Queen Mary
University of London

Computational Thinking: Cut Hive Logic Puzzles

Paul Curzon
Queen Mary University of London

How do we solve logic puzzles? By solving Cut Hive puzzles, find out about why logical thinking is a core part of computational thinking, but how experts, from chess players to firefighters, as well as computer scientists 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, 22 February 2016

Logic Puzzles

If you enjoy logic puzzles and are good at them you will probably enjoy computer science. Above anything else, being able to think logically is important to computer scientists. It runs through the whole subject but is especially important in writing programs. Programs are founded on logic, and as we have already seen, thinking clearly through all the possibilities is important for writing correct programs (and magic tricks). They have to work under all circumstances, so when both writing them and evaluating them, the programmer has to sweat the detail.

At one level when we talk about **logical thinking** we just about thinking clearly, chasing down the small details. However, there is a deeper meaning of working with mathematical logic, and if you can do that you will be much, much stronger at arguing cast iron cases. It is all about applying rules precisely. Arguments founded on logic have no holes in them: something that the Ancient Greek Philosophers realised was an important skill. Being able to come up with solid arguments is useful whatever you do, not just for computer scientists. Logic puzzles are really about constructing an argument, but cut down to the pure logic. Thinking logically is just a skill like any other that can be learnt. It just takes practice and doing puzzles is a fun way to develop it. It is as much as anything about attention to detail.

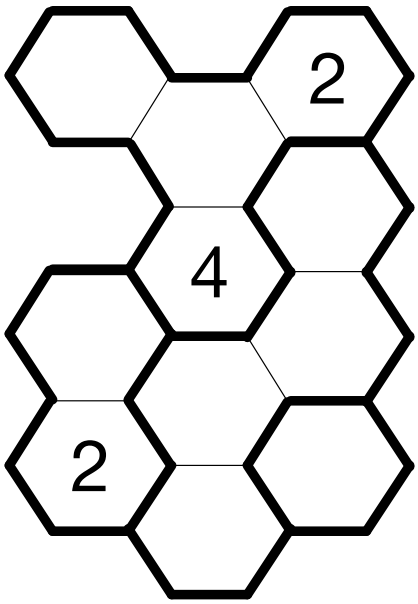
Cut Up Hives

You've probably seen Sudoku: logic puzzles based on a grid of numbers. There are a lot of different kinds of logic puzzles, and they all rely on the same ability to think logically. Let's explore logical thinking using a simple kind of logic puzzle, called 'Cut Hive' puzzles. It is inspired by puzzles by Japanese puzzle inventor Naoki Inaba, called 'Cut Blocks'.

A Cut Hive puzzle consists of a block of hexagons, with different areas marked out using thicker 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 hexagons in the area. For example, the topmost area in the puzzle below consists of 4 hexagons so those hexagons must be filled with the numbers: 1, 2, 3 and 4 with no repeated numbers. If the area has two hexagons, 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, along a shared edge. So in the grid below, the fact that there is a 4 in the middle means there cannot be a 4 in any of the 5 hexagons surrounding it.

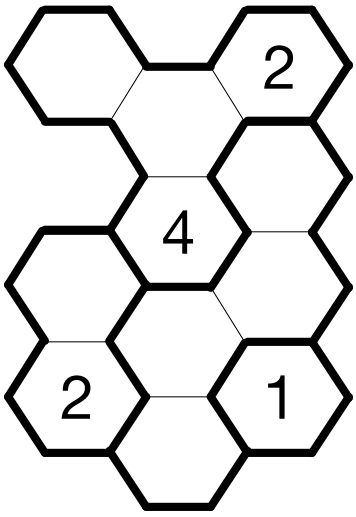
Overleaf is a simple Cut Hive Puzzle for you to solve. Try to complete it before you read on.



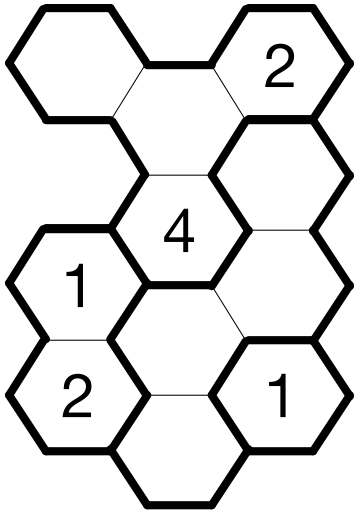
Solving a Cut Hive

Here is the logical reasoning I used to complete the puzzle, based on the rules and the numbers given. It is a cast iron argument that the filled out grid that I am claiming is a solution, really is a solution.

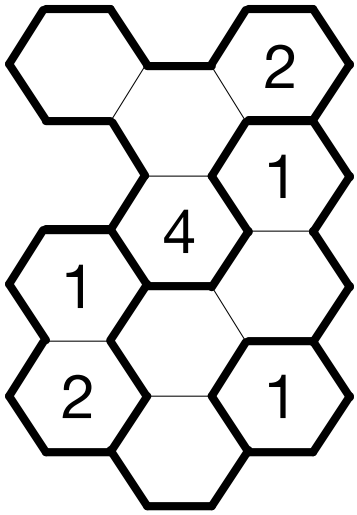
At the bottom right of the grid is an area containing a single cell. That area has size one so must contain the numbers from 1 up to … well 1. That means it must be 1 as below. Let's fill it in :



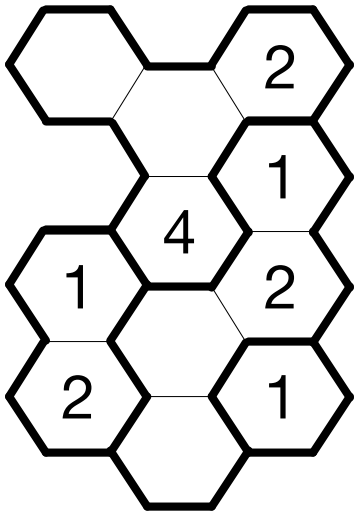
Next at the bottom left we have an area of two hexagons. It must contain the numbers 1 and 2. One hexagon already has a number 2 in it, so the only possibility left for the other hexagon is 1.



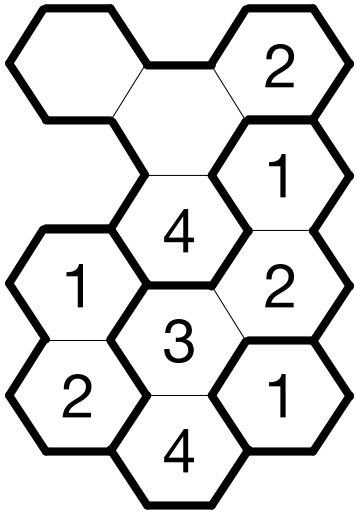
The remaining two areas are made of 4 hexagons each. We now have to be a bit cleverer than so far. Look at the 1 in the bottom corner. The fact that it is a 1 means none of the three hexagons round it can be a 1. However that area has only four hexagons in it and one of them must be a 1. That means the last hexagon 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 the following hive.



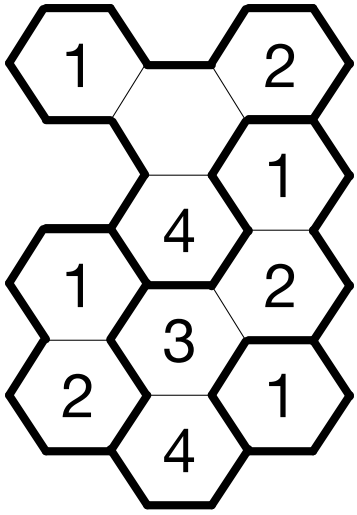
Next we can try and work out where the 2 goes in that same area. There is a 2 that touches both the lower two hexagons, leaving the hexagon sandwiched between the two 1s down the right side as the only possibility for the 2.



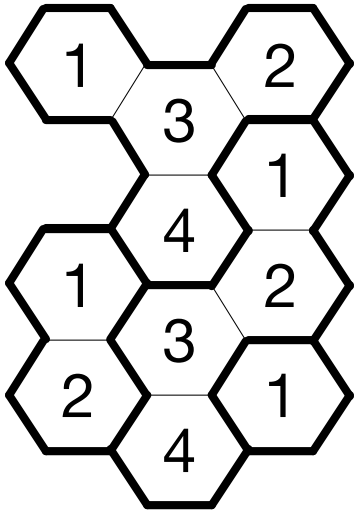
There is also a 4 above that area and the new 4 can't be next to it. It must be at the bottom. That determines which way round the 3 and 4 must be in the remaining two hexagons:



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 in the top left corner.



That means the final hexagon is a 3 as that area must have numbers 1 to 4 and only the 3 is missing. The final solution is:



We have solved the puzzle. We did it by applying the two rules and our basic facts of which numbers were already known. 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, ultimately allowing him to solve crimes. Computer scientists and mathematicians use similar reasoning. Good programmers use exactly this kind of reasoning to convince themselves that their programs work, always.

Matching Patterns, Creating Rules

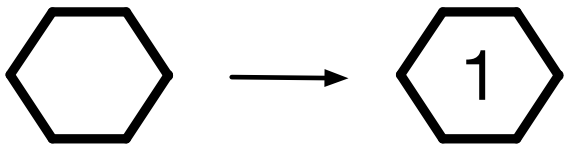
What we have just been doing is deducing facts directly from the two rules. As you do more puzzles, and get more experience though you start to solve the puzzles in a different way. You start to use some of your natural computational thinking skills: **pattern matching** against situations you have seen before, for example. 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. You can then do logical reasoning at a higher level, based on these more powerful rules. This is possible because of using logical thinking, not to solve a puzzle, but to create the new rules. That way you are still sure that they are guaranteed to follow from the basic ones. Let's see how.

The Single Hexagon Rule

Going back to the way we solved the puzzle above, we worked out that when we have an area consisting of a single hexagon, 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) IF an area has only one hexagon, THEN that hexagon 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.

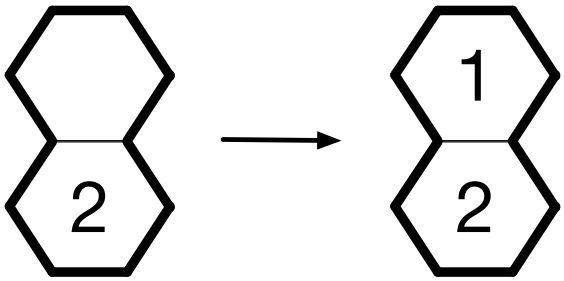


Rules like this are called 'production rules' or 'rewrite rules'. This diagrammatic rule says that if we find an empty area of size one then we can transform it to a hexagon with 1 in it.

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

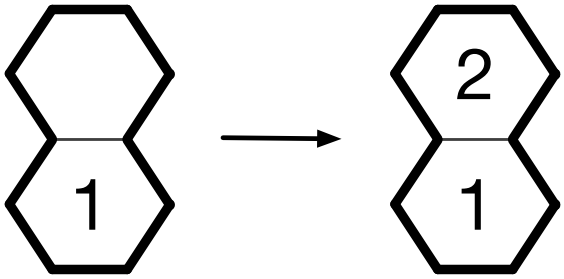
The Two Hexagon Rule

We can create another new rule for areas consisting of two hexagons. We saw that if we have an area of size two, with one hexagon filled with a 2 then the other hexagon must be 1.



Notice we can treat this as a generalised rule from the actual example in our puzzle. It doesn't matter which hexagon holds the 2, the same logic applies. Our picture applies upside down too! It also can be applied if the hexagons are linked diagonally in any direction as well.

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 hexagon must be 2.

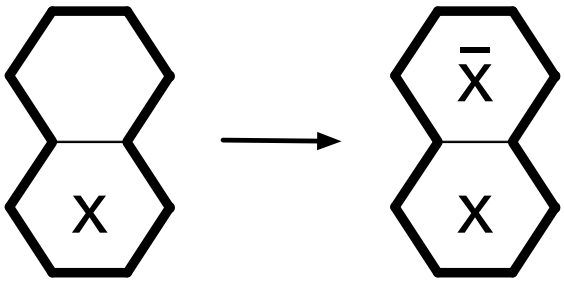


Combining these two facts gives us the full generalised rule:

> 4) IF a hexagon in an area of size two holds a 1 or a 2 THEN the other hexagon holds the other number.

We can give it as a diagram if we use a letter x to represent any number (just as mathematicians use x and y 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. A diagram of the rule is given below. We use $\bar{x}$ in the diagram to mean the other number that the x isn't this time. So if x is 1 then $\bar{x}$ is 2, and if x is 2 then $\bar{x}$ is 1. This rule can match an area of size 2 rotated in any way and so whichever way round the two numbers are. This diagram turns in to our

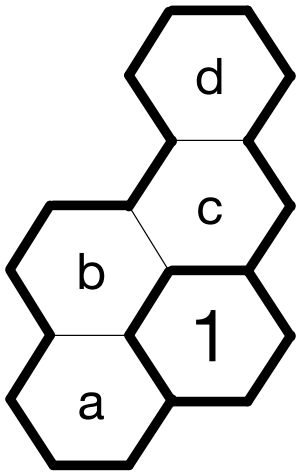
original rules (and their diagrams) just by setting x to 1 or 2. We are starting to invent a mathematical-like notation for the same reason mathematicians use symbols. It gives us a precise way to talk about things, and as our rules become more complicated that is important if we aren't going to make mistakes.



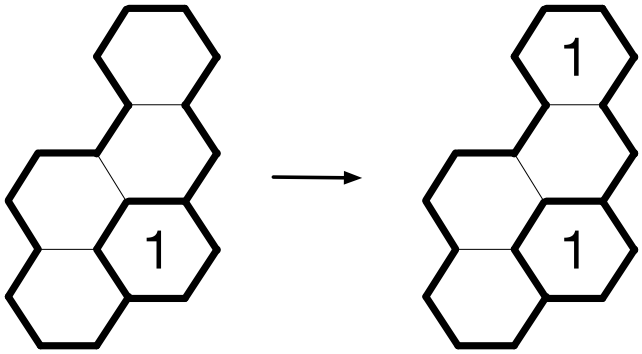
Rather than deducing facts from given facts using the rules, we are now deducing new rules that hold given the original rules. These are called '*derived inference rules*'. Whenever we see a situation that matches the pattern of one of our new rules, we don't have to think any more, we can just apply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

The Corner Rule

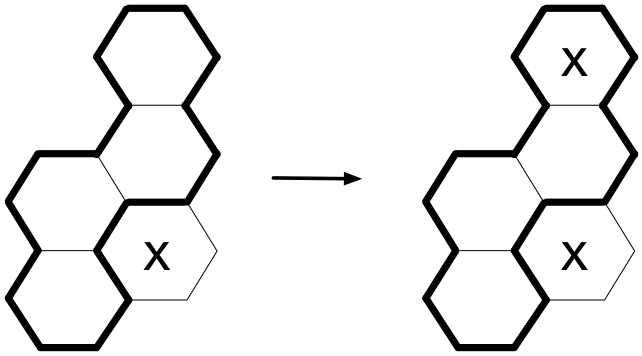
Let's look at a final example of creating a bigger rule from our solution to the simple cut hive puzzle that turns out to be quite useful. In the bottom right corner, we were able to deduce where the next 1 in the area of four hexagons should go. It was possible because there was already a 1 in the adjacent 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.



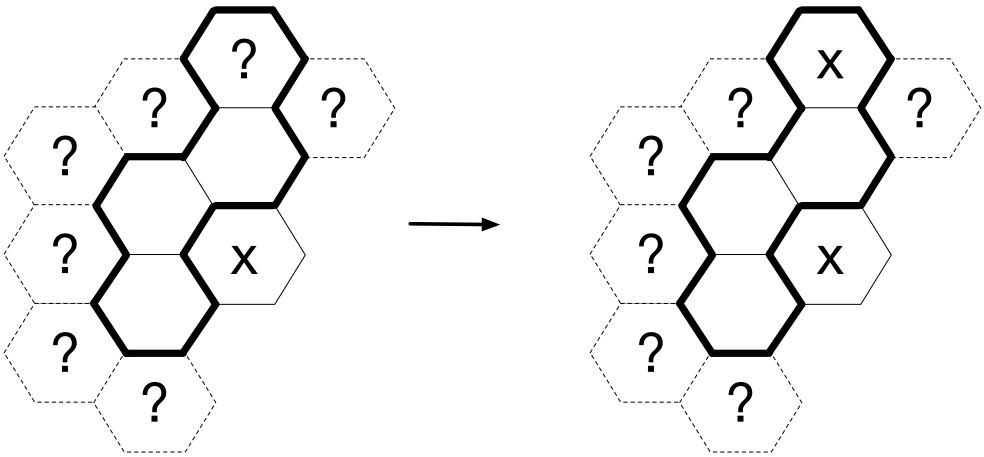
Of course any of the hexagons shown as blank could already be filled with other numbers and the rule will still apply: that is another way of generalising our new rule. Also, as before, the number we are pattern matching against doesn't have to be a 1. It could be any number formed as part of a bigger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



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 hexagon could be in any of positions round the far edge of the bigger area: anywhere that is connected but doesn't touch the corner hexagon. In the version of the corner rule below we've used question marks to act as variables that show the possible positions of the hexagon of interest.

Written in english we get a generalised rule:

> 5) IF a hexagon is surrounded by an area of size four, with only three of the four hexagons touching it THEN the fourth hexagon holds the same number as the surrounded hexagon.



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 derive and use. They just remember something that worked in the past and apply it 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. It also makes things precise. It is easy to have a false recollection of a rule that worked in the past, or for someone who has learnt it to misunderstand the detail. In either case it could lead to the rule being applied incorrectly or applied it to a new situation that it doesn't actually quite match. Writing a precise version down helps avoid this kind of faulty reasoning.

We are stretching the limits of what we can do with pictures now, though. In reality computer scientists tend to use mathematical notation (formal logics) to express rules. These languages for expressing logic are a bit like programming languages though very flexible and have the big advantage that they can easily be processed by computers and so be the basis of computers doing this kind of reasoning. The logic becomes the basis for computer programs that can solve the puzzles.

Many Artificial Intelligence systems are based on this idea of programming using this kind of *production rule*. Instead of drawing diagrams we write rules of the form IF <some situation> THEN <action to do>. A list of such rules makes up a program. If a rule applies then the computer can do the action. This is done over and over again.

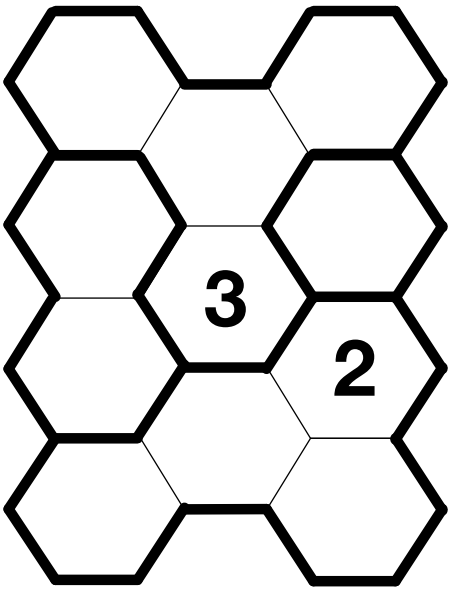
We aren't just using **logical thinking** as we become better and better at doing the puzzles. We **pattern match** the rules against the current situation to know which to apply. A production rule program is doing the same kind of pattern matching. It is doing some simple computational thinking. In writing the rules down to create that

program, **generalisation** goes hand in hand with **abstraction**: we are hiding detail about other parts of the puzzle to make things easier to think about and to make the rules as general as possible. 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 hexagon in our representation of the rule. We have also abstracted away from the orientation in the way we write the rule: the diagram can be rotated or reflected in any way to match a puzzle state.

More Puzzles

Another simple puzzle

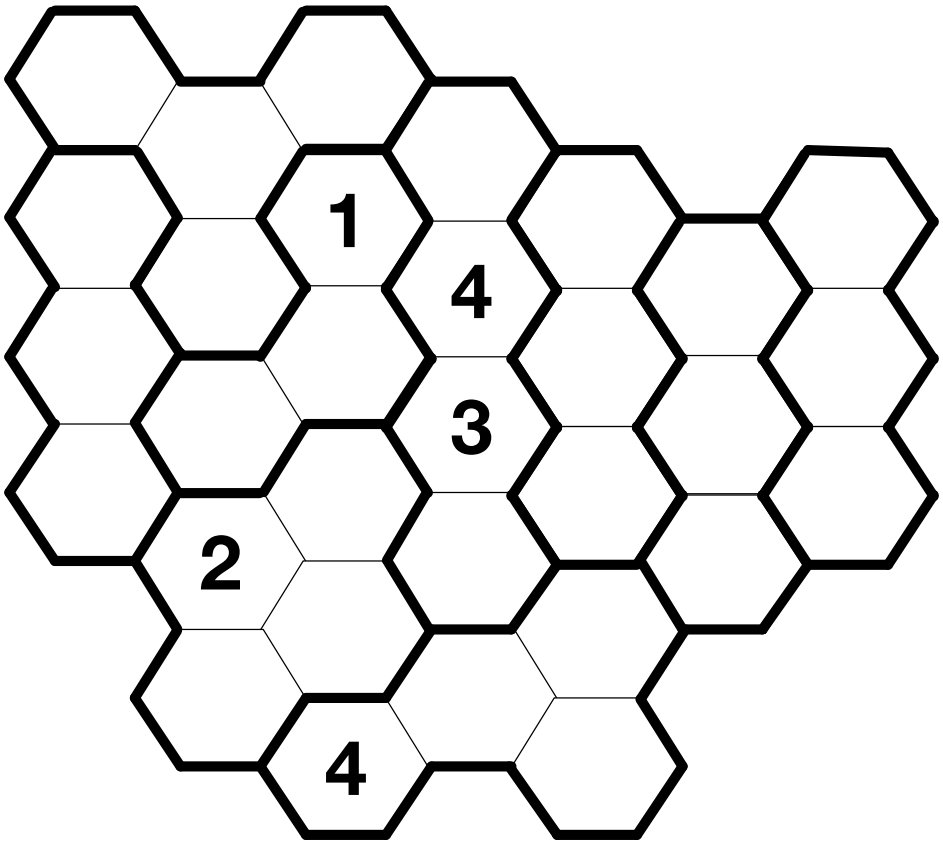
Here is another puzzle to try. See if you can use any of our rules above to solve it. As you fill in numbers you will find that new rules apply. If none of our derived rules apply you might have to go back to the original puzzle rules. Remember that rule 2 says that a number can't be next to itself.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



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 3 next to one another.



Logical Thinking and Expertise

Logical Thinking Matters

Why does logical thinking matter to a computer scientist? Because it 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 think logically 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 as well as when evaluating them in other ways, 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. Our initial two rules of the puzzle are axioms of the puzzle. From 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, so those designing programming languages have to think logically too! Having logic as 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. Our rules for the puzzle, written in a logic programming language would become a program for solving the puzzles. Whatever kind of language you program in, though you have to apply logical thinking, one way or 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 using both **logical thinking** and **pattern matching** and have built up a wealth of informal rules about the game.

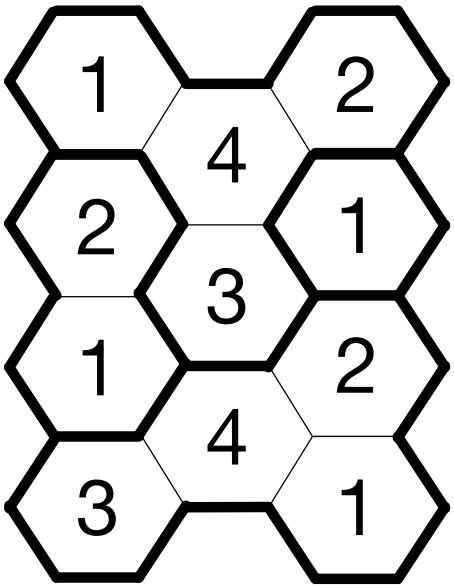
It isn't just expert chess players that think like that though, pretty well all experts work the same way, whatever skill they are an expert at. Fire fighters, for example, do the same. When they have a hunch that a situation is bad and get out of a burning building just before the roof collapses, it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

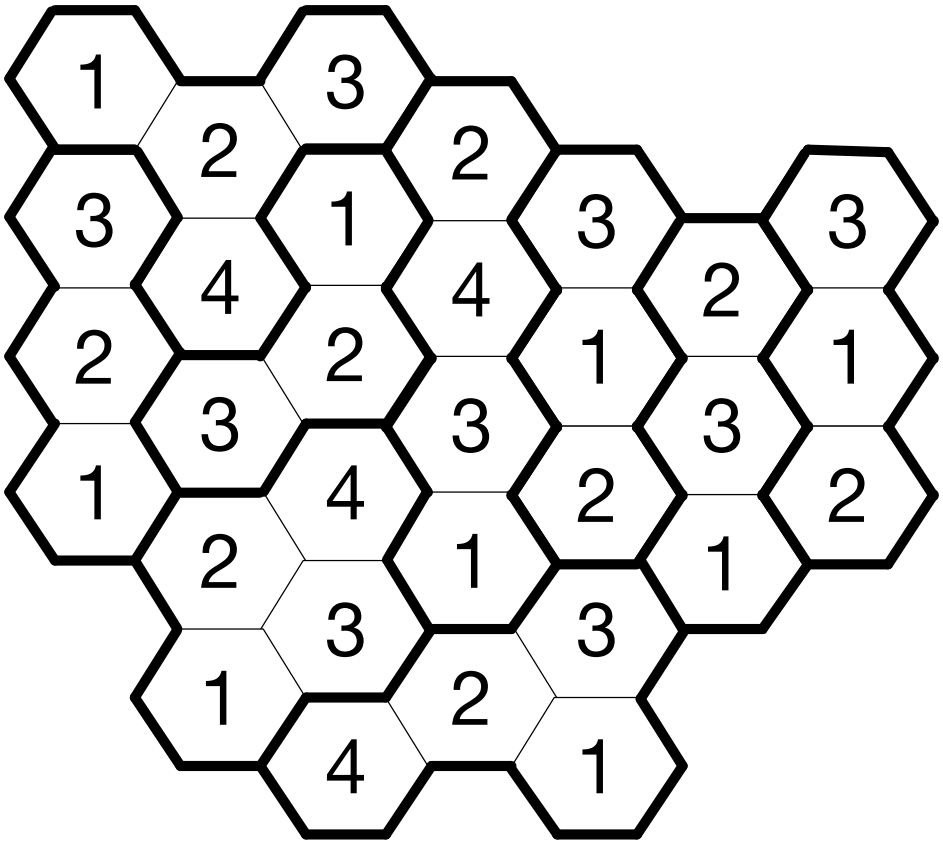
If you want to be an expert at anything, develop your pattern matching and generalisation skills. For any skill, to be considered a genius at it and be stunningly

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 your computational thinking skills now. Even if you don't want to be a programmer, developing the skills like logical thinking, generalisation and abstraction and pattern matching will make you better at whatever career you follow, whatever you want to be an expert at. Doing logic puzzles is a really good way to start to develop them, especially if you think about how you are solving puzzles as you do them, and try to write down your rules.

Answers





More Puzzles

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).

Queen Mary University of London

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 to help us create resources around puzzles.

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

With support from Google

Department for Education

SUPPORTED BY MAYOR OF LONDON

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE