Queen Mary
University of London

Computational Thinking: Cut Block Logic Puzzles

Paul Curzon
Queen Mary University of London

Learn how to solve a logic puzzle and find out about why logical thinking is a core part of computational thinking. Discover how generalisation and pattern matching are the secret skills of experts, both in computer science and other areas too, from chess players to firefighters.

Version 1.0, 30 November 2015

A Logic Puzzle

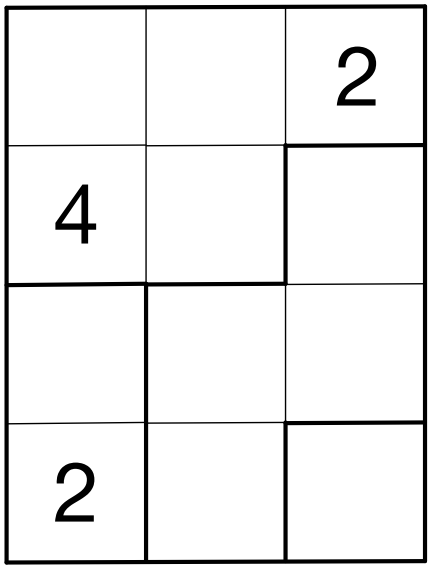
If you enjoy doing logic puzzles and are good at them you will probably make a good computer scientist. You've probably heard of Sudoku: logic puzzles based on a grid of numbers. There are a lot of different kinds of logic puzzles though, that all need the same abilities to think logically. My favourite kind of logic puzzle at the moment was invented by Japanese puzzle inventor Naoki Inaba called 'Cut Blocks'.

A Cut Block puzzle consists of a block of squares, with different areas marked out using darker lines. There are two rules that must hold of a completed block.

1) Each area must contain the numbers from 1 up to the number of squares in the area. For example, the topmost area in the puzzle below consists of 5 squares so those squares must be filled with the numbers: 1, 2, 3, 4 and 5 with no repeated numbers. If the area has two squares, like the one bottom left below, then it must be filled with the numbers 1 and 2.

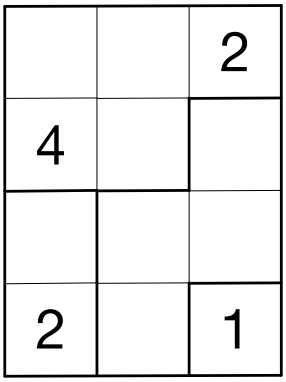
2) No number can be next to the same number in any direction, whether horizontally, vertically or diagonally. So in the grid below, the fact that there is a 4 on the side means there cannot be a 4 in any of the 5 squares surrounding it.

Here is a simple one for you to solve. Try to complete it before you read on.

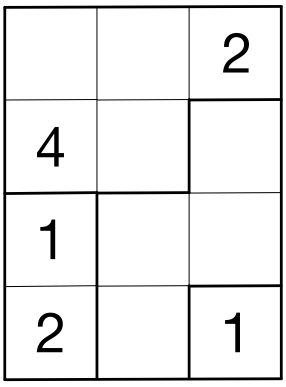


A way to solve it

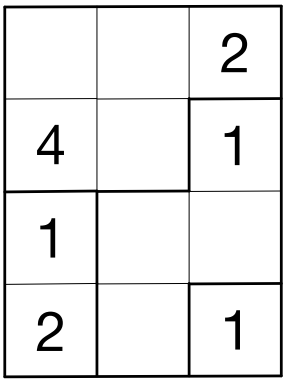
Here is the logical reasoning I used to complete the puzzle, based on the rules and the numbers given. At the bottom right is an area containing a single square. That area has size one so must contain the numbers from 1 up to … well 1. That means it must be 1 as below.



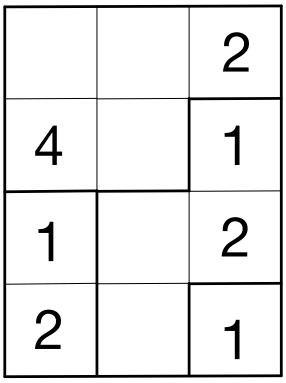
Next at the bottom left we have an area of two squares. It must contain the numbers 1 and 2. One square already has a number 2 in it, so the only possibility left for the other square is 1.



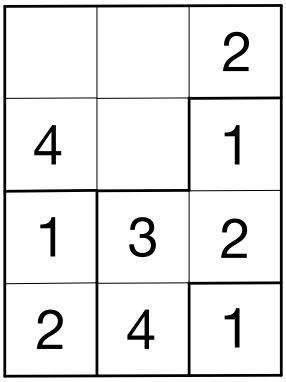
The remaining two areas have 4 and 5 squares respectively. We now have to be a bit cleverer than so far. Look at the 1 in the bottom corner. The fact that it is a 1 means none of the three squares round it can be a 1. However that area has only four squares in it and one of them must be a 1. That means the last square in the area that isn't next to the 1 must be the 1 because there isn't any where else for it to go. We get:



Next we can try and work out where the 2 goes in that same area. There is a 2 next to both the lower two squares, leaving the square sandwiched between the two 1s as the only possibility for the 2.



There is also a 4 above that area and the new 4 can't be next to it. It must be at the bottom. That determines which way round the 3 and 4 must be in the remaining two places.



We are now left with the top area. We can fill it in using similar reasoning. The 1 in the adjacent area means there is only one possible place for the last 1 to go.

1		2
4		1
1	3	2
2	4	1

Because of the 3 in the middle just below the area, there is then only one place for the 3 to go, leaving the final square for the 5.

1	3	2
4	5	1
1	3	2
2	4	1

We have solved the puzzle. We did it by applying the two rules and our basic facts of known numbers. From them we repeatedly worked out new facts about our puzzle. We have been using a particular kind of **logical reasoning** called **deduction** where we work from known facts and the rules of the puzzle to give us new facts. It is essentially the way Sherlock Holmes works his detective miracles. He notices things about people and deduces new facts as a consequence from them. The more facts he learns the more he can then go on to deduce. Computer scientists and mathematicians use similar reasoning. Good programmers use this kind of reasoning to convince themselves that their programs work.

Matching Patterns, Creating Rules

What we have just been doing is deducing facts directly from the two rules. As you do more puzzles, and get more experience though you will start to solve the puzzles in a different way. You will start to use some of your natural

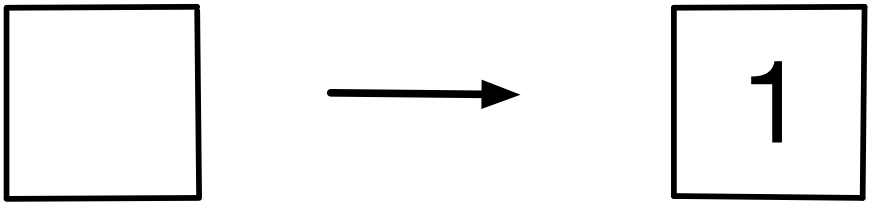
computational thinking skills. You will start to do **pattern matching** against situations you have seen before. That will let you solve puzzles faster with less thought. It will also allow you to do **generalisation,** widening the situations you pattern match against. You will with experience start to create some new quicker and very general rules to use. Let's see how.

The Single Square Rule

Going back to the way we solved the puzzle above, we worked out that when we have an area consisting of a single square, it must contain the number 1. Having realised that, we don't have to think it through again, we can just treat it as a new rule that follows from the original.

3) The square in an area with only one square holds the number 1.

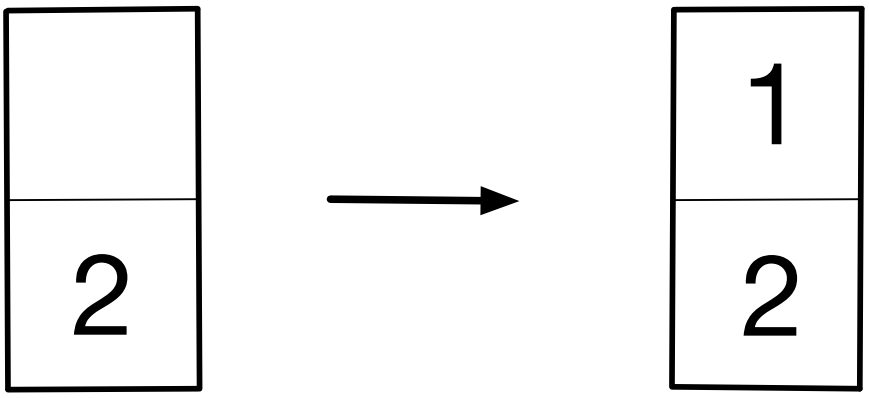
We can draw a diagram to represent the rule, rather than just use words. We use an arrow to show the change we make to the grid. On the left hand side we draw the position we pattern match against and on the right hand side what we would change that to. Rules like this are called **rewrite rules**. This diagrammatic rule says that if we find an empty area of size one then we can transform it to a square with 1 in it.



We can now just apply this rule directly without ever thinking about why it holds. Our logical thinking can now work at a higher level at least in this case.

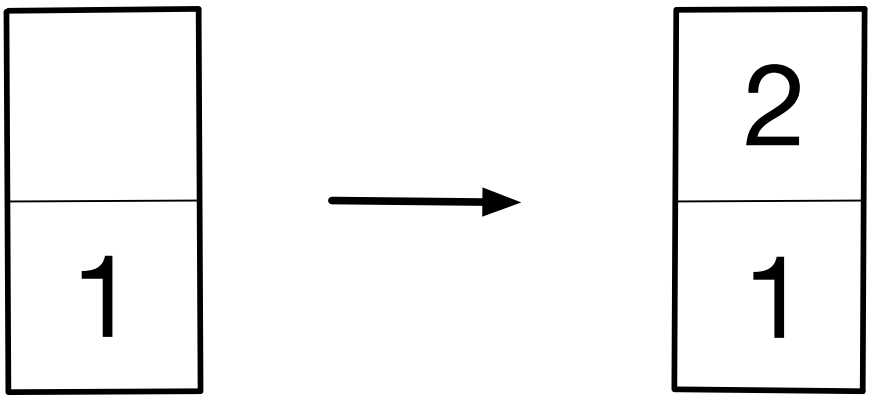
The Two Square Rule

Similarly we saw that if we have an area of size 2 with one square filled with a 2 then the other square must be 1.



Notice we have already generalised this from the actual example in our puzzle. It doesn't matter where the 2 is. Our picture applies upside down too!

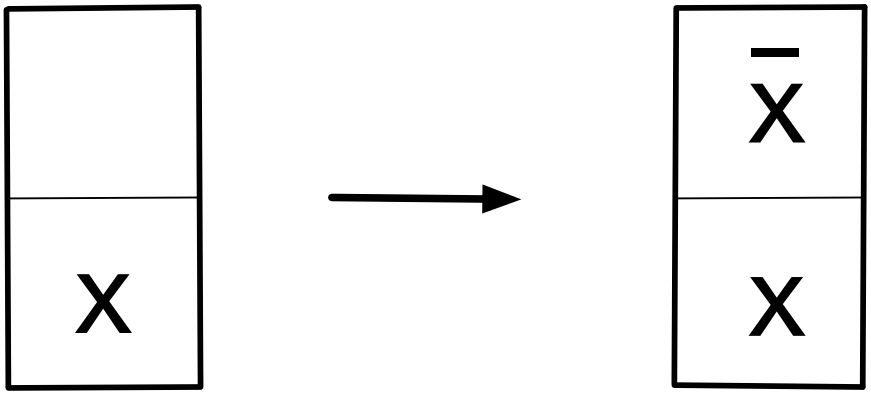
We can **generalise** the rule further though. By the same reasoning, if an area of size 2 has a 1 already placed then the other square must be 2. In a diagram:



Combining these two facts gives us the generalised rule:

> **4)** If a square in an area of size two holds a 1 or a 2 then the other square holds the other number.

We can give it as a diagram if we use a letter x to represent any number (just like the way mathematicians use x s and y s as variables in algebra). An x can stand for 1 one time we use the rule, and as a 2 another time, as long as it doesn't change in the middle of any particular time we apply it. Our rule then becomes:



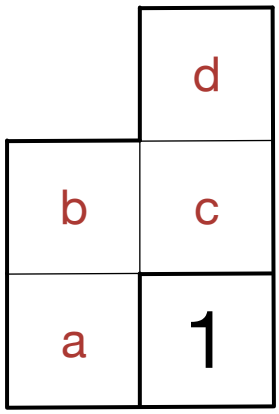
Here we are using $\overline{x}$ here to mean the other number that the x isn't this time. So if x is 1 then $\overline{x}$ is 2, and if x is 2 then $\overline{x}$ is 1. This rule can match an area of size 2 in any orientation, whether horizontal or vertical and whether the known

number is on top or below, to the left or to the right. This diagram turns in to our original diagrams just by setting x to 1 or 2.

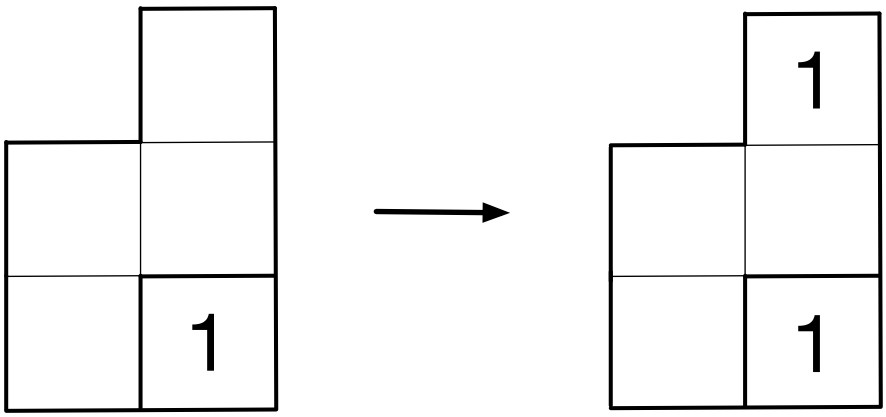
Rather than deducing facts from given facts using the rules, we are now deducing new rules that hold given the original rules. These are called **derived inference rules**. Whenever we see a situation that **matches the pattern** of one of our new rules, we don't have to think any more, we can just apply the rule.

The Corner Rule

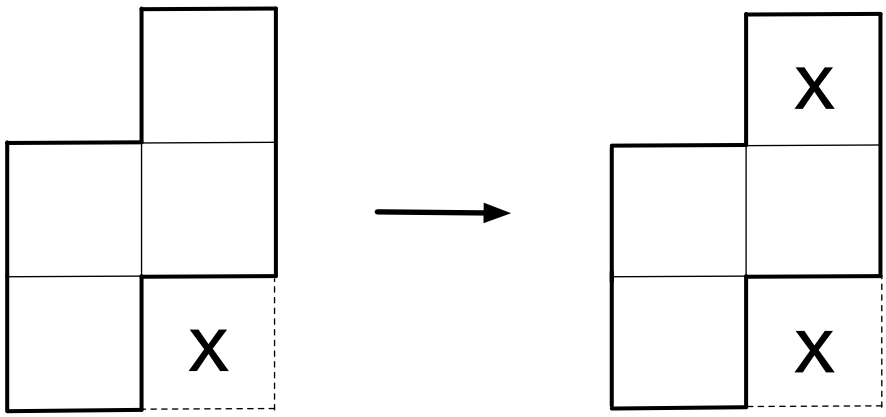
Let's look at a final example from our simple puzzle that turns out to be quite useful. In the bottom right corner, we were able to deduce where the next 1 in the area of four squares should go. it was possible because there was already a 1 in the adjacent area, nestled into a corner as below.



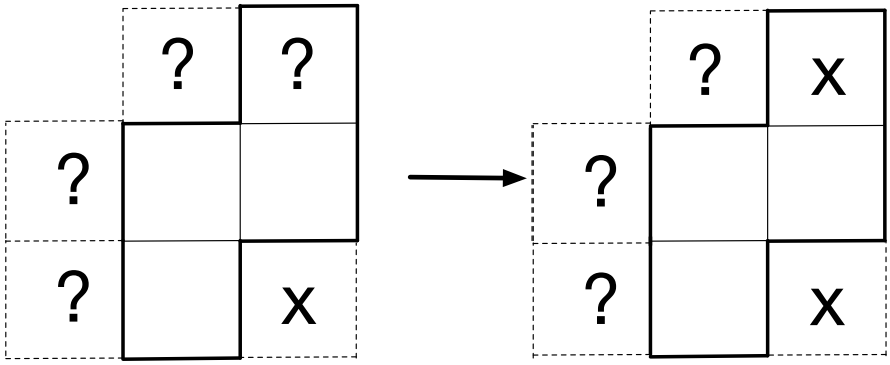
There must be a 1 in position a, b, c or d. However, there can't be a 1 next to another 1. That rules out positions a, b and c. The 1 must be in position d as it is all that is left. We can draw that step as a rewrite rule diagram.



Of course any of the squares shown as blank could already be filled with other numbers and the rule will still apply - that is another way of generalising our new rule. Also, as before, the number we are pattern matching against doesn't have to be a 1. It could be any number formed as part of a bigger area. If we use a letter x again to represent any number then our rule becomes:



We can even generalise our rule in a further way too. The area we are filling doesn't have to be exactly that shape. The extra square could be in any of four positions round the far edge of the corner. We've used question marks in the following to act as variables that show the possible positions of the square of interest.



Also as before, the rule will apply upside down or on its side, rotated or reflected. Perhaps you can think of even more ways to generalise the rule.

Equipped now with this very general rule, if you find any situation that you can pattern match it against in a puzzle, then you can apply it. That means you can fill in a missing number, as indicated by whatever matches the x.

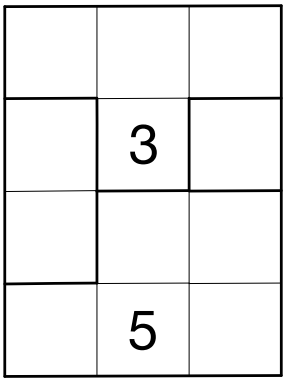
Most people who do puzzles don't bother to write down the rules they work out as they get better and better at doing the puzzles - they just remember them and apply them when the chance arises without much thought.

Computer Scientists like to write things like that down though. Why is it a good idea? Well, for one thing you can use them to teach other people how to do the puzzles without them having to work it all out for themselves (as I just did for you). They can even be used to teach computers how to do the puzzles. That is because, rules like this can be used as the basis of computer programs.

Notice how, in writing the rules down, generalisation here goes hand in hand with **abstraction**: the idea of hiding detail to make things easier to think about. In the diagram for the rule, we have used several abstractions to describe what we can pattern match against. For example, the variable x is an abstraction. It abstracts away from (i.e., hides the detail of) the actual number involved - we can apply it whatever the number. Similarly we have abstracted away from the details of the area of the shape that already has a number and we are using question marks as another kind of variable to abstract away from the position of the fourth cell in our representation of the rule. We have also abstracted away from the orientation - the diagram can be rotated or reflected in any way to match a puzzle state.

Another puzzle

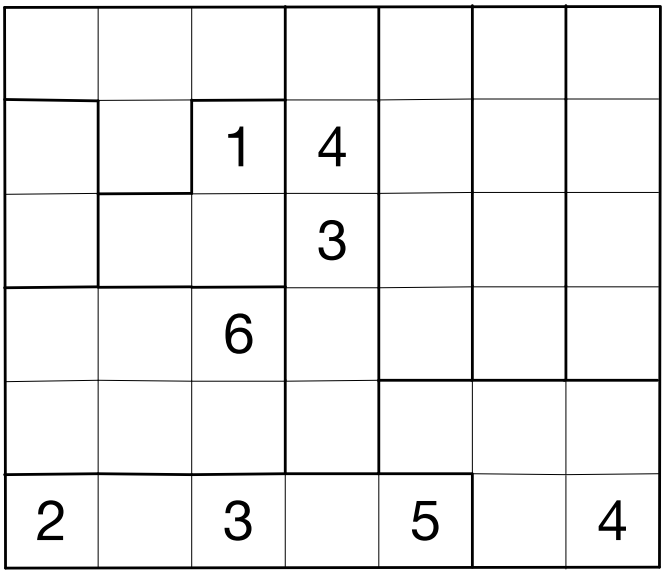
Here is another puzzle to try. See if you can use any of our rules above to solve it. As you fill in numbers you will find that new rules apply. If none of our derived rules apply you might have to go back to the original puzzle rules. Remember that rule 2 says that a number can't be next to itself.



The answer is at the end of the booklet.

A harder puzzle

Here is another, much bigger, harder puzzle. As you solve it, look out for other rules you might devise, either that are immediately useful again in solving this puzzle or that might be useful for future puzzles.



HINT: In looking for a new rule, think about what happens when you have lines of areas of size 4 next to one another.

Experts at work

The more experience we have doing the puzzles, the more rules we mentally accumulate and the faster, more easily we can do them. This is the way that chess grand masters play chess too. They recognise positions of the game as being similar to situations they have seen before. They then use rules that their experience suggests will be a good move. By thinking that way they avoid having to think through multiple moves ahead, which is slow and error-prone for a human. Computers on the other hand do play like that, playing out lots of alternative moves and the consequences. Human chess players are geniuses at the game because they are pattern matching.

It isn't just expert chess players that think like that, pretty well all experts work the same way, whatever skill they are an expert at. Fire fighters, for example, do the same. When they have a hunch that a situation is bad and get out of a burning building just before the roof collapses, it is a similar pattern matching that is going on, but sub-consciously. Intuition is just sub-conscious pattern matching against lots of prior experience.

If you want to be an expert at anything, develop your pattern matching and generalisation skills. For any skill, to be considered a genius at it and be stunningly successful, there is a rule of thumb of how many hours of practice you must put in: 10,000 hours. Virtuoso violinists, for example have practiced playing the violin at least that amount. Similarly, the most successful programmers, the ones who have become billionaires, for example, practiced writing programs for around 10,000 hours. Even Tibetan monks who are renowned for their serenity, inner peace and compassion will have practiced meditation to gain that inner peace for a similar length of time.

If you want to be a great computer scientist, start practicing now.

Logical Thinking Matters

Why does logical thinking matter? Because logical thinking is at the heart of computer science. Computers work using logic so to be able to program them: to give them instructions you have to as well if you aren't to make mistakes. It is a key part of computational thinking that runs through all aspects of it, whether creating algorithms or evaluating them.

Logical thinking especially matters to programmers. They need to use it when developing a new program, when looking for bugs in their programs and when modifying an existing program to do something new.

Logics themselves are very simple and precise mathematical languages. Like our puzzles logics comes with a set of rules - called their **axioms** (thats essentially what our initial two rules of the puzzle are). From those axioms mathematicians can derive higher level rules, just as we did. Such logics form the foundation of programming languages, defining what each construct in the language means. Even those designing programming languages have to think logically! Having logic the basis of programming languages means we can use logical thinking to reason about what our programs do. We can even prove programs are correct. To make that possible Computer Scientists also write descriptions of what a program should do directly in logic. Then the logical reasoning is used to show that the program's logical effect is equivalent to that of the program.

A final twist is that computer scientists have even invented ways that logical rules can be treated directly as programs themselves. In this style of programming, called Logic Programming, writing a program involves coming up with rules that when applied do some computation.. Whatever language you program in, whether logic programming, procedural programming, object-oriented programming, you have to apply logical thinking, one way or another.

Computational Thinking

Computational thinking isn't really one skill, it is a collection of skills used together in different combinations. **Logical Thinking** is behind it all though**.** At its simplest all logical thinking is about is thinking clearly, being careful about every last detail and being very pedantic. One kind of logical thinking is **deduction,** where you work from facts and the rules of a situation (whether the rules of a game or the laws of physics) to deduce new facts. From those facts you deduce yet more facts, and keep going until you have worked out whatever you needed to know. For our puzzles, it was the full answer to the puzzle we wanted to know.

A step up from just doing basic logical thinking is to use **pattern matching** - spotting a situation you have seen before, allowing you to do the same thing without working out the details from scratch. A step further is to use **generalisation,** where your patterns apply to situations that aren't exactly the same as ones seen originally. They are flexed to apply to a wide variety of similar situations.

In this way, as you start to see patterns in the reasoning you are doing, you can derive new rules that take bigger steps but that also apply to a wider range of situations. We created a series of these derived rules for solving the puzzles. Of course once one person has dont he hard logical thinking work to create and write down the rules, other people (or computers) can just work with them directly.

Now solving the puzzle becomes a question of **pattern matching** against these bigger rules**.** It just involves spotting when a situation matches one of your rules, and then applying it. Anyone with the derived rules no longer has to think through the detailed logic each time. That is another computational thinking trick called **abstraction** - hiding detail. We hide the details of the low level logic in our rule, so we no longer have to deal with that detail. That allows us to think at a higher level, the level of our patterns. In creating the rules, and doing the generalisation, we are using other sorts of abstraction too.

Doing logic puzzles is a really good way to develop computational thinking skills, especially if you think about how you are solving them and try to write down the rules you develop and are using.

Answers

Here is the answer to the second puzzle.

1	4	2
2	3	1
1	4	2
3	5	1

Here is the answer to the harder third puzzle.

1	3	2	5	3	2	3
2	4	1	4	1	4	1
1	3	2	3	2	3	2
2	4	6	1	4	1	4
3	1	5	2	3	2	5
2	4	3	1	5	1	4

More Puzzles

This kind of logic puzzle now has its own puzzle book (see http://www.puzzler.com/puzzles-a-z/suguru for more details, for example). Why not create your own puzzles, or once you can program, write a program to solve them (or invent new ones for you to do).

Use of this booklet

This booklet was created by Paul Curzon of Queen Mary University of London, cs4fn (Computer Science for Fun www.cs4fn.org) and Teaching London Computing (teachinglondoncomputing.org).

We are grateful for support provided by Google and the EPSRC funded CHI +MED project involving QMUL, UCL, Swansea and City Universities. This booklet was distributed in London with support from the Greater London Assembly.

See the Teaching London Computing activity sheets in the Resources for Teachers Section of our website (http://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/) for linked activities and resources based on this booklet.