CS4FN: Computer Science for Fun

Computational Thinking: Searching to Speak[1]

Helping people with locked-in syndrome

- **What is Computational Thinking?**
- **How do computers find things?**
- **How do we tell which algorithm is best?**
-

Searching to Speak

One of the worst medical conditions I can imagine is locked-in syndrome. It leaves you totally paralyzed except perhaps for the blink of an eye. Your intelligent mind is locked inside a useless body, able to sense everything 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?

There's no cure for locked-in syndrome so there isn't a lot medics can do beyond making their patients comfortable. One big problem to tackle though is how to help people with locked-in syndrome 'talk'. What a computer scientist might do then seems obvious – they could invent some new technology to help. However, with some computational thinking we can give a much better answer than just "we need technology".

'The Diving Bell and the Butterfly' is an incredibly uplifting book. It's the autobiography of Jean-Dominique Bauby, written after he woke up in a hospital bed totally paralysed. In the book, he describes life with locked-in syndrome. He did have a way to communicate not only to write the book but also with medics, friends and family. He did it without any technology at all, though. How?

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 eye. You can't move in any other way. That means you can't speak. Can you come up with a way to communicate?

[1] Note that a 'glossy' version of this booklet for students is in preparation and will be available from teachinglondoncomputing.org

Simple as A, B, C

What you need is to agree a way of turning blinks into letters. Your first idea 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 doing computational thinking: the kind of problem solving that computer scientists do. It’s a kind of computational thinking called ‘***algorithmic thinking***’. A computer scientist calls the agreed way of communicating an ***algorithm***: *a series of steps to follow in a given order that achieves some goal (here to communicate letters and words)*. Algorithmic thinking is about coming up with algorithms to solve problems.

The beauty of algorithms is that the steps can be followed without those involved having any understanding of what they are doing. With our algorithm 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 letters. We could give them a table to look up the letters in so they could do it completely without any thought at all. The beauty of algorithms is that they allow people to do things ‘mechanically’ like this – and that means computers can blindly follow the instructions too.

Our algorithm for communicating 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 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 – losing 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 just came up with a solution like that for Bauby. It doesn’t just tell us what he is trying to say now. It is a way we can *always* work out what he wants to say. 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?

Bauby did have a better way, a better algorithm. We should remember that the helper can speak, so we should make use of that. The algorithm 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 wrote that letter down and then started again, letter after letter. Try it with a friend – communicate your initials to them that way. Then 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?

Sorting out the details

One thing you may have worked out is that there is more than the 26 letters to deal with – we need spaces, digits, full stops, command so on too. We need to add them to the list of letters the helper works through. 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 letters again”. One way might be to agree that blinking twice quickly means that. Perhaps 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 then coming up with the best one for the situation. Notice too that one of the problems was about what people do. 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! Computational thinking is about ‘***understanding people***’ too.

Doing it better

We can speed things up if we realize that sometimes halfway through a word we 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 a guess though – perhaps the rule could be that they just blink if the word is right and do nothing if not. This is of course how predictive texting works – it’s the algorithm phones use. Maybe that’s where you got the idea from if you thought of it! If you did then you’ve just used another computational thinking skill: ‘**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. 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.

Bauby’s helpers did use 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 a lot about language. He knew that some letters are more common than others in human languages. 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 how common they are – their frequency. In English the order is "E...T...A...O...". In French it is "E...S...A...R..." He spoke French so he used the French order. That way the helper got to the common letters more quickly.

A similar trick has been used through the ages to crack secret codes. The algorithm of using letter frequencies was actually invented by Muslim scholars

over a thousand years ago. In fact Mary Queen of Scots was beheaded because Queen Elizabeth I's spymaster Sir Francis Walsingham was better at computational thinking in this way than she was. That's another story though.

Bauby's idea of using frequency analysis is another example of '***transforming problems***'. Once we have recognized that cracking codes and guessing letters are similar problems, 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 way 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 do some '***analytical thinking***'. We will use some simple maths to work out an answer. First, rather than think about time let's think about the work done. If we count how many letters of the alphabet the helper has to say, then we can always then turn that in to the time taken just by knowing how long it takes to say one letter. We have done something called '***abstraction***'. It is another part of computational thinking, used to simplify problems. 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 taken.

So how do we work out how many letters have to be said? There are several questions we can ask. The simplest is: what is the best case? 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.

The best and the worst

Let's, for the sake of argument, stick to communicating just letters of the alphabet without digits and punctuation. We will analyse our simple algorithm of the helper saying 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 telling 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. It's always 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', 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 how much work was done to write it. Multiply that by the average time for the helper to say a letter and you have the time taken to write 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 the frequencies of the letters. So it is an improvement, but the worst case for a letter is still 26.

As any computer scientist knows, though, 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 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 by asking questions like "Is it Adele?", "Is it Usain Bolt?", "Is it the Queen?" You would never have got the answer in 20 questions that way. You only ask that sort of question at the end when you are pretty sure you know who it is (as we just did). 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 millions 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 people 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 the original million it could be. Keep going…500 left after another question, 250, 125, 64 (ish) 32, 16, 8, 4, 2 and on the 20th 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 person I am thinking of out of a million possibilities. Compare that with our saying 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?", "Is it Mickey Mouse?" 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

That's where the idea of '**transforming problems**' comes in again. 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 alphabet each time. 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. That way we are sure to get to any letter of the alphabet that the person is thinking of in only 5 questions.

We can even improve things more 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. We could also still use the predictive texting trick to guess words that were only partly completed. All those solutions still apply here.

Search algorithms

Our solution carried over because the problem was essentially the same. It is a 'search problem': given a series of things, find one particular one we are looking for. The solutions to this problem are called 'search algorithms'. They are sure-fire ways of finding things. The first approach of checking each of the possibilities in turn (Is it A?, Is it B?...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 that did it! Linear search works well when there is no order to the things you are searching through. If you 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.

Our other algorithm 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 far, far 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 remains – 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 each name 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 problem and see it as just a search problem. Then our search algorithm is a ready-made solution for lots of problems.

Thinking about it another way, once we have come up with a strategy to win at 20 questions, we can **generalise** that solution to the idea of divide and conquer – we have a *general* strategy that works for other problems too.

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 talking 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. It has all been about two people 'talking'. Now we have worked out a good way, a good algorithm, 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 is, 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. Computing 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. Our divide and conquer algorithm 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 algorithm is better. We don't know the answer, because we didn't ask the question. We should have asked 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. Thinking about people is important!

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 algorithm would not then be about the needs of the book, but about 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 talk to 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 need to find out what he really wants.

In an extreme usability situation like this the important thing is that the user really is involved throughout. We call this '**user-centered design**'. In fact it's better when designing any system for people, not just in extreme situations. It is they who ultimately have to adapt what's available to make it work for them, not only technically but also emotionally and socially. Otherwise we may devise a 'solution' that is in theory wonderful but in practice hell on earth. Computer Scientists have to think about much, more than just computers.

The computer science

Search algorithms

Given something to search for (known as the 'key') a **'search algorithm'** guarantees to find it if it is there. The key could be a letter someone is thinking of, a number in an array, a film star's web page, or a record in an employee database.

One simple search algorithm is called '**linear search'**. It involves lining up all the things you are searching through and checking them one at a time from one end to the other. If you find the thing you are looking for you can stop. If you note its position, you can go straight back to it. If you get to the end without finding the key then you know for sure that it is not there at all.

A faster way of searching is called '**binary search**'. It involves lining everything up in a known order, like numerical order or alphabetical order. That allows us to do checks that rule out half of the list at every step. We check the middle entry. If the key is before the middle entry in the order then the key must be in the first half of the list (because they are in order). If the key comes after the middle entry then it must be in the second half. We rule out half the list and do the same again on the part that remains – repeatedly until there is only one thing left. It is either the key, or the key is not there.

Efficiency Analysis

There are lots of different algorithms for searching. How can we choose between them? One way is on the basis of how efficient they are. We can choose a particular critical operation that gives a good idea of how much work is done – like the number of questions asked or the number of blinks needed. We can then work out how often that operation happens in the best case, the worst case and on average.

Computational Thinking

Computational thinking is about solving problems for people. People therefore come first. You have to understand the problem you are solving from their point of view, before you dream up solutions. Otherwise your great technical solution will be useless. To be a great computer scientist, you have to **understand people**.

Algorithmic thinking is about devising a precise way to do a task, with all the details covered. Given an algorithmic solution other people or computers can then follow the instructions mechanically. They don't need to solve the problem themselves to get answers. Follow a search algorithm and you find whatever you are looking for.

One way to come up with solutions is to spot when one problem is the same as another. If we **can transform problems** into ones we have seen before then we can just reuse the solution. Once we have a good search algorithm we can adapt it for use with lots of different search problems.

Thinking about it another way, once we have come up with a strategy to solve a particular new problem, we can **generalise** that solution to a strategy that works for other problems too. Asking 50-50 questions in a game generalises to the divide and conquer strategy. Similarly we can **generalise** an algorithm

for a specific problem to give a search algorithm. Generalising the idea of asking “Is it A?”, Is it B?” gives us the linear search algorithm that applies to any search problem.

We can **use analytical thinking** to give us solid ways to compare different algorithms. By using **abstraction** we focus on the details that matter – though we have to make sure we don’t lose the details that matter! The method that is best for our purpose might be the fastest but other properties like memory needed could matter too.

Brought to you by:

Teaching London Computing:

teachinglondoncomputing.org

Computer Science for Fun:

cs4fn.org

This cs4fn story was written by Paul Curzon, 2014. The cs4fn team gave support, particularly Jonathan Black. Zali Collymore-Hussain gave valuable comments that led to improvements.

For courses and resources developed for teachers in London, including our classroom activities and slides directly linked to this booklet visit teachinglondoncomputing.org a joint project between Queen Mary, University of London and King’s College London funded by the Greater London Assembly.

For more cs4fn resources for schools, visit cs4fn.org/teachers/

Department for Education
SUPPORTED BY MAYOR OF LONDON
COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE