CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment 9

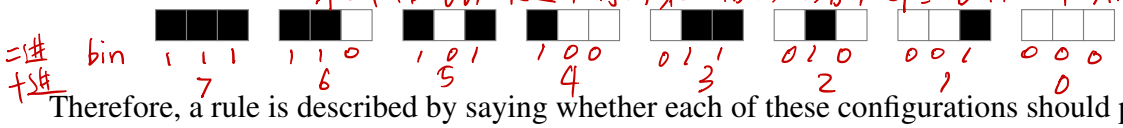
Due:	Monday, December 4th, 9:00 pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	See notes below
Files to submit:	`ca.rkt`, `rectangles.rkt`, `bonus.rkt`
Warmup exercises:	HtDP *Without using explicit recursion:* 9.5.2, 9.5.4
Practice exercises:	HtDP 20.2.4, 24.3.1, 24.3.2

- **Make sure you read the OFFICIAL A09 post on Piazza** for the answers to frequently asked questions.
- Most sub-questions will give specific instructions about what sorts of recursion are allowed (if any), and what built-in functions may be used. If no restrictions are applied, you may assume that you have full access to all forms of recursion, all techniques learned this term, and all built-in functions in Intermediate Student with Lambda.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.

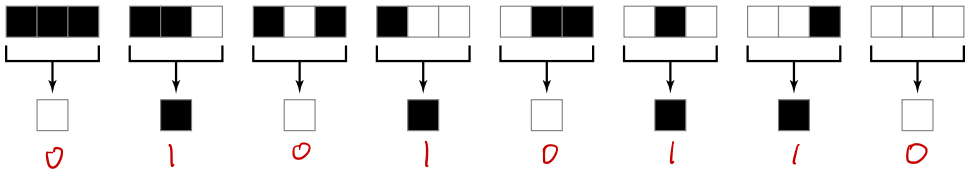
没有专门声明用哪种
方法的问题随你怎
么做，当然能用
build-in的尽量用
build-in

1. A one-dimensional elementary cellular automaton (CA) is a computational machine that evolves a row of "cells", each coloured black or white, according to a fixed "rule". In each successive generation, the rule determines the colour of a cell based on the colours of that cell and its immediate left and right neighbours in the previous generation.

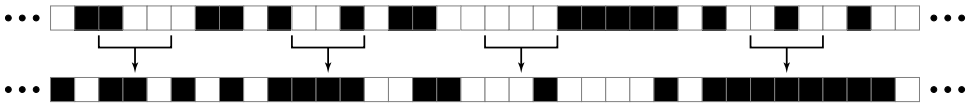
   A cell and its two neighbours can be in eight possible configurations.



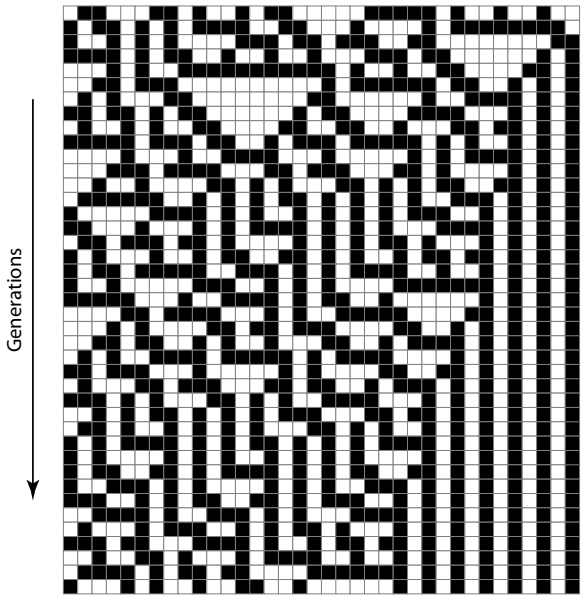
   Therefore, a rule is described by saying whether each of these configurations should produce a black or white cell in the next generation. Here is one possible rule.



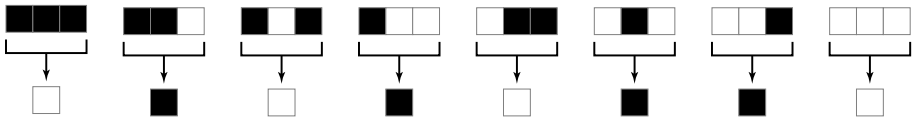
   Then, given a row of black and white cells, the rule is applied to each trio of cells to generate the cells in a new row. A few examples are highlighted below, though of course the rule is applied everywhere. (Don't worry for now about what happens at the start or end of the row; we'll work around that later.)



   We can produce interesting pictures by iterating this process. We draw a sequence of rows of black and white squares, where each row is generated by applying the rule to the row above it.



Because the rule depends on making a binary (black vs. white) choice for each of the eight possible trios of input squares, there are precisely $2^8 = 256$ possible rules. We encode a rule as a number between 0 and 255 (inclusive) by assigning the number 0 to white squares and 1 to black squares and converting the rule's outputs into a binary number, as in the diagram below, where we learn that the rule we're working with is number 86.



$$\begin{array}{cccccccc} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 2^7\times 0 & +\ 2^6\times 1 & +\ 2^5\times 0 & +\ 2^4\times 1 & +\ 2^3\times 0 & +\ 2^2\times 1 & +\ 2^1\times 1 & +\ 2^0\times 0 \\ 128\times 0 & +\ 64\times 1 & +\ 32\times 0 & +\ 16\times 1 & +\ 8\times 0 & +\ 4\times 1 & +\ 2\times 1 & +\ 1\times 0 \quad = 86 \end{array}$$

Constructing images showing the generations of a CA's operation is a prime example of generative recursion: each successive row is generated from the row before it through a general computation that doesn't follow purely from the structure of the data. In this question you will build up to a Racket function to execute a CA, producing a list of rows where each row follows from the previous one through the application of a given rule.

(a) Write a function *apply-rule* that consumes four natural numbers *a*, *b*, *c*, and *r*. The numbers *a*, *b*, and *c* are either 0 or 1, representing consecutive white and black squares in a row, from left to right. The number *r* is between 0 and 255, and encodes a CA rule as described above. The function should apply the given CA rule and produce either 0 or 1 depending on whether the resulting square should be white or black. Based on the diagram above, we would expect (*apply-rule* 1 1 1 86) ⇒ 0, (*apply-rule* 1 1 0 86) ⇒ 1, and so on.

The best approach is first to convert *a*, *b*, and *c* into a single natural number between 0 and 7, as shown below. (Hint: treat the three numbers as bits in a three-bit binary number).

7 6 5 4 3 2 1 0

If the three cell colours produce some combined number *v*, we then check whether the *v*th bit is set in *r*. Note that in Racket, one way to perform this test would be to check (*odd?* (*floor* (*/ r* (*expt* 2 *v*)))).

(b) Using the function *apply-rule*, write a *next-row* that consumes two values: a list of 0s and 1s of length at least 1, representing a row in a CA, and a rule number between 0 and 255. It produces a new list of 0s and 1s, of the same length as the input list, containing the result of applying the given rule at every position in the original list.

The consumed list does not contain enough information to determine what should happen to the first and last cells. You should assume that all cells that are not explicitly encoded in the list are white.

You may find the functions *append*, *second*, and *third* useful in this question. A reasonable approach is for *next-row* to be a small wrapper around a structurally recursive helper function.

(c) Write a higher-order function *iterate*. It consumes three values: a function f, a base value b, and a natural number n, and produces a list of length n containing $(b, f(b), f(f(b)), f(f(f(b))), \ldots, f^{(n-1)}(b))$, where each $f^{(i)}(b)$ is the result of applying f i times to b. For example, (*iterate sqr* 2 4) would produce the list '(2 4 16 256).

Your function should use explicit recursion, and not use any helper functions defined globally or in a **local**. Each element of the resulting list should be generated *from the one before it* (meaning that your function will use generative recursion). That is, don't generate $f^{(i)}(b)$ by starting from b every time and applying f to it i times; a given call to *iterate* should require only $n-1$ applications of f in total. (Hint: in your recursive call, both n and b should change.)

(d) Write a function *run-automaton*. It consumes three values: an initial non-empty row of 0s and 1s, a rule number, and a number of generations n. It produces a list of n lists, where the first sub-list is the consumed row, and each subsequent row is the result of applying the CA rule to the row before it.

Your function should be a one-liner: aside from the header of the function, it should consist of a single short line of code with no explicit recursion. Do not write any additional helper functions and do not use **local**. You might wish to use **lambda**. You may, of course, use the functions you have written for previous parts of this question.

You can write tests for this function by manually computing a few (short) rows of CA generations for a few different rules. You could also try using an online CA simulator like the one at `http://www.mattlag.com/html5/automata.html` to create tests; try choosing small values for the numbers of columns and iterations, and a larger number (around 25) for the Cell Size.

Place your solution in a file named `ca.rkt`.

Just for fun: It's natural to borrow the drawing library from Assignment 05 to turn the result of *run-automaton* into an actual image. If you want to try that, download the auxiliary file `drawinglib-a09.rkt` into the same directory as `ca.rkt` and add the line (*require* "drawinglib-a09.rkt") at the top of your solution. The file provides a function *draw-ca* that consumes two values: a list of lists, as produced by *run-automaton*, and a natural number giving the side length of a square in the grid. The function produces an image of the grid like the diagrams here (without the grey grid lines). As before, use *draw-ca* for interactive testing, but do not include any applications of that function in your submission.

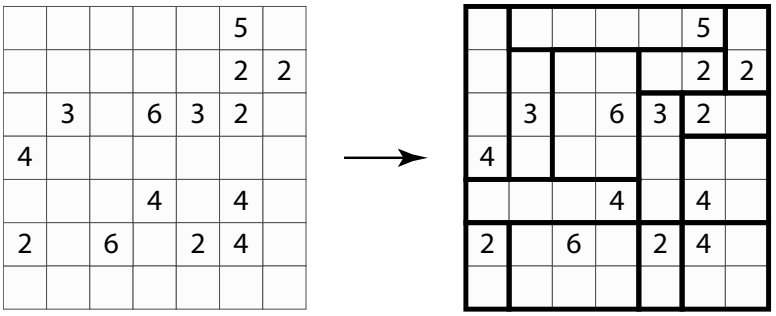
Decimal 0 in BIN is 00000000
Decimal 1 in BIN is 00000001
Decimal 2 in BIN is 00000010
Decimal 3 in BIN is 00000011
Decimal 4 in BIN is 00000100
Decimal 5 in BIN is 00000101
Decimal 6 in BIN is 00000110
Decimal 7 in BIN is 00000111
Decimal 8 in BIN is 00001000
Decimal 9 in BIN is 00001001
Decimal 10 in BIN is 00001010
Decimal 11 in BIN is 00001011
Decimal 12 in BIN is 00001100
Decimal 13 in BIN is 00001101
Decimal 14 in BIN is 00001110
Decimal 15 in BIN is 00001111
Decimal 16 in BIN is 00010000
Decimal 17 in BIN is 00010001
Decimal 18 in BIN is 00010010
Decimal 19 in BIN is 00010011
Decimal 20 in BIN is 00010100
Decimal 21 in BIN is 00010101
Decimal 22 in BIN is 00010110
Decimal 23 in BIN is 00010111
Decimal 24 in BIN is 00011000
Decimal 25 in BIN is 00011001
Decimal 26 in BIN is 00011010
Decimal 27 in BIN is 00011011
Decimal 28 in BIN is 00011100
Decimal 29 in BIN is 00011101
Decimal 30 in BIN is 00011110
Decimal 31 in BIN is 00011111
Decimal 32 in BIN is 00100000
Decimal 33 in BIN is 00100001
Decimal 34 in BIN is 00100010
Decimal 35 in BIN is 00100011
Decimal 36 in BIN is 00100100
Decimal 37 in BIN is 00100101
Decimal 38 in BIN is 00100110
Decimal 39 in BIN is 00100111
Decimal 40 in BIN is 00101000
Decimal 41 in BIN is 00101001
Decimal 42 in BIN is 00101010
Decimal 43 in BIN is 00101011
Decimal 44 in BIN is 00101100
Decimal 45 in BIN is 00101101
Decimal 46 in BIN is 00101110
Decimal 47 in BIN is 00101111
Decimal 48 in BIN is 00110000
Decimal 49 in BIN is 00110001
Decimal 50 in BIN is 00110010
Decimal 51 in BIN is 00110011
Decimal 52 in BIN is 00110100
Decimal 53 in BIN is 00110101
Decimal 54 in BIN is 00110110
Decimal 55 in BIN is 00110111
Decimal 56 in BIN is 00111000
Decimal 57 in BIN is 00111001
Decimal 58 in BIN is 00111010
Decimal 59 in BIN is 00111011
Decimal 60 in BIN is 00111100
Decimal 61 in BIN is 00111101
Decimal 62 in BIN is 00111110
Decimal 63 in BIN is 00111111
Decimal 64 in BIN is 01000000
Decimal 65 in BIN is 01000001
Decimal 66 in BIN is 01000010
Decimal 67 in BIN is 01000011
Decimal 68 in BIN is 01000100
Decimal 69 in BIN is 01000101
Decimal 70 in BIN is 01000110
Decimal 71 in BIN is 01000111
Decimal 72 in BIN is 01001000
Decimal 73 in BIN is 01001001
Decimal 74 in BIN is 01001010
Decimal 75 in BIN is 01001011
Decimal 76 in BIN is 01001100
Decimal 77 in BIN is 01001101
Decimal 78 in BIN is 01001110
Decimal 79 in BIN is 01001111
Decimal 80 in BIN is 01010000
Decimal 81 in BIN is 01010001
Decimal 82 in BIN is 01010010
Decimal 83 in BIN is 01010011
Decimal 84 in BIN is 01010100
Decimal 85 in BIN is 01010101
Decimal 86 in BIN is 01010110
Decimal 87 in BIN is 01010111
Decimal 88 in BIN is 01011000
Decimal 89 in BIN is 01011001
Decimal 90 in BIN is 01011010
Decimal 91 in BIN is 01011011
Decimal 92 in BIN is 01011100
Decimal 93 in BIN is 01011101
Decimal 94 in BIN is 01011110
Decimal 95 in BIN is 01011111
Decimal 96 in BIN is 01100000
Decimal 97 in BIN is 01100001
Decimal 98 in BIN is 01100010
Decimal 99 in BIN is 01100011
Decimal 100 in BIN is 01100100
Decimal 101 in BIN is 01100101
Decimal 102 in BIN is 01100110
Decimal 103 in BIN is 01100111
Decimal 104 in BIN is 01101000
Decimal 105 in BIN is 01101001
Decimal 106 in BIN is 01101010
Decimal 107 in BIN is 01101011
Decimal 108 in BIN is 01101100
Decimal 109 in BIN is 01101101
Decimal 110 in BIN is 01101110
Decimal 111 in BIN is 01101111
Decimal 112 in BIN is 01110000
Decimal 113 in BIN is 01110001
Decimal 114 in BIN is 01110010
Decimal 115 in BIN is 01110011
Decimal 116 in BIN is 01110100
Decimal 117 in BIN is 01110101
Decimal 118 in BIN is 01110110
Decimal 119 in BIN is 01110111
Decimal 120 in BIN is 01111000
Decimal 121 in BIN is 01111001
Decimal 122 in BIN is 01111010
Decimal 123 in BIN is 01111011
Decimal 124 in BIN is 01111100
Decimal 125 in BIN is 01111101
Decimal 126 in BIN is 01111110
Decimal 127 in BIN is 01111111
Decimal 128 in BIN is 10000000
Decimal 129 in BIN is 10000001
Decimal 130 in BIN is 10000010
Decimal 131 in BIN is 10000011
Decimal 132 in BIN is 10000100
Decimal 133 in BIN is 10000101
Decimal 134 in BIN is 10000110
Decimal 135 in BIN is 10000111
Decimal 136 in BIN is 10001000
Decimal 137 in BIN is 10001001
Decimal 138 in BIN is 10001010
Decimal 139 in BIN is 10001011
Decimal 140 in BIN is 10001100
Decimal 141 in BIN is 10001101
Decimal 142 in BIN is 10001110
Decimal 143 in BIN is 10001111
Decimal 144 in BIN is 10010000
Decimal 145 in BIN is 10010001
Decimal 146 in BIN is 10010010
Decimal 147 in BIN is 10010011
Decimal 148 in BIN is 10010100
Decimal 149 in BIN is 10010101
Decimal 150 in BIN is 10010110
Decimal 151 in BIN is 10010111
Decimal 152 in BIN is 10011000
Decimal 153 in BIN is 10011001
Decimal 154 in BIN is 10011010
Decimal 155 in BIN is 10011011
Decimal 156 in BIN is 10011100
Decimal 157 in BIN is 10011101
Decimal 158 in BIN is 10011110
Decimal 159 in BIN is 10011111
Decimal 160 in BIN is 10100000
Decimal 161 in BIN is 10100001
Decimal 162 in BIN is 10100010
Decimal 163 in BIN is 10100011
Decimal 164 in BIN is 10100100
Decimal 165 in BIN is 10100101
Decimal 166 in BIN is 10100110
Decimal 167 in BIN is 10100111
Decimal 168 in BIN is 10101000
Decimal 169 in BIN is 10101001
Decimal 170 in BIN is 10101010
Decimal 171 in BIN is 10101011
Decimal 172 in BIN is 10101100
Decimal 173 in BIN is 10101101
Decimal 174 in BIN is 10101110
Decimal 175 in BIN is 10101111
Decimal 176 in BIN is 10110000
Decimal 177 in BIN is 10110001
Decimal 178 in BIN is 10110010
Decimal 179 in BIN is 10110011
Decimal 180 in BIN is 10110100
Decimal 181 in BIN is 10110101
Decimal 182 in BIN is 10110110
Decimal 183 in BIN is 10110111
Decimal 184 in BIN is 10111000
Decimal 185 in BIN is 10111001
Decimal 186 in BIN is 10111010
Decimal 187 in BIN is 10111011
Decimal 188 in BIN is 10111100
Decimal 189 in BIN is 10111101
Decimal 190 in BIN is 10111110
Decimal 191 in BIN is 10111111
Decimal 192 in BIN is 11000000
Decimal 193 in BIN is 11000001
Decimal 194 in BIN is 11000010
Decimal 195 in BIN is 11000011
Decimal 196 in BIN is 11000100
Decimal 197 in BIN is 11000101
Decimal 198 in BIN is 11000110
Decimal 199 in BIN is 11000111
Decimal 200 in BIN is 11001000
Decimal 201 in BIN is 11001001
Decimal 202 in BIN is 11001010
Decimal 203 in BIN is 11001011
Decimal 204 in BIN is 11001100
Decimal 205 in BIN is 11001101
Decimal 206 in BIN is 11001110
Decimal 207 in BIN is 11001111
Decimal 208 in BIN is 11010000
Decimal 209 in BIN is 11010001
Decimal 210 in BIN is 11010010
Decimal 211 in BIN is 11010011
Decimal 212 in BIN is 11010100
Decimal 213 in BIN is 11010101
Decimal 214 in BIN is 11010110
Decimal 215 in BIN is 11010111
Decimal 216 in BIN is 11011000
Decimal 217 in BIN is 11011001
Decimal 218 in BIN is 11011010
Decimal 219 in BIN is 11011011
Decimal 220 in BIN is 11011100
Decimal 221 in BIN is 11011101
Decimal 222 in BIN is 11011110
Decimal 223 in BIN is 11011111
Decimal 224 in BIN is 11100000
Decimal 225 in BIN is 11100001
Decimal 226 in BIN is 11100010
Decimal 227 in BIN is 11100011
Decimal 228 in BIN is 11100100
Decimal 229 in BIN is 11100101
Decimal 230 in BIN is 11100110
Decimal 231 in BIN is 11100111
Decimal 232 in BIN is 11101000
Decimal 233 in BIN is 11101001
Decimal 234 in BIN is 11101010
Decimal 235 in BIN is 11101011
Decimal 236 in BIN is 11101100
Decimal 237 in BIN is 11101101
Decimal 238 in BIN is 11101110
Decimal 239 in BIN is 11101111
Decimal 240 in BIN is 11110000
Decimal 241 in BIN is 11110001
Decimal 242 in BIN is 11110010
Decimal 243 in BIN is 11110011
Decimal 244 in BIN is 11110100
Decimal 245 in BIN is 11110101
Decimal 246 in BIN is 11110110
Decimal 247 in BIN is 11110111
Decimal 248 in BIN is 11111000
Decimal 249 in BIN is 11111001
Decimal 250 in BIN is 11111010
Decimal 251 in BIN is 11111011
Decimal 252 in BIN is 11111100
Decimal 253 in BIN is 11111101
Decimal 254 in BIN is 11111110
Decimal 255 in BIN is 11111111

rule的所有可能
给你列出来了

2. A *rectangle puzzle* is a simple logic puzzle played on a grid of squares, where some of the squares contain positive integers. The goal is to divide the grid into rectangles so that

   - Every square in the grid belongs to exactly one rectangle
   - Every rectangle contains exactly one numbered cell, and the number is equal to the area of the rectangle.

   Below, we see an initial puzzle grid on the left, and the corresponding solution on the right. You may wish you build up your intuition for these puzzles by printing the puzzle on the left, and solving it yourself with a pencil. You can also try Rectangle puzzles online at http://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/rect.html.

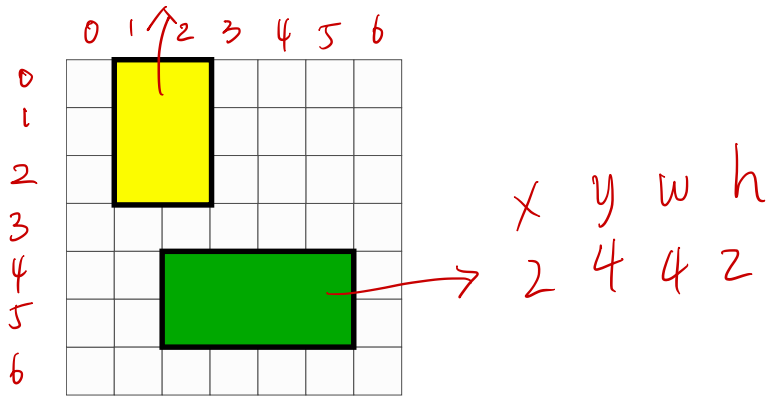


   In this question, you will develop Racket functions that can consume an empty rectangle puzzle and find the list of rectangles that solve it. In order to do so, let us develop a few necessary data definitions to talk about the grid, its cells, and rectangles.

```
(define-struct cell (num used?))
;; A Cell is a (make-cell Nat Bool)
;; A Grid is a (listof (listof Cell))
;; requires: the grid contains a non-empty list of non-empty lists,
;; all the same length.
(define-struct rect (x y w h))
;; A Rect is a (make-rect Nat Nat Nat Nat)
(define-struct state (grid rects))
;; A State is a (make-state Grid (listof Rect))
```

   A *Cell* (not at all related to the cells in the cellular automaton question!) holds information about one grid square: the number written in the square (we use 0 to denote that the square does not have a number in it) and whether that square currently belongs to any rectangle.

   A *Rect* describes a rectangle whose top-left corner is given by the coordinates (x, y), and extends to the right by width w and downward by height h. The top-left corner of the grid is considered to be at $(0, 0)$. For example, in the diagram below, the yellow rectangle is defined by (*make-rect* 1 0 2 3), and the green rectangle by (*make-rect* 2 4 4 2).



To begin, download the files `rectanglelib.rkt` and `rectangle.rkt`. Immediately modify the header in `rectangle.rkt` with your name and student ID. Then write the functions below.

(a) Write a higher-order Racket function *map2d* that behaves like the two-dimensional analogue of *map*. It consumes a function *f* and a list of lists of values, and produces a new list of lists in which *f* has been applied to every element of the input. For example, (*map2d add1* '((3 4 5) (10 9 8))) would produce '((4 5 6) (11 10 9)).

Do not use any explicit recursion or helper functions to write *map2d*. Instead, use a short one-line function body based on two uses of *map* and a single **lambda**.

(b) Write a function *construct-puzzle* that consumes a (*listof* (*listof Nat*)) and produces a *State* representing the initial state of a puzzle. Your function should use *map2d* to create the initial *Grid*, and the initial *State* should have no rectangles in it.

(c) Write a predicate *solved?* that consumes a *State* and produces a boolean value that indicates whether the puzzle described by the state is fully solved. We consider a puzzle to be solved if and only if all of its cells are used. (This isn't ideal, in that the puzzle could be lying to us—perhaps we simply marked all the cells as used without bothering to find any rectangles. But we'll assume that we're honest, and that if we mark a cell as used, it's because we put a rectangle in the state's list of rectangles that uses it.)

(d) Write a function *get-first-unused* that consumes a *Grid* and finds the topmost, leftmost cell in the grid that isn't marked as used. That is, the cell you find should be in the topmost row that contains unused cells, and it should be the leftmost unused cell in that row. The function should produce a (*list Nat Nat*) containing the *x* and *y* coordinates of the first unused cell. You can assume that if this function is being called, the grid contains at least one unused cell.

(e) Write a function *neighbours* that consumes a *State* and produces a list of new states that might legitimately follow from the given state after adding a single new rectangle. If no legal rectangles can be added from the current state, produce the empty list.

To find a state's neighbours, first find the topmost, left unused cell in the grid. Construct all rectangles that fit within the bounds of the grid and that have the unused cell at their top-left corners. Now identify the subset of those rectangles that are legal in

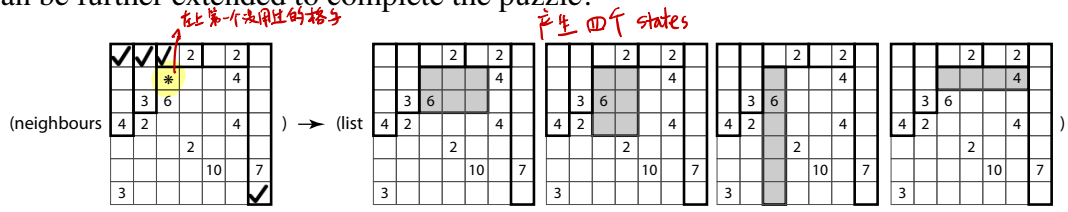
这一问就是这个作业最难的一个问题了，通过当前 state，找到所有下一个可能的 states，注意，我们会用左上开始第一个出现的没用过的格子开始找例有可能，也就是说，你要找的是所有包含左上开始第一个没用过的地方的可能的下一个 state

the puzzle: rectangles that do not contain any already used cells, and that contain a single numbered cell, where the number equals the area of the rectangle. For each such rectangle, construct a new neighbour state in which all cells inside the rectangle have been marked as used, and where the rectangle itself has been added to the list of the state's rectangles (making up a potential partial solution to the puzzle). You can output these neighbour states in any order.

每个产生的可能的 state 还要把对应格子里面 used? 变成 true，同时要加一个新的 rect 到 state 里面

这道题你很难一气做出来，你肯定需要若干个 helpers，这样可以把你的 logic 拆开，容易一步步做，一定要认真分析你需要哪几步，方法有很多但是 local functions 和 local constant 都会对你很有帮助

The diagram below visualizes an example of computing a state's neighbours. The marked cell is the leftmost, topmost unused cell in the source state; the neighbours are the four states on the right with the new rectangles shaded. Only one of those neighbours can be further extended to complete the puzzle!



This step is, by far, the most complicated one in this question. It's likely your solution will depend on a number of helper functions, which you should use liberally, either at the top level or inside a **local** (or both).

(f) Now let's put it all together. Write a function *solve-rectangle-puzzle* that consumes a (*listof* (*listof Nat*)) describing an initial puzzle. It attempts to solve the puzzle, producing either the list of rectangles that describe a solution if one exists, or *false* if no solution can be found.

Your solution must perform a graph search with backtracking on an implicit graph, as suggested in Module 12 of the course notes. In `rectanglelib.rkt` we provide a higher-order function *search* that performs the backtracking search for you; all you have to do is plug in the initial state to search from, and functions to enumerate a state's neighbours and determine whether a given state is a goal state, all of which are solutions to previous sub-questions. This function can therefore be fairly short.

Place your solution in a file named `rectangle.rkt`.

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

---

3. **5% Bonus**:

   In a classic math puzzle, a father dies leaving behind an estate of 35 camels. His will asks to divide the camels among his three children so that the eldest child gets $1/2$ of the camels, the middle child gets $1/3$, and the youngest gets $1/9$. The children are stuck: they don't want to resort to messy fractional camels!

The town sage devises a solution. She borrows a camel to bring the total to 36, and then distributes $36/2 = 18$, $36/3 = 12$, and $36/9 = 4$ camels to the three children. They receive 34 camels in total; the sage returns the borrowed camel and keeps the final remaining animal as a fee, leaving everyone happy.

Let's ignore the borrowing step, and simply say that 36 camels can be divided into fractions of $1/2$, $1/3$, and $1/9$, with two camels left over. Other combinations of numbers yield similar division problems; for example, 54 camels can be divided into fractions of $1/2$, $1/3$, and $1/9$ with three camels left over ($54/2 + 54/3 + 54/9 + 3 = 54$). We can also extend the problem to any number of children; for example, with five children, 1320 camels can be divided into $1/3$, $1/4$, $1/5$, $1/8$, and $1/11$ with one camel left over.

Write a function *camels* that produces similar division equations. Specifically, the function consumes two *Nat*s: the number of brothers ($k > 0$), and the number of camels that should be left over at the end of the division process ($r > 0$). The function should find *all* sets of natural numbers $t_1, t_2, \ldots, t_k$ and N that satisfy:

$$\frac{N}{t_1} + \frac{N}{t_2} + \ldots + \frac{N}{t_k} + r = N$$

$$t_1 \leq t_2 \leq \ldots \leq t_k$$

$$N \text{ is a multiple of } t_i \text{ for all } i$$

Your function should produce a (*listof* (*listof Nat*)), where each sublist is of the form (*list* t_1 t_2 … t_k N). So, for example, the following might be a valid test:

```
(check-expect (camels 2 1) '((2 3 6) (2 4 4) (3 3 3)))
```

Because $6/2 + 6/3 + 1 = 6$, $4/2 + 4/4 + 1 = 4$, $3/3 + 3/3 + 1 = 3$, and there are no other valid equations of this type.

Submit your solution in the file `bonus.rkt`.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Consider the function (*euclid-gcd*) from slide 7-16. Let f_n be the nth Fibonacci number. Show that if $u = f_{n+1}$ and $v = f_n$, then (*euclid-gcd u v*) has depth of recursion n. Conversely, show that if (*euclid-gcd u v*) has depth of recursion n, and $u > v$, then $u \geq f_{n+1}$ and $v \geq f_n$. This shows that in the worst case the Euclidean GCD algorithm has depth of recursion proportional to the logarithm of its smaller input, since f_n is approximately ϕ^n, where ϕ is about 1.618.

You can now write functions which implement the RSA encryption method (since Racket supports unbounded integers). In Math 135 you will see fast modular exponentiation (computing $m^e \bmod t$). For primality testing, you can implement the little Fermat test, which rejects numbers for which

$a^{n-1} \not\equiv 1 \pmod{n}$, but it lets through some composites. If you want to be sure, you can implement the Solovay–Strassen test. If $n-1 = 2^d m$, where m is odd, then we can compute $a^m \pmod{n}$, $a^{2m} \pmod{n}, \ldots, a^{n-1} \pmod{n}$. If this sequence does not contain 1, or if the number which precedes the first 1 in this sequence is not -1, then n is not prime. If n is not prime, this test is guaranteed to work for at least half the numbers $a \in \{1, \ldots, n-1\}$.

Of course, both these tests are probabilistic; you need to choose random a. If you want to run them for a large modulus n, you will have to generate large random integers, and the built-in function *random* only takes arguments up to 4294967087. So there is a bit more work to be done here.

For a real challenge, use Google to find out about the AKS Primality Test, a deterministic polynomial-time algorithm for primality testing, and implement that.

Continuing with the math theme, you can implement the extended Euclidean algorithm: that is, compute integers a, b such that $am + bn = \gcd(m, n)$, and the algorithm implicit in the proof of the Chinese Remainder Theorem: that is, given a list $(a_1, \ldots, a_n)$ of residues and a list $(m_1, \ldots, m_n)$ of relatively coprime moduli ($\gcd(m_i, m_j) = 1$ for $1 \leq i < j \leq n$), find the unique natural number $x < m_1 \cdots m_n$ (if it exists) such that $x \equiv a_i \pmod{m_i}$ for $i = 1, \ldots, n$.