# Computational Thinking

Paul Curzon

Version 0.1

# Preface

Magic, intelligent pieces of paper, helping people with severe disabilities, a brain built out of rope and toilet roll: what have they got in common? Something called ‘Computational Thinking’. It’s a unique way of thinking, a unique way of solving problems. It is the way of thinking that has driven the computing revolution and changed the way we live. It’s what computer scientists do.

What is computer science really about? It’s a fun, exciting subject, that is both fascinating and useful. Despite the name it’s not really about computers though they obviously come in to it. Really it’s about computation: both the theory and practice. The theory side is about understanding the limits of computation, and how we can understand them. Practically it is about how we design and build things that compute. Computers compute, but so do other things including people, so it’s not just about computers.

Computer science is also about gaining a unique set of skills that blend together to give a different way to both think about and solve problems. That skill set is called computational thinking.

This book aims to give an introduction to computational thinking as well as a range of computer science topics more generally. We will come across things like:

- algorithmic thinking,
- transforming problems,
- analytical thinking,
- logical thinking,
- rigorous thinking,
- Mathematical and Computational modeling and
- understanding people.

They are all aspects of computational thinking. There are other more common skills too, like team-working skills and communication skills, but we will focus on the ones that you may not have come across. They may just be long meaningless words now, but if you read on I hope you will get an idea of what they mean, how they work together and why they give such a powerful way of thinking about solving problems. I hope you will see why computational thinking is such an important skill for everyone to learn at school, whether they go on to become a computer scientist or not.

*This is an early draft of a booklet on Computational Thinking. If you are a teacher I would love to hear feedback about it and especially if you think it would be worth my writing more and getting it published.*

# Chapter 1: Searching to Speak

One of the worst medical conditions I can imagine is locked-in syndrome. It leaves you with all your mental abilities intact, but totally paralyzed except perhaps for the blink of an eye. Your intelligent mind is locked inside a useless body, able to sense all around but unable to communicate. It could happen to anyone, out of the blue, as a result of a stroke. If you wanted to help people with locked-in syndrome, the obvious thing might be to become a doctor or nurse, but how could a computer scientist help?

As it happens, we don't have a cure so there isn't a lot medics can do beyond trying to make them more comfortable. An obviously big issue though is to help people with locked-in syndrome communicate. The answer to what a computer scientist might do then seems obvious – we could invent some new technology: based on eye tracking software perhaps. However, some computational thinking can give a more powerful answer than just "we need technology".

One of the most uplifting books I have read is "The Diving Bell and the Butterfly". It is the autobiography of Jean-Dominique Bauby, written after he woke up in a hospital bed with locked-in syndrome. In the book, he describes life with locked-in syndrome. Bauby did have a way to communicate not only to write the book but also with medical staff, friends and family. He did it without any technological help at all. How did he do it?

Put yourself in his position, waking up in a hospital bed. How could you communicate. How could you write a whole book? You have only a helper with a pen and paper to write down your "words"? All you can do is blink one eyelid. You can't otherwise move, including speak, in any way. Can you come up with a way to do it?

**Simple as a,b,c**

What you need is some kind of agreed method of turning blinks into letters. A first idea then might be that one blink means 'A', 2 blinks means 'B' and so on. The helper just has to count the blinks and write down the corresponding letter.

In coming up with this idea, we are already doing some computational thinking. In particular, we've done a form of '***algorithmic thinking***'. A computer scientist would call that method an *algorithm*: a series of steps to follow in a given order that achieves some goal (here to communicate letters and words). The beauty of algorithms is that the steps can be followed without those involved having any understanding of what they are doing. Here the helper presumably would know what they were doing and why, but the book would still get written even if they didn't. All the helper needs to do is count blinks and write down the letter. We could give them a table to lookup the letters in so they could do it mechanically without much thought at all. The beauty of algorithms is that they allow things to be done 'mechanically' - that means computers can blindly follow the instructions too.

This algorithm actually comes in two parts. There is one part for Bauby to follow (blinking the right number of times) and one for the helper (count through the number of blinks and write down the corresponding letter when

the blinks stop). In fact computer scientists have a special name for this kind of algorithm that passes information between two people or computers – it's called a '*protocol'*. If both people follow their part of the protocol then the words Bauby is thinking will end up written on the piece of paper. If either makes a mistake – loosing count for example so not following the protocol – then the message won't get through. The great thing about computers is they don't get things wrong like that – they follow their instructions exactly, every time.

Algorithmic thinking is a particular kind of problem solving – one where you don't just come up with an answer like '42' but come up with a solution in terms of steps that others (including a computer) can follow to get answers. We came up with a solution like that for Bauby. It sounds pretty slow though. Maybe there is a better way. Thinking about better solutions is also a part of algorithmic thinking.

**How did Bauby do it?**

We should remember that the helper can speak, so we should make use of it. The protocol Bauby used involved the helper reading the alphabet aloud "A...B...C..." When the letter he was thinking of was spoken he blinked. The helper would write that letter down and then start again, letter after letter. Try it with a friend – communicate your initials to them that way...now think about that being the only way you have to talk to anyone. I hope your name isn't Zebedee Zacharius Zog or Zara Zootle!

Once you've tried it you may have realized there are some more problems we have to solve to really make it work. Having tried it a few times, you might also be able to think of other ways to improve the algorithm. What can you come up with?

One thing you may have realized is that there are more characters than the 26 letters to deal with – we need spaces, digits and punctuation too. We need to add them to the list too. We need to allocate a number everyone agrees on (their position in the list) to each new character. Another thing to deal with is what happens if the person blinks by mistake? We need a way to say; "Ignore that last blink and start the letter again". One way might be agree that blinking twice quickly means that. Maybe you thought of other problems that need solving too.

Algorithmic thinking is about thinking about all those details and finding solutions. It's about realizing there can be many ways of doing things and coming up with the best for the particular situation. Also notice that one of the issues was about people. In theory our solution works. Just blink at the right time. We could arrogantly say the people should just do the right thing and it's their fault if they get it wrong. In practice they will sometimes blink at the wrong time. It's better if we solve the problem in a way that does work for people. After all it is a person we are trying to help in the first place! Computational thinking is about '***understanding people***' too

**Doing it better**

One way to speed things up is to realize that sometimes halfway through a word you can guess what it is. If you have got "a-n-t-e-l" it would be a pretty good bet to assume it was antelope. So you could change the rules to allow

the helper to make guesses like that. We'd need a way for the person to say no after such a guess – perhaps the rule could be they just blink if the word is right and do nothing if not. This is of course essentially just predictive texting – it's the algorithm phones use (and maybe that's where you got the idea from if you thought of it!)

Bauby's helpers did do a version of predictive texting. Bauby also realized that the ABC algorithm could be improved upon in a different way. He had been the Editor-in-chief of the French women's magazine, *Elle*, before that hospital bed so knew about language. He knew that some letters are more common than others in natural language. E is the most common letter (in both English and French) for example. He therefore got the helper to read out the letters in order of frequency in French "E...S...A...R..." That way the helper got to the common letters more quickly.

The algorithm for finding the letter frequencies was actually invented by Muslim scholars over a thousand years ago. A similar trick has been used through the ages to crack secret codes. In fact Mary Queen of Scots was behead because Queen Elizabeth I's spymaster Sir Francis Walsingham was better at computational thinking than she was. That's another story though.

Bauby's idea does illustrate another part of computational thinking though – that of '***transforming problems***'. Often problems turn out to be essentially the same as something you've already seen in a different situation. If you already have a solution for that other problem then you can just use it. Algorithms are just a way of giving this kind of general solution. We've seen this twice already: with predictive texting and frequency analysis. A phone has the same problem working out what words are being typed as a helper has working out the word someone with locked-in syndrome is thinking. Once we realize that then any solution we come up with for one can be used for the other. Similarly when we recognize the commonality in cracking codes and guessing letters then we can see that the frequency analysis solution invented for one can be used for the other.

**How fast is that?**

Let's get back to Bauby's algorithm. We've improved things for sure. The new ways must be better than our original idea. An obvious question though is how fast actually is it – "How long did it take to write that book?" Is it the best we can possibly do, or could we have come up with an even better algorithm, and so helped him write the book much more easily ?

We need a way of measuring how good an algorithm is. One way would be to do it experimentally – to time how long each takes to communicate some specific passage. We could do it lots of times with different people and see which way is fastest on average. That would take lots of time and effort. There is a better way.

We can use some more computational thinking and do some '***analytical thinking***'. We will use some simple maths to work out an answer. First, rather than think about time lets think about the work done. If we count how many letters of the alphabet the helper has to say, then we could always then turn that in to time just by working out how long it takes to say a letter. We have done something called '***abstraction***' another aspect of computational thinking

to simplify the problem. Abstraction is just a long word meaning ‘hiding some of the details’. The idea is used throughout computing as a way of making things easier to do. Here we are using “number of letters said” as an abstraction of the actual time.

So how do we work it out? There are several questions we can ask. What is the best case? That is, what is the fewest letters the helper would possibly have to say to write the book? We could also look at the worst case. If we are unlucky, how bad could it be? Finally we can look at the average case – that will give us a realistic estimate of how much work it actually took.

Let’s, for the sake of argument, stick to communicating just letters of the alphabet without punctuation, etc. We will analyse our simple algorithm of asking A, B, C …

In the best case, the whole book would be nothing but A’s: “AAAAAAAA” (perhaps expressing the pain he is in). To communicate a single letter ‘A’ we just say one letter ‘A’ (one question) and we have the answer. Multiply that by the number of letters in the book and we have the best case for writing the whole book. The worst case, perhaps dictating a story where someone snores the whole time, "ZZZZZZ", takes 26 questions to get each letter. That gives us the bounds on what communicating anything would be – no better than 1 and no worse than 26 letters spoken per letter communicated.

A closer estimate would be the average number of questions asked per letter: the average case. But that’s easy to work out. In a long message, for every ‘A’ you communicate, on average there will also be a ‘Z’ somewhere else in the message. For every ‘B’ there will be a ’Y’, and so on. That means on average over the whole book roughly 13 questions will be asked per letter dictated. Multiply the number of letters in the book by 13 and you have an estimate for the amount of work done to write the book. Multiply that by the time for the helper to say a letter and you have the time taken to communicate the book.

Bauby's modification, asking about common letters first, improves things a bit – maybe it will be down to 9 or 10 letters spoken. We could work that out more precisely using a different kind of maths, based on the frequencies of the letters. So it is an improvement, but the worst case for a letter is still 26 and as any computer scientist knows we can do far better. It is possible to work each letter out with only 5 questions, guaranteed. That’s not the average case, it’s the worst case!

Can you work out what 5 questions you need to ask?

**Do it in 5**

Whether you came up with the answer or not, I guarantee you know instinctively what the right sort of question is, but only if we look at a different problem.

Let’s play a game of 20-questions – the children’s game where I think of a famous person and you try and guess who I’m thinking of by asking questions. The twist is that I will only ever answer yes or no. Play a game with a friend, thinking about the kind of questions you ask as you do.

Let’s see how a game might go.

“Are they female?”

No

“Are they alive?”

No

“Are they a film star?”

No

“Are they from Britain?”

No

“Are they from America?”

No

“Are they from Asia?”

Yes

“Are they from India?”

Yes

“Are they a politician?”

Yes

“Is it Ghandi?”

Yes

Chances are when you played the game, you asked similar questions. You almost certainly didn’t start asking questions like “Is it Adele?”, “Is it Jessica Ennis?”, “Is it the Queen?”. You would never have got the answer in 20 questions that way. You only ask that sort of question (as in our game) at the end when you are pretty sure you know who it is. Instead you probably asked a question like "Are they male?" first.

Why is that a good first question? Well, it’s because it rules out half the possibilities, whatever the answer. If you ask “Is it Adele?”, then you rule out all but one if you are right, but if wrong (more likely) you only rule out one person. You would have to be lottery-winning lucky to do well that way. So the secret to playing 20 questions is to ask questions that rule out half the possibilities each time.

**How good is that?**

How good is that? Well let’s suppose I might be thinking of one of a million people at the start. If I rule out half the people each question, how many questions does it take? After one question, we are down to 500,000 people left, 2 questions 250,000, then 125,000 people, about 64,000 people (simplifying a little to make the numbers easier!), 32,000 people, 16,000, 8000, 4000, 2000, 1000… After 10 questions there are only 1000 people left out of those million it could be. Keep going…500 left after another question, 250, 125, 64 (ish) 32, 16, 8, 4, 2 and on the $20^{th}$ question there is only one person left it could be. If you can ask perfect halving questions you are guaranteed to win.

So with the right questions, in the worst case it takes only 20 questions to find the one thing I am thinking of out of a million possibilities. Compare that with our saying on average it takes us 13 questions (and worst case 26) to find one thing out of 26 letters of the alphabet. Yes/No is no different to Blink/No Blink. When we asked, is it A? Is it B? we were doing the equivalent of asking "Is it Nelson Mandela?" You are trying to work out one of many things I am thinking of, just the same. It is actually the same problem!

**A new algorithm**

So if it's the same problem then surely the same strategy will give us a better solution than the ones we came up with so far. What is the equivalent of our halving solution for letters of the alphabet? We need to halve the letters of the alphabet left. The obvious first question is "Is it before N?" The next question depends on the answer to the first one. If the answer was "Yes" then we next ask, "Is it before F?" If the answer was no we ask "Is it before T?" and so on. This way we guarantee to get to any letter of the alphabet the person is thinking in only 5 questions.

In fact we can even improve things a little using the frequency analysis trick. With only 26 letters we could for example make it so we get the letter E in only 3 questions. Similarly we could still use the predictive texting trick to guess words that were only partly completed. All those solutions still apply here.

**Search algorithms**

The reason our solution carried over is because the problem was essentially the same. The problem is a 'search problem': given a series of things, find one particular one we are looking for. The solutions to this problem are clalled 'search algorithms'. They are methodical ways of finding things. The first approach of checking each of the possibilities in turn (Is it A?, Is it B?...or Is it Adele? Is it James Bond, …) is an algorithm called '**linear search**'. Sometimes it's the best you can do. For example, if you see a robbery and the police set up an identity parade, you couldn't do better than linear search – check each face in turn until you see the person in the line who did it! Linear search works well when there is no order to the things you are searching through. If yo are searching for a jumper that could be in any draw of your chest of drawers, start at the top and check them one at a time.

The other algorithm we looked at involved finding halving questions: Is it before N? Are they female? Finding halving questions is a general problem solving strategy called '**Divide and conquer**'. If you can come up with a divide and conquer solution to a problem, it is likely to be very fast as repeated halving gets you down to one answer very quickly, and certainly faster than checking one thing at a time. The simplest divide and conquer search algorithm is called '**binary search**'. Imagine lining all the things you are searching through in order, smallest at one end, largest at the other. Binary search involves going to the middle and checking whether the thing you are looking for comes before or after it. You then discard the other half and do the same again on what is left. You keep doing that until only one thing is left – the thing you were looking for. That is probably close to what you do if given a big paper telephone directory and want to find a particular name. You certainly wouldn't start at page 1 and check name each in turn

until you find the one you are looking for!

There are many more search algorithms than just these two. For example, how does Google search through every web page on the planet in fractions of a second? It needs a better algorithm still?

Search algorithms make use of another form of abstraction. We abstract from the details of the particular search problem and see it as just a search problem. Then our search algorithm is a ready made solution.

**Improving life for Bauby**

So Bauby should have got the helper to ask halving questions. Think about it. 5 questions at worst rather than 13 on average, multiplied up by all the letters in his book. It's not only the book either, it's communicating with his friends and family, the doctors and nurses too. If only he had known some computer science, how much easier his life would have been!

**Algorithmic thinking first**

The thing to notice though is we haven't been looking at technology at all. So far it has all been about two people communicating. Now we have worked out a good method we can think how we could automate it with suitable technology. We could build an eye tracking system that detects blinks or an electrode cap that can pick up whether he is thinking yes or no, perhaps. The point is that whatever technology we use it would need a search algorithm underneath it. Pick the wrong one and however good the technology the communication will still be slow – 13 questions instead of 5. It makes no difference whether the helper is a computer or a human for that. If we hadn't thought about the algorithms first we could have come up with a frustratingly slow system. Computer science is not just about the technology it is about the computational thinking that goes into coming up with good solutions.

**Understanding people first?**

So we all agree with a little bit more computational thinking Bauby's life could have been improved. But wait a minute. Perhaps we got it wrong. Perhaps we would have ensured his book was never completed and his life was even more a hell. We did not start with technology but we did start with computer science. Perhaps we should have started with the person. Were we counting the right thing?

As our measure of work – our 'abstraction' we used the number of questions asked. That is the job of the helper and it may be tedious but it's not difficult. What if blinking was a great effort for Bauby. His solution involved him blinking only once per letter. Ours divide and conquer approach requires him to blink 5 times. Multiply that by a whole book. We could have made it 5 times harder.

It could be blinking is easy and our method is better. We don't know the answer, because we didn't ask the question. We should have asked it first. We should have started with the person.

Furthermore, his solution is easy for anyone to walk in and understand. Ours is more complex to follow and might need some explaining before the visitor understands and Bauby is not going to be the one to do the explaining.

**It worked for him**

One thing is certain about Bauby's solution – it worked for him. He wrote a whole book that way after all. Perhaps the helper did more than just write down his words. Perhaps they opened the curtains, talked to him about the outside world or just provided some daily human warmth. Perhaps the whole point of writing the book was that it gave him an excuse to have a person there to communicate with all the time, paid for by his publisher!

The communication method would not then be about the needs of the book, but the book helping a deep need for direct communication with a person. Replace the human with technology and perhaps you have replaced the thing that was actually keeping him alive.

On the other hand, perhaps once he is able to communicate with a computer he can get out of his hospital bed into the virtual world, emailing friends, tweeting, keeping a face book page, controlling an avatar. Perhaps we have made things better. Again we would need to find out what he wanted most.

In extreme usability situation such as this the important thing is that the user really is involved throughout. In fact it's better when designing any system for people, not just in extreme situations that they are involved. It is they who ultimately have to adapt the available resources to something that works for them, not only technically but also emotionally and socially. Otherwise we may devise a "solution" that is in theory wonderful and in practice hell on earth. Computer Scientists have to think about much, more than just computers.

***Computational Thinking***

*Computational thinking is ultimately about solving problems for people. People therefore have to come first. You have to understand the problem you are solving from their perspective, before you start coming up with solutions. Given that, the other skills then come into play. That means you have to* ***understand people****.*

***Algorithmic thinking*** *is about then coming up with a precise way to do some task, with all the details, all possibilities covered. Given an algorithmic solution other people (or computers) can then follow the instructions mechanically. They don't need to think through the problem themselves to get answers, or even do much thinking at all. Just follow a search algorithm and you will find the thing you are looking for.*

*How do you come up with algorithmic solutions? One way is to be able to spot when one problem is essentially the same as another. If we* ***can transform problems*** *into ones we have seen before then we can just reuse the solutions, perhaps adapted slightly to fit the new situation.*

*Different algorithms are better in different situations. We can* ***use analytical thinking*** *to give us solid ways to compare our different solutions. By using* ***abstraction*** *we can focus just on the details that matter in our analysis – though we had better make sure we don't lose the details that matter! We can work out which method is best for our purpose – that might mean the one that is the fastest – the most efficient – but other properties could matter too.*

# Chapter 2: The Intelligent Piece of Paper

# Chapter 3: Becoming a Wizard

In Harry Potter, a bunch of kids who happen to be lucky enough to have powers get to learn to be wizards at school. The best computer programmers are sometimes called wizards. It turns out the link is closer than you think. If you learn computing at school you really are learning the skills of a wizard: computational thinking skills. The difference is anyone that is willing to put in the time and effort, and has the desire, can become a programming wizard. There are no muggles in computing!

### A magic trick

Let's learn a simple magic trick. What you are going to do is show you can "read the mind' of a volunteer, guessing a secret card they chose even though they've told no one what it is.

Get a normal pack of cards. Lay out 21 cards in three piles of 7 face up, spreading them out so they can all be seen. Now get a friend you want to amaze. Ask her to choose a card out of the 21 laid out on the table. They shouldn't tell you or anyone else what it is. Instead they should just think of nothing but that card. Explain you are going to read their mind and work out which one they are thinking of. They must think hard about just this one card throughout, clearing their mind of everything else.

Stand in front of her and stare at her forehead intently. Tell her she mustn't say anything and definitely don't giggle She must keep a completely straight face. There is likely to be some noise and the volunteer is likely to giggle. Explain giggles are like bubbles that cloud everything so any giggles or noise makes it impossible. Emotions do the same. Say that you got a glimpse of the card but then it was clouded over by her other thoughts (or giggles if she did). You will need to start again. Ask her to point to the pile the card was in but don't say which card it actually was. Keep thinking of the same card.

Pick up the three piles but place the pile she indicated in the middle. Deal the cards out again across the rows. Do the 'mind reading' again, this time standing closely behind her staring at the back of her head. You might want to note that you are trying a different path to her mind – as the front of the head is pretty solid. Point out again that noise, emotions, giggles are all to be avoided. After a few moments give up again – too much emotion. Get her to point to the pile it was in this time. Pick the cards up with her pile in the middle and deal them out across the rows again. Try again. This time stand at her side – ask her to pull her hair away from her ears to give a clear path through her ears to her mind.

Say you think you've seen the colour! You need to try once more "to triangulate". Get them to point to a pile, pick the cards up and deal them out the same way one last time. This time stare at her other ear. Claim you've seen it – you know her card now. Look at the cards … Turn the two end piles over - "It's not in this pile....and it's not in this pile". "It's ... this one". Point to the middle card of the middle pile. Ask her if it was the card she was thinking of. It will be.

### Computing: self-working tricks

So now you can do a little magic, but what has it to do with computing? This trick is the sort of trick that magicians call 'self-working'. If you follow the steps

exactly, then you are guaranteed to get the right card. There is no sleight of hand involved at all. What matters is that you always put the pile pointed to in the middle and you always deal the cards out across the rows every time. By the last deal the chosen card will have moved to the middle position of the middle pile. A self-working magic trick is just a series of steps to be followed in a precise order. A computer scientist, as we saw in the last chapter, would call that 'an algorithm'. Magic tricks and computer programs are at heart the same thing. They are both 'algorithms'.

Let's look at the instructions in the form a computer scientist might write them:

1. Repeat 3 times:
    a. *Deal out the 21 cards* ***across the rows***
    b. Ask which pile the chosen card is in
    c. *Pick up piles* ***placing that pile in middle***
2. Deal out the 21 cards a final time
3. Reveal the card as the one in the middle card of the middle pile.

They consist of two of the common structures that a program uses: sequencing and repetition. Sequencing is just where you give instructions to do one after the other (like the steps a, b, c). You not only are being told to do those 3 steps at that point but to do them in exactly that order. Repetition (as it sounds) just means follow the instructions some number of times. Here we are told to do the (a, b, c) steps 3 times.

A magician who performs a magic trick is actually doing a very similar thing to a computer executing a program. They are both following instructions precisely: the magician has memorized the steps, the computer stored them in its memory. A magician who thinks up completely new tricks or adapts old ones is actually doing a very similar thing to a programmer writing a new program, or modifying an existing one. They are both doing 'algorithmic thinking'.

**Prove it!**

Now, I've told you this trick always works, so if you tried it hopefully it did. How can you be sure it always works though? Maybe you were just lucky. I know before I stand up in front of a live audience I want to be absolutely certain the tricks are going to work! Perhaps you should try it some more on your own before you try it with an audience! To make sure you check different possibilities each time, you'd better keep shuffling the cards and picking a different 'chosen' card. Programmers call this testing – in their case, it involves running the program lots of times with different inputs – different data typed in whenever the computer asks for it. If you were testing a calculator's program, you would test the different numbers and operations and check it gave the right answer.

Now the question is, how much testing do you need to do before you can be sure it works every time? Ideally you would test every possibility. Even for something as simple as our card trick that would take a long, long time. Now imagine testing every possibility in a video game: that's every combination of button pushes through all possible games. It's just infeasible.

This is where some rigorous thinking comes in. Programmers don't test willy-nilly. They do it in an organized way. They apply some rigorous thinking and test representative values rather than all, in a way that checks if there is a problem then it should be found. It's called a test plan. There are different ways you can do it, but we will skip the details and just give you the idea.

So back to our magic trick. Can you come up with an argument for not checking every possibility?

Well, one thing to notice is that the actual values of the cards shouldn't make a difference to whether the trick works. All that shuffling didn't really help matters. It is the position chosen (whatever card is there) that matters. So, actually we can argue that if we just check that a card starting in every start position of the 21 ends up in the middle of the middle pile then we can be sure the trick always works. We only need to do 21 tests! If they all work we can be confident it always works.

Notice providing an argument doesn't mean coming up with an excuse for doing less work – you have to come up with a solid reason that any missed tests won't matter. It's about being logical. Programmers learn similar skills, and there it matters. If the program you are writing is actually the autopilot of an aircraft or a medical device injecting a cocktail of potentially lethal cancer therapy drugs into a patient then you want to be really, really sure it gets it right. You can't test all and just testing some possibilities at random isn't good enough.

So that's a lot less work. We've turned something infeasible into a fairly simple task. We can do even better though. We can prove that it always works without any tests at all!

We can give a simple argument together with a picture – what is called a diagrammatic proof – that all of those 21 positions end up in the middle if they are the position being thought of. Look at the diagrams. They track what happens to each card as the selected pile is placed in the middle. What we have done first is some modeling. We don't mean making plastic planes, or the houses of a train set, but a mathematical model. That just means creating a mathematical representation of some situation we are interested in. Sound scary? Its not! You are looking at one. The picture is a simple model of the 3 piles of 7 cards. We've just used a number from 1 to 21 to represent each card. This is also using abstraction again – remember that just means hiding details to make a situation simpler. In our model we have hidden the detail of what each card actually is. For our problem proving our trick always works, all we need to know about is it's position. That's what the number tells us.

So we have a model. What do we do with it? Well we do some logical reasoning about it. Let's look at the first steps of the trick. We do some ham acting pretending to read their mind (ignore that – more abstraction). Then we get them to point to the pile and we put that pile in the middle. That means that whatever pile it started in, the chosen card is one of the seven in the middle as seen in the second diagram. We don't need to worry about all 21 starting positions from this point on. All that matters is where the middle seven end up.

So what happens next? Well we deal the cards out across the rows. We end

up with the cards as in the third diagram. The seven that matter: cards 7-13 end up spread across the middle of the three piles. Of course we now do it all again, so if the cards had ended up anywhere else we are about to move them to the middle. There are now only three places that those 21 original cards could possibly be. The places labeled in this diagram XXX, YYY and ZZZZ.

We deal across the rows and where do those three cards end up? One in the middle of each pile (see Diagram 3). That means when the person points to a pile at this point we immediately know which card it is. (Maybe we could use that fact to come up with a new presentation of the trick!) This time when we gather the cards putting the pile pointed to in the middle we are guaranteeing the chosen card is in the middle of the pack. If we deal them out in the same way the chosen card must end up the middle card of the middle pile (see Diagram 4).

With the help of some modeling, a bit of abstraction and a lot of logical reasoning we have proved beyond doubt that the trick always works. You can do it in front of people you want to impress with confidence! That said proofs can have mistakes in them just as much as tricks can, so you may want to double check the proof. Even when you have a proof it is also a good idea to do some testing too, to sanity check the results – after all our proof is a proof about the model not the real world. If the model is wrong the proof tells us nothing about the trick for example. Testing and proof go together to increase confidence in an algorithm whether it's a magic trick or a computer program.

Computer programmers really are wizards, but they don't leave it to chance over whether their spells (algorithms) work. With a bit of maths they can be sure they work.

### *Computational Thinking*

*The core skill of computational thinking is **algorithmic thinking**, coming up with algorithmic solutions that always have the desired effect. It's one thing to believe your algorithm always works. It's a completely different thing for it actually to. Writing algorithms involves thinking through all the oddball cases when something different might happen – and making sure the algorithm deals with them. How can you be sure? By using **logical thinking** to create a rigorous argument, either to reduce the amount of cases you have to test or to prove it always works, at least in theory. The good computer scientist knows though that it's vital to sanity check your proof so testing is still important. It's also good for shaking out the big obvious problems, while proof is good for finding the tricky situations where you got some small detail wrong. That sort of problem will only crop up occasionally so is very hard to find by testing as you have to be lucky enough to test the right thing.*