Assignment: 10
Due: Tuesday, April 3rd, 9:00 pm
Language level: Intermediate Student with Lambda
Allowed recursion: See notes below
Files to submit: binarypuzzle.rkt
Warmup exercises: HtDP 28.1.6, 28.2.1, 30.1.1, 31.3.1, 31.3.3, 31.3.6, 32.3.1
Practice exercises: HtDP 28.1.4, 28.2.2, 28.2.3, 28.2.4, 31.3.7, 32.3.2, 32.3.3

- Make sure you read the OFFICIAL A10 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 and special forms in Intermediate Student with Lambda that have been covered in the course.

This includes abstract list functions map, filter, foldr, build-list, andmap, ormap, quicksort, and foldl.

You may also use reverse.

- All helper functions must be defined locally or written as lambda expressions.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide. However, there are no marks assigned to your own testing for this assignment. Although you are not required to include full test suites in your files, you are encouraged to do your own testing as you develop your solutions.
1. In this question you will be writing functions that will help you solve a binary puzzle. A binary puzzle is an $n \times n$ grid. The grid’s dimensions ($n$) will always be a positive even number. Here are the rules of a solution for a binary puzzle as described at binarypuzzle.com

- Each box of the grid must contain a zero or a one.
- No more than two of the same numbers may appear next to or below each other in the puzzle. In other words, you cannot have three or more of the same number in adjacent positions in any row or column.
- Each row and each column must contain an equal number of zeros and ones.
- Each row is unique relative to other rows and each column is unique relative to other columns.

If you want to solve a binary puzzle, you will be given a starting grid that has some of the values filled in with randomly placed zeros and ones. You must logically determine how to fill in the rest of the puzzle. Each puzzle has exactly one solution. For example, if you started with this grid

```
1 0 0 1
0 0 1 0
0 0 1 0
1 0 0 0
```

the solution would be

```
1 0 1 0 1 0
0 1 0 0 1 1
1 0 0 1 0 1
0 1 1 0 1 0
0 0 1 1 0 1
1 1 0 1 0 0
```

There are many puzzles you can try yourself at binarypuzzle.com. This is also a source of test data.

You will be working with the following data definition:

```scheme
;; A BinaryPuzzle is a (listof Str)
;; requires:
;;  the length of the list equals the length of each of the strings
;;  the length of the strings in the list is a positive even number
;;  the characters in the strings are (anyof #\0 #\1 #\-)
;;  the #\- character represents a position in the puzzle that is unfilled
```
For example, the starting puzzle shown previously could be represented as follows:

\[
\text{(define \textit{start-6x6}}
\begin{array}{c}
\text{call} \text{list} \\
\text{"1--0--"} \\
\text{"--00-1"} \\
\text{"-00--1"} \\
\text{"------"} \\
\text{"00-1--"} \\
\text{"-1--00"})
\end{array}
\]

You will be writing several functions that will help you solve binary puzzles. Any function you are required to write can be used as a helper function in any other part of the question. You have been provided a starting file \texttt{binarypuzzle.rkt} that contains several constants representing starting puzzles and solutions for those puzzles. Some of the puzzles are easy to solve, and some of them are very hard to solve.

(a) Write the function \texttt{line-done?} that consumes a string that contains only the characters \#\(\text{0}\), \#\(\text{1}\), and \#\(-\). The length of the string is a positive, even number. The function produces \texttt{true} if it is a string representing a completed, valid row or column of a binary puzzle, and produces \texttt{false} otherwise. In other words, a line is complete and valid if it contains an equal number of zeros and ones, there are no unfilled spots, and there is not a sequence of three or more adjacent characters that are the same.

There are no restrictions on use of recursion or abstract list functions in this part of the question.

(b) Write the function \texttt{transpose} that consumes a \texttt{BinaryPuzzle} and produces a \texttt{BinaryPuzzle} that has the columns of the puzzle consumed appear as rows in the puzzle produced. For example, if you consumed the puzzle \texttt{start-6x6} the function would produce

\[
\text{(list "1---\text{0}-"} \\
\text{"--\text{0}-\text{0}1"} \\
\text{"-\text{0}0--\text{1}"} \\
\text{"00--1--"} \\
\text{"-----\text{0}"} \\
\text{"-1-00"})
\]

You may not use explicit recursion in this part of the question. The function \texttt{build-list} may be helpful here.

(c) Write the function \texttt{puzzle-done?} that consumes a \texttt{BinaryPuzzle} and produces \texttt{true} if the puzzle is completely filled with zeros and ones and meets all of the requirements of a binary puzzle, and produces \texttt{false} otherwise.

You may not use explicit recursion in this part of the question.

(d) One approach to solving puzzles is to make a random guess at a partial solution, and follow it through to see if the guess leads to a correct solution. The next three functions facilitate that approach.
Write the function **find-unfilled** that consumes a **BinaryPuzzle** and produces a two element list representing the location of the first unfilled spot. The first element of the list represents the row and the second element of the list represents the column. The first unfilled spot appears at the leftmost position of the topmost row that contains an unfilled spot in the puzzle. For example the first unfilled spot in `start-6x6` is row 0 and column 1. So if `find-unfilled` consumed `start-6x6` it would produce `'(0 1)`. If there are no unfilled spots, the function should produce `'(1 -1).

There are no restrictions on use of recursion or abstract list functions in this part of the question.

(e) Write the function **try** that consumes a **BinaryPuzzle**, a natural number representing the row position of the first unfilled spot in the puzzle, a natural number representing the column position of the first unfilled spot in the puzzle, and a string that is either "0" or "1", in that order. You may assume that the **BinaryPuzzle** consumed contains at least one unfilled spot. The function should produce a **BinaryPuzzle** that is a copy of the puzzle consumed, except that the unfilled spot indicated by the natural numbers is filled with the string that is consumed. For example, `try start-6x6 0 1 "0")` produces:

```
(list "10-0--"  
"--00-1"  
"-00--1"  
"------"  
"00-1--"  
"-1--00")
```

There are no restrictions on use of recursion or abstract list functions in this part of the question.

(f) Write the function **neighbours** that consumes a **BinaryPuzzle** and produces a list containing **BinaryPuzzles** that are the next attempt to find a solution. If there are no empty spots in the puzzle consumed, the function should produce empty. Otherwise, the function should produce a two element list where the first element is a copy of the puzzle consumed with a "0" filled in the first unfilled spot, and the second element is a copy of the puzzle consumed with a "1" in the first unfilled spot.

This function should not include any recursion.

(g) There are many strategies that can be used to logically fill in some or all of a binary puzzle, rather than making random guesses. For the next two functions you will be using a specific set of logical rules to help find a partial or full solution for a puzzle. In this part, you will be using the following logic to fill in unfilled spots from a single row or column of the puzzle.

- If you have already identified half of the values in the a particular row or column as being a 0, then you know each of the remaining unfilled spaces must be filled with a 1.
- Similarly, if you have already identified half of the values in the a particular row or column as being a 1, then you know each of the remaining unfilled spaces must be
filled with a 0.

- If you have two of the same character adjacent to each other, then you know that an unfilled spot on either side of those adjacent values must be the opposite number. For example, if you see "--00--" you can fill in two of the unfilled spots and end up with "-1001-".

- If you have two of the same character separated by a single unfilled spot, then you know that unfilled spot must be the opposite number. For example, if you see "--1-1-" you can fill in the unfilled spot in between the two ones and end up with "--101-".

- Once you have filled in some of the unfilled spots according to the previous rules, you may now be able to revisit the string and fill in some other spaces. For example, if you start with "0--001-1", then by the previous rules we can fill in some of the unfilled spots and end up with "0-100101". But now you can use the logical strategy that recognizes that since you have already filled in half the spots with a 0, then each of the remaining spots (in this case just one spot) must be filled with a 1. Thus you can fill in this line completely and end up with "01100101".

Write the function `fill-spots` that consumes a string representing a row or column from a binary puzzle. It produces the string where as many of the unfilled spots are logically filled in according to the strategies listed above.

Note that it is possible that you cannot fill in any of the unfilled spots. For example, the string "0--001-1" does not have enough information alone to fill in any of the unfilled spots. Also, there are other logical strategies that you could use when actually solving the puzzle that would allow you to fill in some unfilled spots. However, you must not use them when creating a solution for this question.

There are no restrictions on use of recursion or abstract list functions in this part of the question.

(h) Write the function `update-puzzle` that consumes a `BinaryPuzzle` and produces a `BinaryPuzzle` that has filled in as many unfilled spots as possible using the strategies described in the previous part of this question. For example, if `update-puzzle` consumes `start-6x6`, the function would produce a `BinaryPuzzle` the represents the completed puzzle described at the beginning of this question. For your own testing purposes, the Easy and Medium puzzles at binarypuzzle.com can usually be completed using the strategies of `update-puzzle`.

There are no restrictions on use of recursion or abstract list functions in this part of the question.

(i) Now it is time to write the function `solve-puzzle` that consumes a `BinaryPuzzle` and produces a completed and correct `BinaryPuzzle`.

Some puzzles are easy, and can be solved just using `update-puzzle`. However, other puzzles are hard, and we need to do more to determine the solution. If after updating the puzzle, you still have some unfilled spaces, one strategy is to try filling in a space with a 0 or a 1 and then continue trying to solve the puzzle. Then you can update the
puzzle based on this guess. If you still have not completed the puzzle, you can make
another guess. If you get to the point where you have filled in all the spaces and the
puzzle is correct, then you are done. However, if you get to the point where you have
filled in all the spaces but the puzzle is not correct, then you know a value you guessed
was incorrect. Now you have to go back and try a different guess.

This guessing technique can be implemented by trying to find a route from the starting
puzzle to the solution. You can do this by modifying the find-route algorithm from
slides 19 and 20 in Module 12. The finished puzzle is the last node in the route that
you find. Incorrect guesses will automatically lead to backtracking in the find-route
algorithm.

In the modified find-route function, the nodes of the graph are not listed explicitly;
they need to be generated by functions. The nodes of the graph are partially completed
puzzles that have been updated as much as possible using the update-puzzle function.
The neighbours of the nodes of the graph come from puzzles where only one unfilled
space has been filled in with a 0 or a 1. These puzzles are generated implicitly using the
neighbours function.

There are no restrictions on use of recursion or abstract list functions in this part of the
question. If you have properly implemented all of the other parts of this question, then
solving the puzzle is relatively straight-forward code.

Notes:

• Since we carefully choose the neighbours by always making guesses at the first
  unfilled spot in the puzzle, we do not have to worry about cycles or diamonds in
  our graph.
• All puzzles are guaranteed to have a solution.
• It might take several seconds for your solution to solve the start-10x10-vh-10
  puzzle provided to you in the starting file for this question. However, it should
  not take more than a minute if you have followed the strategies outlined in this
  assignment. Most of our tests will be based on puzzles that can be solved quickly.
  However, if your solution for solve-puzzle is very slow, you will not receive full
  correctness marks.

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.

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 \( n \)th 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 \( \phi^n \), where \( \phi \) 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 \( m^e \mod r \)). 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 \( a^m \pmod{n} \), \( a^{2m} \pmod{n} \),..., \( a^{n-1} \pmod{n} \). 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 \texttt{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 \( (\gcd(m_i, m_j) = 1 \text{ for } 1 \leq i < j \leq n) \), find the unique natural number \( x < m_1 \cdots m_n \) (if it exists) such that \( x \equiv a_i \pmod{m_i} \) for \( i = 1, \ldots, n \).