- CPS506 Project -

Polarity Placement Puzzle

Preamble

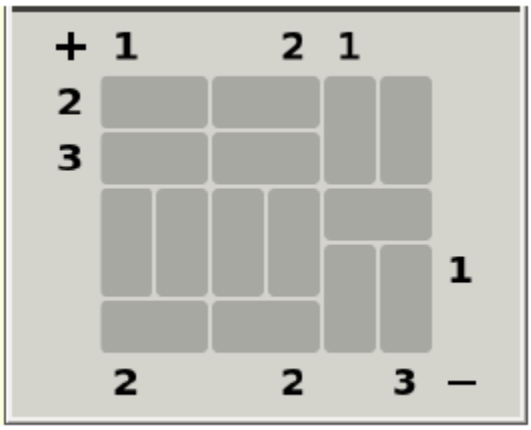
You will write a program that solves a puzzle that involves placing bar magnets on a two-dimensional board, while satisfying a set of constraints. The description for this puzzle can be found below.

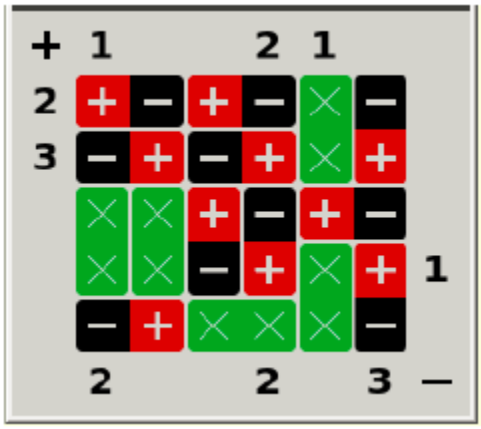
In addition to the general requirements of the project, each language comes with its own language-specific requirements. These specify the format of the input and output, as well as submission instructions for each language.

Aside from these requirements, anything you do inside your program is up to you. Use as many helper functions or methods as you want, use any syntax you find useful whether we covered it in class or not.

Puzzle Description

The polarity puzzle requires you to place domino shaped bar magnets on a board in such a way that certain row and column constraints are satisfied. This is best demonstrated with a visual example. The empty board is shown on the left, and the solved board is on the right:





On the left, the orientation of the dark gray rectangles indicates the orientation of the magnet that can be placed there.

Numbers placed along the top and left side of the board indicate the precise number of positive poles that must be placed in that column or row. Similarly, numbers along the bottom and right side of the board indicate the precise number of negative poles that must be placed in that column or row.

If a position along any side of the board does not contain a number, then there is no requirement for that particular polarity in that row or column. In the example above, the left side of the board does not contain any numbers in the bottom three rows. This means that for those rows, we can place as many positive poles as we want. This applies to the left and top for positive poles, and to the right and bottom for negative poles.

One final constraint is that when placing magnets, positive poles cannot be vertically or horizontally adjacent. The same is true of negative poles. This one is intuitive, as it mirrors the behavior of real magnets.

Analyze the solved board on the right and see how all these constraints are met.

Additional Notes:

- Any number of magnets may be placed, and any number of tiles may be left empty, as long as the constraints are met. The solved board above contains 10 magnets, and five tiles were left empty.
- There may be multiple solutions to the same board. For this project, all solutions that satisfy the requirements are equally valid and correct. Your program must return one solution, and one solution only.
- It is possible to create a board that cannot be solved. You may assume that your program will not be tested on such examples. There will always be at least one solution.

Strategy and Approach:

You may take any algorithmic approach to the puzzle that you like. A brute force approach is simple enough – generate every possible permutation of tile placements, checking the constraints on each until you find a valid solution. It's not a bad place to start, and it might work fine for small boards, but for larger boards the exponential time complexity of this approach will require the lifespan of ten trillion multiverses to compute. You will have to employ clever use of data structures and board exploration strategies to be successful on the large-scale tests.

Input and Output Format

Precise types will vary by language, but in general, there will be five inputs to your program: Four lists, one each for the top, bottom, left, and right board constraints. The fifth input will be a 2D data structure representing the board itself that indicates the orientation of the tiles (vertical or horizontal)

The output of your program will also be a 2D data structure whose contents represent the placement of magnets on the input board.

Technical Details & Language Requirements

Below are described the type constraints for input arguments and return values for each language. Note that you are not stuck with the types you are given. The arguments enter your function in one form, but there's nothing stopping you from converting it to some other form internally. Same goes for the return value. Build your solution however you like as long as you convert it to the expected type before returning.

Smalltalk:

Create a class named **`Polarity`** that implements a class method called:

```
solveWith: specs and: board
```

Parameter **specs** is passed as a Dictionary whose keys are the strings 'left', 'right', 'top', 'bottom'. The values for each of these keys are arrays of integers containing the constraints for those rows and columns. If a row or column does not have a constraint, there will be a value of -1 in that position. For the example above, the dictionary entries would appear as follows:

```
'left'   -> #(2 3 -1 -1 -1)
'right'  -> #(-1 -1 -1 1 -1)
'top'    -> #(1 -1 -1 2 1 -1)
'bottom' -> #(2 -1 -1 2 -1 3)
```

Parameter **board** is an array of strings representing the orientation of the tiles. Each row in the board is represented by a string containing the characters T, B, L, and R, standing for top, bottom, left, and right of an individual magnet placement. For example, a single magnet may only be placed over L and R cells, or T and B cells. The board in the example above would be represented as follows:

```
#( 'LRLRTT'
   'LRLRBB'
   'TTTTLR'
   'BBBBTT'
   'LRLRBB' )
```

The return value of this method should also be an array of strings, representing the magnet configuration. The characters '+' and '-' represent the poles of the magnets, and the character 'X' represents no magnet being present. The solution to the above example would be represented as:

```
#( '+-+-X-'
   '-+-+X+'
   'XX+-+-'
   'XX-+X+'
   '-+XXX-' )
```

Elixir:

You are given a mix project with a file called polarity.ex that contains the following function:

```
def polarity(board, specs) do
    # Your code here!
end
```

Parameter "board" will be represented as a tuple of strings. The example seem previously would be represented as:

```
{ "LRLRTT", "LRLRBB", "TTTTLR", "BBBBTT", "LRLRBB" }
```

The constraints will be in parameter "specs", and will be represented as a map (similar to a dictionary) of tuples. The keys will be the strings "left", "right", "top", "bottom":

```
%{ "left"=>{2, 3, -1, -1, -1},
   "right"=>{-1, -1, -1, 1, -1},
   "top"=>{1, -1, -1, 2, 1, -1},
   "bottom"=>{2, -1, -1, 2, -1, 3}
 }
```

You will return your solution as a tuple of strings, just like the board input:

```
{ "+-+-X-" , "-+-+X+", "XX+-+-", "XX-+X+", "-+XXX-" }
```

Run the tests for Elixir by using **mix test --seed 0** from the project directory. We will use the **--seed 0** flag to prevent Elixir from randomizing the test order. This is important when it comes to creating the output log file, which can be found in the tester directory after the tests are run. It is called ex_log.txt.

Haskell:

You are given a cabal project with a source file called Polarity.hs that contains the following function and type signature:

```
polarity :: [String] -> ([Int], [Int], [Int], [Int]) -> [String]
```

The above type signature indicates the parameter and return type of the function. The board is a list of Strings, as is the return value. The specs are a Tuple of **[Int]**, where each **[Int]** contains row/column constraints. The order of the constraints is left, right, top, bottom.

Rust: TBA – This document will be updated for each language as we go

Testing & Evaluation

Your code will be evaluated using an automated tester written by me. Therefore, it is of ***utmost importance*** that your code compile, handle input properly, and return results in the specified format for each language. Do not deviate from the requirements or your code will fail the tester outright. Your code must ***compile and run*** as a baseline. Half-finished code that doesn't compile or is riddled with syntax errors will not be accepted.

To help you achieve the baseline of "code that works", you will be provided with mix/cabal/cargo projects with a handful of test cases for Elixir/Haskell/Rust. For Smalltalk, you will be provided a Pharo image containing unit tests. The testers will verify that the board solution you return is legal and adheres to the constraints.

Your grade for each project will be based on the number of tests you pass within the required time limit of 30 seconds.

Marking Rubric – 50 marks total

10 marks – Test #1

This is the example given in this document. Five rows, six columns, and a known solution. Note that your code must ***find*** the solution – You may not simply hardcode it. Hardcoding a solution for any test is forbidden, and steps will be taken to identify and mitigate hardcoding when marking.

12 marks – Test #2a & Test #2b

Two very simple 2x2 boards, easy to brute force. Six marks each.

16 marks – Test #3a & Test #3b

Two 4x4 boards. Getting trickier. Eight marks each.

12 marks – Test #4a & Test #4b

Two 8x8 boards. Brute force won't work so well on these. You'll need some cleverness in how you assign the magnets to the tiles. Six marks each.

BONUS – Test #5

This is a single massive 16x16 board. Anything resembling brute force will exceed the time limit of 30 seconds.

50 marks total

When marking, slight alterations to the tests will be made to prevent hardcoding. Other than that, these are the same tests that will be used to calculate your grade.

Submission

Projects must be submitted ***individually***.

Smalltalk submission: Export your Polarity class the same way you did for Lab #2. If you wrote additional helper classes, submit those also.

Elixir/Haskell/Rust submissions: Submit your polarity.ex, polarity.hs, or main.rs file on D2L.

- Do NOT rename any files.
- Do NOT submit an archive (zip/rar/7z/whatever) containing the files.
- Do NOT submit the entire mix/cabal/cargo project.