Informatics 1 - CL: Tutorial 4

Satisfaction and Counting

Week 6: 22-26 October 2018[1]

Please attempt the entire worksheet in advance of the tutorial, and bring all work with you. Tutorials cannot function properly unless you study the material in advance. Attendance at tutorials is **obligatory**; if you cannot attend your allocated tutorial go to another session. Please let the ITO know if you cannot attend.

You may work with others, indeed you should do so; but you must develop your own understanding; you can't phone a friend during the exam. If you do not master the coursework you are unlikely to pass the exams.

For Part 2 of this tutorial you should load the file `CL4.hs`.
For Part 3 load the file `australia.hs`.
For the tutorial activity load the file `latin.hs`.

In tutorial CL3 we introduced Boolean functions – Boolean-valued functions of Boolean variable. In addition to the operations `(&&)`, `(||)`, and `not`, we saw that `(==)`, `(<=)`, and `(/=)`, when they act on Booleans, with type `Bool -> Bool -> Bool`, are just the familiar Boolean operations, equivalence (iff) $\leftrightarrow$, implication $\rightarrow$, and xor $\oplus$.

Situations describe things and their relationships, as collections of sets and relations. A relation, such as $(<)$, is a Boolean-valued function `(<) x y` returns `True` iff the value of `x` is less than the value of `y`; otherwise it returns `False`. This is an example of a binary relation – we also allow unary and ternary relations, and indeed n-ary relations, of any arity, n.

In this tutorial we will use logic (in Haskell) to model situations.

We look at *properties* of individuals – such as `isHappy :: Person -> Bool`; and more-general *predicates* or *relations*– Boolean-valued functions of several arguments – such as `loves :: Person -> Person -> Bool`. (We generally write our relations in curried form.) Properties are just unary predicates. The truth values `True` and `False` 0-ary predicates.

In this tutorial you will use the Boolean functions introduced last week together with the generic functions `and, or, any, all`, specialised to the following types:[2]

```
and :: [Bool] -> Bool                   any :: (a-> Bool) -> [a] -> Bool
or  :: [Bool] -> Bool                   all :: (a-> Bool) -> [a] -> Bool
```

[1]Version 1.10, 21st October 2018
[2]The library versions can handle any `Foldable` type constructor – we just need `[]`, which is `Foldable`.

1 Prelude: four-variable exercises

We start this tutorial with a few exercises whose results we will need very soon. In addition to the use of Karnaugh maps, we can produce CNF (conjunctive normal form) for an expression using algebraic manipulations. Our manipulations are rules for rewriting sub expression. The following three steps (each step is repeated until its rules no longer apply) lead to a CNF for the original expression.

- Eliminate operators other than $\wedge, \vee, \neg$. For example:
  replace $x \to y$ by $\neg x \vee y$
  replace $x \leftrightarrow y$ by $(\neg x \vee y) \wedge (\neg y \vee x)$
- Push negations in:
  replace $\neg\neg x$ by x
  replace $\neg(x \vee y)$ by $(\neg x \wedge \neg y)$
  replace $\neg(x \wedge y)$ by $(\neg x \vee \neg y)$
- Distribute $\vee$ over $\wedge$:
  replace $x \vee (y \wedge z)$ by $(x \vee y) \wedge (x \vee z)$

For some simple expressions this may be easier than using a Karnaugh map. For expressions with more than four variables, it's all you have (so far).

1. Use a Karnaugh map to derive a CNF for the expression $\mathrm{ITE}(B, G, A)$ defined as $(B \to G) \wedge (\neg B \to A)$ (in Haskell, `if b then g else a`). (You will use this in the next question.)
2. For each of the following expressions, derive a CNF using the rewriting process outlined above, then compare your CNF with one produced using a Karnaugh map.
   (a) $R \leftrightarrow (A \vee G)$
   (b) $R \leftrightarrow (A \vee G \vee B)$
   (c) $R \leftrightarrow \neg(A \vee G \vee B)$
   (d) $R \leftrightarrow (A \wedge G)$
   (e) $R \leftrightarrow (A \wedge G \wedge B)$
   (f) $R \leftrightarrow \neg(A \wedge G \wedge B)$
   (g) $R \leftrightarrow (A \to G)$
   (h) $R \leftrightarrow (A \leftrightarrow G)$
   (i) $R \leftrightarrow \mathrm{ITE}(B, G, A)$

2 A social graph

Our first example[3] of a situation has 100 individuals, `people`, given as a list of names:

```
type Person = String
people :: [Person]
isHappy :: Person -> Bool
girl :: Person -> Bool
boy :: Person -> Bool
```

```
scientist :: Person -> Bool
artist :: Person -> Bool
politician :: Person -> Bool
loves :: Person -> Person -> Bool
```

These are defined in the Haskell file `CL4.hs`.

We can ask lots of questions about a situation like this. As usual in this course we will stick with questions with Boolean answers. Each question corresponds to a statement, which is `True` if the answer is *Yes*, and `False` if the answer id *No*. For example, to answer the question, *Is Flo happy?*, we evaluate the truth value of the statement, *Flo is happy*, which is rendered in Haskell as `isHappy "Flo"`.

(a) *Is Flo happy?*
There are, at least, two ways to find out:

```
isHappy  "Flo"
not (isHappy  "Flo")
```

(b) *Does Flo love somebody?*
We can ask this in two ways.

```
or [ "Flo" `loves` y | y <- people ]
any ( "Flo" `loves`) people
```

(c) *Does someone love Flo?*
Again, we can ask this in two ways.

```
or [ x `loves`  "Flo"| x <- people ]
any (`loves`  "Flo") people
```

(d) *Does everyone love Flo?*
Once more, we can use both patterns.

```
and [ x `loves`  "Flo"| x <- people ]
all (`loves`  "Flo") people
```

Things now become more interesting ... *Does everybody love somebody?* In mathematical notation we would write the corresponding statement as, $\forall x.\ \exists y.\ x\ \mathbf{loves}\ y$.
To render this in Haskell, we first replace `"Flo"`, in question (b) above, by a variable `x`, to ask *Does x love somebody?*.

```
or [ x `loves` y | y <- people ]   -- x loves somebody
```

If `x` has a `Person` value, this expression will tell us whether *x loves somebody*.
We place this expression, `or [ ... ]`, inside the expression below:

```
and [ or [ x `loves` y | y <- people ] | x <- people ] -- everybody loves somebody
```

The Boolean value `or [ x ‘loves‘ y | y <- people ]` *(x loves somebody)* is computed for each person, `x`, drawn from `people`. The outer comprehension includes all these results. Applying `and` returns the overall result, `True` iff *everybody (every x) loves somebody.*

We have used `and` and `or` to express the meaning of $\forall x.\ \exists y.\ x\ \mathbf{loves}\ y$.

****

It is harder to use `all` and `any`. As a first step, we can write, *x loves somebody:*

```
any ( x `loves`) people
```

`all` takes, a property as its first argument — i.e. a function of type `Person -> Bool`.

We define this with a λ-expression:

```
\x -> any (x `loves`) people
```

Then we can write $\forall x.\ \exists y.\ x\ \mathbf{loves}\ y$ as:

```
all (\x -> any (x `loves`) people) people
```

[3]This is a purely fictional situation. There are 100 individuals, 50 of each sex. The other properties are generated stochastically. Any resemblance to actual persons, living or dead, is entirely coincidental.

We introduced this discussion in terms of questions, but really we just interpret statements about the situation, and see if they are true or false.

3. In the questions that follow, you are asked to use Haskell to answer questions/interpret statements about our situation.

   (a) *Does somebody love everybody?* Mathematically, the corresponding statement would be written as
$$\exists x.\ \forall y.\ x\ \mathbf{loves}\ y\,.$$
   Give a Haskell expression that evaluates to the truth value of this statement.

   Hint: you only need to make minor changes to the $\forall x.\ \exists y.\ \ldots$ *statement we interpreted above.*

   (b) In our model, as in real life, love can be unrequited (the relation is not symmetric). *Does everybody love someone who loves them?*
   $\forall x.\ \exists y.\ x\ \mathbf{loves}\ y \wedge y\ \mathbf{loves}\ x$ We call such a pair a *couple*.

   (c) *Does somebody love someone who loves them?*
   $\exists x.\ \exists y.\ x\ \mathbf{loves}\ y \wedge y\ \mathbf{loves}\ x$

   (d) *Does some **x**, love some **y**, who loves some **z**, who loves **x**?*
   $\exists x.\ \exists y.\ \exists z.\ x\ \mathbf{loves}\ y \wedge y\ \mathbf{loves}\ z \wedge z\ \mathbf{loves}\ x$ We call such a triple a *triangle*.

   (e) *Does everybody (**x**) love some (**y**) who loves some (**z**), who loves them (**x** – again)?*
   $\forall x.\ \exists y.\ \exists z.\ x\ \mathbf{loves}\ y \wedge y\ \mathbf{loves}\ z \wedge z\ \mathbf{loves}\ x$

   (f) *Does every politician love an artist?*
   $\forall x.\ \ \mathbf{politician}\ x \to \exists y.\ \ \mathbf{artist}\ y \wedge x\ \mathbf{loves}\ y$

In this style you can give precise mathematical meaning to quite complex statements in (a limited subset of) natural language. This can help us to explain and resolve ambiguities in natural language, and allow us to implement our questions as computational queries.

4. Use mathematical notation to explain the possible meanings of the following sentences – and any ambiguities that may arise.
   Test Haskell interpretations of your queries in our situation.

   (a) "Somebody loves everybody; the Pope loves everybody."
   (b) "Somebody loves Flo; somebody loves everybody."
   (c) "Nobody loves Flo."
   (d) "Nobody loves everybody."
   (e) "Every philosopher has two loves."
   (f) "Some artist has only one love."
   (g) "Some politicians are scientists."
   (h) "No artist loves a politician."
   (i) "Every artist loves a scientist."
   (j) "Everyone loved by an artist loves a scientist."
   (k) "No triangle includes an artist and a politician."

(l) "Some triangle includes two happy scientists and an artist."

When giving formal interpretations, you can often save some typing, and make your code clearer by using Haskell to introduce abbreviations. For example, we can define:

```
couple x y = (x `loves` y) && (y `loves` x)
triangle x y z =  (x `loves` y) && (y `loves` z)  && (z `loves` x)
```

Does every couple form part of a triangle?

$$\forall x, y.\ C(x, y) \rightarrow \exists z.\ T(x, y, z) \vee T(y, x, z)$$

```
and[ couple x y ==>
   or [ triangle x y z || triangle y x z
        | z <- people]
   | x <- people, y <- people]
```

Six degrees of separation is the idea that everyone is connected to everyone else in the world, so that a chain of at most six "a friend, of a friend, of a friend, ..." statements are needed to connect any two people in a maximum of six steps. *WIkipedia*

5. How many degrees of separation are there in our example?
   (a) We say z is a friend-of-a-friend (`foaf`) of a iff $\exists b.\ a$ **loves** $b \wedge b$ **loves** z. Translate this into Haskell and test to see whether everyone is a friend of a friend.

   ```
   foaf :: Person -> Person -> Bool
   foaf = undefined
   test = and [ foaf a z | a <- people, z <- people]
   ```

   (b) Write a definiton for `foafoaf` (friend-of-a-friend-of-a-friend); test this; iterate until you have discovered the maximum degree of separation for our population.

Food for thought

We will not go deeply into this here, but the friend network you have been examining is just a directed graph, quite like the transition graphs for DFA we examined in Tutorial 2.

6. Suppose were to represent our dfa as graphs with a transition *relation* `tau` (τ) (instead of the transition function `delta` δ): `tau st sy st' = delta st sy == st'`

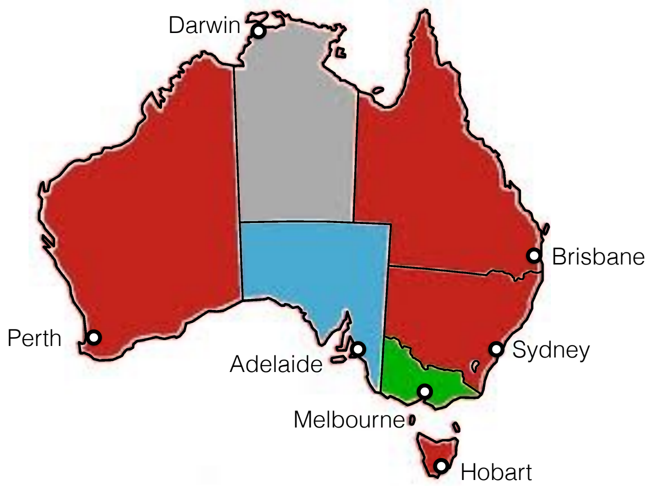
```
-- DFA another representation
type Trans state symbol =
  state -> symbol -> state -> Bool
type DFA state symbol = (
    [state]             -- k
  , [symbol]            -- sigma
  , state               -- q0
  , Trans state symbol -- tau
  , state -> Bool       -- f (accepting?)
  )
```

Complete the definition of the function `acceptsCats` that checks whether a DFA accepts the string `"Cats’’`

```
acceptsCats :: DFA Char String -> Bool
acceptsCats dfa@(states, symbols, start, trans, f) =
```

3 Australia

Consider this map of Australian States. It is coloured, with four colours, but two adjacent states (states that share a common border) have the same colour. We want a better colouring.



The four-colour theorem says that any planar map can be coloured with four colours, so that no two adjacent states have the same colour. However, here we have not coloured the sea, and to extend a four-colouring of the states to include the sea we would need a fifth colour.

So, there must be a 3-colouring of the states. Of course, it is easy to find one, but we want to use this problem to make a connection with logic. We first throw away some irrelevant detail by converting this to a problem about graphs. An (directed) graph is just a binary relation E on a set N of nodes, we call this an adjacency relation. If $E(a, b)$ we say there is an edge $\langle a, b\rangle$, from a to b. We can draw a diagram to visualise the adjacency relation.[4]

For example, a triangle has three nodes and three edges, a tetrahedron has four nodes and six edges, and a cube has eight nodes and twelve edges.

7. Draw these three graphs. Can you draw them so that no edges cross each other? (If you can, they are *planar* graphs.) Can you find an example of a non-planar graph?

[4]For graph colouring the direction of the links is irrelevant. It suffices to have one link for each adjacent pair.

8. In this question you will work through a number of steps to translate the 3-colouring problem for the map of Australia to a satisfiability problem.

   (a) To present our example as a graph, let the nodes be the seven states, and let X and Y be adjacent iff they share a common border. Draw the graph.
   Use the initial letters of the state capitals as names for the seven nodes, `M,S,H,D,P,A,B`.

   (b) How many edges are there in your Australia graph?

   We can now replace the map colouring problem by the graph colouring problem. Can we colour each node so that no pair of adjacent nodes have the same colour — or equivalently, so that any two adjacent nodes have different colours?

   (c) As a first exercise, suppose we have three colours, $C = \{r, a, g\}$ (red, amber, green). Without thinking about logic (for a brief moment) can you count the number of three-colourings of your graph?
   Hint: If you arbitrarily fix the colouring of two adjacent nodes, can you extend your colouring to the entire graph?

To express the problem in logic, we need to specify two properties of a colouring:

- each city must have a colour
  $\forall x \in$ `cities`. $\exists a \in$ `colours`. `colouring` $x\ a$
- adjacent cities cannot have the same colour
  $\forall x, y \in$ `cities`. $\forall a \in$ `colours`. $\neg($ `colouring` $x\ a \ \wedge$ `colouring` $y\ a)$

(d) Consider these two requirements. Do they capture your informal understanding of the colouring problem? Notice that we do not say that we may only apply a single colour to each city.
Should we say this? What difference would it make to the possible solutions?

(e) How many distinct colourings are there of this graph?

Now you are going to represent the problem in Haskell.
We set up the problem in the file `australia.hs`

```
data City   = P | B | H | A | S | D | M deriving (Eq, Ord, Show)
data Colour = Red | Green | Amber       deriving (Eq, Ord, Show)
type Colouring = City -> Colour -> Bool
cities  = [ P, B, H, A, S, D, M ]
colours = [Red, Green, Amber]
neighbours                  :: [(City, City)]
```

The capital letters of the state capitals represent the seven australian states. `neighbours` should be a list of pairs of states that cannot have the same colour.

(f) You have to replace three occurrences of `undefined` to define the list of neighbours, and the two constraints, so that a colouring is legal iff both constraints return `True`.

```
neighbours                  :: [(City, City)]
eachCityHasAColour          :: Colouring -> Bool
adjacentCitiesNotSameColour :: Colouring -> Bool
```

Now you can use `quickCheck` to find a colouring that satisfies these two constraints. A `Colouring` is defined by the values `[ colouring x y | x < cities, y <- colours ]`. The code provided (lines 23 onwards) uses the two constraints you will have defined, to build a `quickCheck` property taking 21 Boolean arguments, that is satisfied if there is no 3-colouring of your graph. If this check fails the counterexample represents a 3-colouring.

This counter-example is printed as 21 Boolean values. You can see what they mean by matching these Booleans against the list `pairs` defined in the code.

```
pairs = [(city, colour) | city <- cities, colour <- colours]
```

Tutorial Activity[5]

1. First review your answers for the homework. Ask a tutor to help if you have any unresolved problems.

   20 m Work through your answers to questions 1 and 2 with a buddy.

   One good Karnaugh-map technique for the problems in Question 2 is to compare the Karnaugh maps for the left- and right-hand sides of the bi-implication, and then produce the Karnaugh map that has a 1 wherever the left- and right- agree, and a zero where they disgree. We will use the results for this question to produce a compact CNF satisfiability problem from any Boolean expression.

   30 m Work through your answers to questions 3–6.

   20 m Check your answers to questions 7 & 8.

   20 m Spend any remaining time on questions 1–5 below.

An explosion

In this section we use latin squares and sudoku as examples of combinatiorial problems. These problems give examples of simple constraints that are hard to satisfy because of a so-called *combinatorial explosion* in the number of cases that must be considered. In many real-life constraint satisfaction problems, the constraints may appear more complex, but understanding these relatively-simple examples will lead us to tools for solving more-general problems.

A **latin square** is an $n \times n$ array filled with n different symbols, each occurring exactly once in each row and exactly once in each column. Latin squares were known to Indian and Arabic mathematicians in the Islamic Golden Age, from around 1000 AD.

Latin squares were used in the construction of magic squares, which arrange the numbers $1..n^2$ in an $n \times n$ square such that the sum of the numbers in any row, column or main diagonal is the same — see the paper linked above for details of one such construction. Later, latin squares were known in Medieval Europe as recreational and/or mystical constructions. The modern mathematical study of latin squares was initiated by Euler.

Shall	wee	all	dye
Wee	shall	dye	all
All	dye	shall	wee
Dye	all	wee	shall

Epitaph to Hanniball Basset (d. 1709)

A 10×10 latin square.

```
[[ 1, 8, 9, 4,10, 6, 7, 2, 3, 5],
 [ 8, 9, 1,10, 3, 4, 5, 6, 7, 2],
 [ 9, 5,10, 7, 1, 2, 8, 3, 4, 6],
 [ 2,10, 4, 5, 6, 8, 9, 7, 1, 3],
 [10, 1, 2, 3, 8, 9, 6, 4, 5, 7],
 [ 5, 6, 7, 8, 9, 3,10, 1, 2, 4],
 [ 3, 4, 8, 9, 7,10, 2, 5, 6, 1],
 [ 6, 2, 5, 1, 4, 7, 3, 8, 9,10],
 [ 4, 7, 3, 6, 2, 5, 1,10, 8, 9],
 [ 7, 3, 6, 2, 5, 1, 4, 9,10, 8]]
```

[5] The numbering of the homework questions has been changed – but the questions and their ordering remain invariant.

The modern name *Latin* squares arises from Euler's use of Latin, (a, b, c, d, ...), as well as Greek, $(\alpha, \beta, \gamma, \delta, \ldots)$ symbols in a presentation (1789) to the Academy of Sciences in St. Petersburg. Latin squares were first applied in 1788, to experimental design, but their use only became routine following the publication R. A. Fisher's work, *Design of experiments* (1935). More recently, latin squares have been used in the design of error-correcting codes.

Latin square also have deep links with other areas of mathematics — see the book, Latin Squares and their Applications. For our present purposes they are just an example of something that is simple to describe, and simple to check; but hard to find.

Some large numbers

1. For each of the following questions, give a mathematical form for the answer, then use Haskell to compute the answers for `n <- [1..5]`.
   (a) How many ways are there to place n different symbols in an $n \times n$ square (using each symbol as many times as you like)?
   (b) How many ways are there to place n copies of n symbols in an $n \times n$ square (i.e using each symbol exactly n times)?

So, there are many, many ways to fill an $n \times n$ square grid with n symbols. The numbers of $n \times n$ latin squares, for $n < 7$, are 1 2 12 576 161,280 812,851,200. There are plenty of latin squares – but if you compare these numbers with those you have just calculated you will see that for $n > 4$ they are rare enough to make a random generate-and-test an impractical procedure for the discovery of any but the smallest latin squares.

Checking a square

Nevertheless, your main task for this activity is to write a test to check whether a square is latin. You will use this to test the 10×10 square given earlier and to generate a 4×4 latin square. In a future tutorial you will explore some techniques for generating larger squares.

A framework is given for you. We will represent a `Square` as a list of lists of digits, `[[Digit]]`. A `Digit` is an `Int`, to be drawn from `[1..n]`. We also use `Int`s for column and row indices, to be drawn from `[0..n-1]`.

```
-- Square
type Column = Int -- [0..n-1]
type Row    = Int -- [0..n-1]
type Digit  = Int -- [1..n]
data Square = Square [[Digit]] deriving (Show)
```

These type declarations, together with the comments setting out ranges for the variables, represent our intentions.

These intentions are realised in the code for a function that should check whether a square is latin:

```
latin :: Square -> Bool
latin (Square m) =
  let n = length m
  in
   if any ((/=n).length) m then undefined
   else
```

```
let -- define a relation p
  p c r d = ((m !! r) !! c) == d
  columns = [0..n-1]
  rows    = [0..n-1]
  digits  = [1..n]
in
undefined -- check that each digit
&&        -- occurs in each row
undefined -- and in each column
```

This code first checks that the `Square` provided is indeed square. The number `n` of lists is the number of rows; if this is a square matrix, each row should have length `n`. If any row has a different length we return undefined. The definitions for `columns`, `rows`, and `digits`, set the ranges we intended. Otherwise we define a predicate `p :: Column -> Row -> Digit -> Bool` to represent the way the saquare is filled in. The expression `p c r d` that returns `True` iff the value in column `c` and row `r` is the digit `d`. By construction for any given row and column this predicate will be true for exactly one digit. To check that the square is latin you must implement the statement to check that each digit occurs in each row: ∀`d` ∈ `digits`. ∀`r` ∈ `rows`. ∃`c` ∈ `columns`. `p c r d`. Then write the statement to check that each digit occurs in each column, and implement it, similarly.

2. Test your implementation of `latin` on a couple of latin and non-latin 3×3 squares.
3. What will your implementation of `latin sq` return if we present an $n \times n$ square with entries outside the range `1..n`?
4. The file `latin.hs` provides a generator `squareGen` for random squares. For example, `generate (squareGen 9)` produces a 9×9 square randomly filled with the digits 1..9. You can provide such squares to `quickCheck` with a call such as `quickCheck (forAll (squareGen 4) latin)`. Unsurprisingly, `quickCheck` will find a non-latin square almost immediately (in fact, normally, it will do so on its first attempt).

   You should modify the property (to the property of not being `latin`, (so that a latin square is a counter-example) to get `quickCheck` to search for a latin square. Then try using `quickCheck` to find latin squares. You will need to tell `quickCheck` to continue beyond the first 100 trials. Try something like this:

   ```
   quickCheck (withMaxSuccess (2^32)(forAll (squareGen 4)  property))
   ```

Sudoku

Sudoku is a puzzle extends the idea of a latin square with an extra twist. In the common version, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid. The puzzle is to complete the grid to find a solution. For example,

	2	6				8	1	
3			7		8			6
4				5				7
	5		1		7		9	
		3	9		5	1		
	4		3		2		5	
1				3				2
5			2		4			9
	3	8				4	6	

is solved by

7	2	6	4	9	3	8	1	5
3	1	5	7	2	8	9	4	6
4	8	9	6	5	1	2	3	7
8	5	2	1	4	7	6	9	3
6	7	3	9	8	5	1	2	4
9	4	1	3	6	2	7	5	8
1	9	4	8	3	6	5	7	2
5	6	7	2	1	4	3	8	9
2	3	8	5	7	9	4	6	1

5. How many ways are there to fill the nine 3×3 blocks so that each includes the digits 1..9?

In future tutorials you will build tools that will enable you to produce larger latin squares, colour more complex maps, and (hopefully) to solve sudoku puzzles.

This tutorial exercise sheet was written by Michael Fourman, and includes material from an earlier version by Mark McConville, Paolo Besana, Thomas French, Areti Manataki and Dave Cochran. Send comments to `michael.fourman@ed.ac.uk`