*

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

Is this code exactly 2-error detecting? It can detect one error and it can detect two errors, but it cannot tell if one or two errors were made. For example, if we receive 011 what was the intended message? Was it 111 (and thus there was one error) or 000 (and thus there were two errors)? We can thus detect that an error has occurred, but we don't know what the error is.

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*

$$C = \{1000, 0100, 0010, 0010, 1110, 1101, 1011, 0111\}; \quad (1.1)$$

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 (1.1); 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 (1.1). 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 be 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

$$C = \{0000, 0011, 0101, 1001, 1100, 1010, 0110, 1111\}. \quad (1.2)$$

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 (1.2). 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 is 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 [?]), 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 [?]. 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

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

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 $\text{Ham}(i)$ denote the i^{th} of the 16 code words.

$\text{min} = 7$; $\text{max} = 0$.

For $i = 2$ to 16,

 For $j = 1$ to $i - 1$,

 Let $d = d(\text{Ham}(i), \text{Ham}(j))$;

 If $d < \text{min}$ then $\text{min} = d$;

 If $d > \text{max}$ then $\text{max} = d$.

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; see **ADD REF** for more details. 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 Thompson's monograph [?]; 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 over the repetition code. 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?

Exercise 1.6.5. Generalize the square code and consider the square (16, 9) code. Show that it can detect one error, and has 9 out of 16 digits (or 56.25%), devoted to our message. Of course, similar to the original square code we don't need the final entry, and could construct a (15, 9) code, which would lead to 60% of our message devoted to information! We could continue to push this method further, and look at square $((n + 1)^2, n^2)$ codes. Almost all of the message is now devoted to the information we wish to transmit; unfortunately, we are still stuck at being able to detect only one error.

Exercise 1.6.6. Try your hand at further generalizations of the square code. What if you looked at a cube? For instance, if you took a $3 \times 3 \times 3$ cube with a $2 \times 2 \times 2$ sub-cube, you would have 8 message digits. How many errors would you be able to detect? What if you looked at larger cubes? What if you looked at higher dimensional analogues, such as hypercubes?

We described the square code for a message with four digits of information; of course, we could do a larger one (and sketched some of the details in Exercise 1.6.5). If we had r message digits, then we need $r = m^2$ for some m as we have to be able to arrange the message in a square. How many check digits are there if our message is of size m^2 ? There are $2m + 1$ check digits: m from the m rows, another m from the m columns, and then the final check for the row and column checks; of course, we can similarly remove the last check and make do with just $2m$ check digits. Note that as m grows, almost all (percentagewise) of the digits of our message are transmitting information, and almost none are being used for checking. Specifically, the size of the code is $m^2 + 2m + 1 = (m + 1)^2$, m^2 digits are used for transmitting information, and thus the percentage being used for information is

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

As m tends to infinity, this can be made as close to 1 as we wish (though always, of course, a little bit less).

What does this mean? Our previous codes had efficiencies like a third or four-ninths; now we can make a code as efficient as desired, with (percentagewise) almost all digits devoted to transmitting information! There is, of course, a trade-off. We don't get anything for free. This code can only detect one error. Correcting one error in a code of length $(m + 1)^2$ is much worse than correcting one error in a code of length four or seven. This means that the square codes are not the end of the story.

Exercise 1.6.7 (Hard). *It is worth mentioning, though, that the square code can be extended to several dimensions, which allows multiple errors to be detected. Try and think of a multidimensional generalization that will detect and correct two errors.*

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

$$\begin{aligned}d_1 &= d_3 + d_5 + d_7 \pmod{2} \\d_2 &= d_3 + d_6 + d_7 \pmod{2} \\d_4 &= d_5 + d_6 + d_7 \pmod{2}.\end{aligned}\tag{1.3}$$

Exercise 1.6.8. *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 (1.3) is satisfied. If there is an error, then at least one of the three equations in (1.3) 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, \dots, 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.

Freedonia security caught 10 spies from the Kingdom of Sylvania who attempted to poison the wine of Freedonia's King, Rufus T. Firefly the Great. 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 force the spies to drink the wine. 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, though each spy's cup is potentially a mixture of wine from many bottles.

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

We first give a solution for 1000 bottles, and then extend slightly to 1024. As a general piece of advice about riddles in particular and math problems in general, often

the numbers chosen have significance. Here 1000 should make you think of either 10^3 , or a number just a little less than $1024 = 2^{10}$. It is the latter point of view that is especially fruitful for solving this riddle; note that if we write 1024 as 2^{10} , the number of spies enters as the exponent of 2!

Let's label the bottles unimaginatively 1, 2, 3, ..., 1000. We have the first spy drink a mixture from bottles 1, 2, ..., 500. If he dies, we know the poison was in one of the first 500 bottles, while if he lives we know the poison is in one of the final 500 bottles. We now have the second spy drink a mixture from bottles 1, 2, ..., 250 and bottles 501, 502, ..., 750. If she dies, the poison was in either bottles 1 through 250 or 501 through 750.

Note that we eliminate half of the bottles from looking at whether or not a spy is alive or dead. Further, if we look at the two of them together we eliminate three-fourths of the bottles! For example, imagine the first spy dies and the second lives. Then the poisoned bottle must be in bottle 251, 252, ... or bottle 500. Why? Since the first spy died, the poisoned bottle is one of the first 500; since the second spy lived, it's not one of the first 250.

Continuing in this manner, we can determine which bottle is poisoned, and we kill at most 10 spies.

Let's recast the answer using binary numbers – this will be far more useful for studying codes. Imagine now we have 1024 bottles, but now we label them in binary. Thus the first bottle is 0000000001, the second bottle is 0000000010, ..., the 784th bottle is