Homework 4 — assigned April 5 — due April 16

General instructions

A skeleton Haskell source file homework4.hs will be provided with the type signatures of the functions you are to write. Please edit that file and submit. The skeleton file will start with `import` declarations for the Haskell libraries we expect you to find useful.

4.1 Genome lists (40pts)

A DNA molecule is a sequence over an alphabet of four "bases", which are conventionally represented using the characters 'A', 'G', 'C', and 'T'. Genomes represented by DNA molecules are subject to different types of point mutations:

- insertions: A base is inserted between two adjacent points in a genome.
- deletions: A point is deleted from a genome.
- substitutions: A base at a point is replaced with another base.
- transpositions: The bases at two adjacent points are exchanged.

Give definitions for Haskell functions `insertions`, `deletions`, `substitutions`, and `transpositions`, which take a genome represented as a string and return a list of all genomes produced by single-point mutations of the specifed kind. For example,

`insertions "GC"` evaluates to
`["AGC","GAC","GCA","GGC","GGC","GCG","CGC","GCC","GCC","TGC","GTC","GCT"];`

`deletions "AGCT"` evaluates to
`["GCT","ACT","AGT","AGC"];`

`substitutions "ACT"` evaluates to
`["ACT","AAT","ACA","GCT","AGT","ACG","CCT","ACT","ACC","TCT","ATT","ACT"];` and

`transpositions "GATC"` evaluates to
`["AGTC","GTAC","GACT"]`.

4.2 Sorting (20pts)

1. Define a recursive function `insert :: Ord a => a -> [a] -> [a]` that inserts an element into the correct position in a sorted list.

2. Using `insert`, define a recursive function `isort :: Ord a => [a] -> [a]` that sorts a list into the correct order using insertion sort. In this definition, the empty list is already sorted, and any non-empty list is sorted by inserting its head into the list that results from sorting its tail.

3. Using `isort`, define `fileisort :: String -> String -> IO ()`, such that executing the action `fileisort fn1 fn2` causes the external file named `fn2` to become a line-by-line sorted version of the file `fn1`. (In other words, it has the same effect as the Unix command `sort fn1 >fn2`.) You can assume that the names `fn1` and `fn2` refer to distinct files.

4.3 Game trees (40pts)

We can represent a tic-tac-toe board as a list of fields, thus:

```
data Field = B | R | G
             deriving (Eq, Ord, Show)
type Board = [Field]
```

with the convention that `B` stands for an unoccupied field, `R` stands for a field occupied by the first player ("red"), and `G` stands for a field occupied by the second player ("green"); a board is a list with 9 fields in row-major order. Thus the board

```
BRG
BRG
BRB
```

(the final board of a game that ended in red's victory) is represented as `[B,R,G,B,R,G,B,R,B]`.

It is possible to play this game perfectly, that is, to prevent the opponent from winning. Write two functions, `strategyForRed, strategyForGreen :: Board -> Int`, that implement a perfect strategy for red and green, respectively.

That is, given a board position `b`, the result of `strategyForRed b` is a number between 0 and 8, namely the index of the field into which red should move when executing the chosen strategy; the function only needs to be defined for those board positions that are actually reachable if the strategy is followed by red—but the opponent is always free to make any legal move.

As part of your solution, you will need to provide evidence that you have comprehensively tested your code. This means that you have verified that each strategy function always produces valid moves and that it guarantees not losing the game, no matter how the opponent plays.

You may freely read, use, and/or adapt the tic-tac-toe code provided in the textbook (Chapter 11).

4.4 (Optional) Drawing game trees and strategies (30pts of extra credit)

There are multiple ways in which useful diagrams can be drawn for games and their strategies.

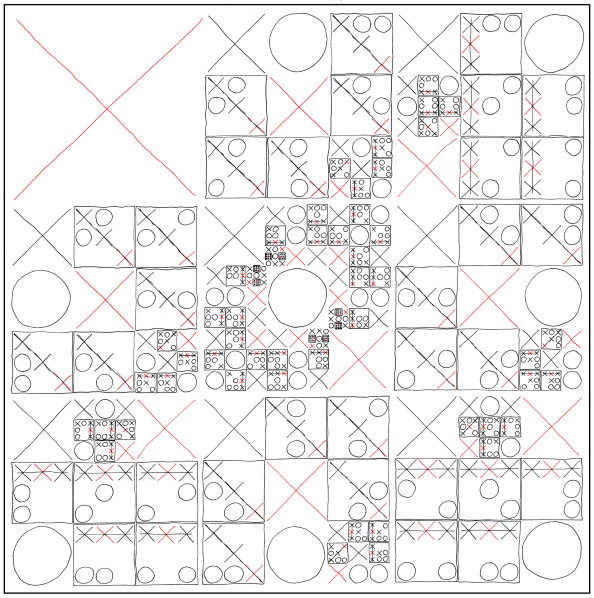
One kind of diagram is a game tree, as shown in the textbook, p. 146, showing all possible game developments from a given starting point (commonly the empty board).

Another kind of diagram is a strategy tree, showing only those paths allowed by the chosen strategy for a player. For tic-tac-toe, this tree can also be drawn in a vivid format like this, from `https://xkcd.com/832_large/`:

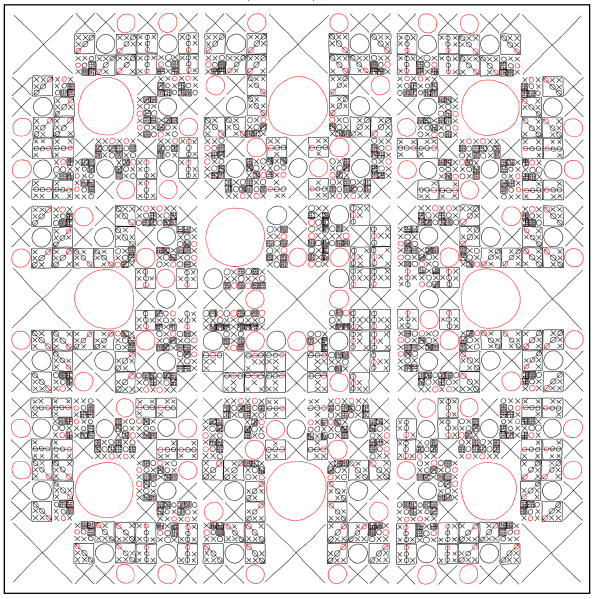
COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

MAP FOR X:



MAP FOR O:



The task is to draw your strategies (`strategyForRed` and `strategyForGreen`) from the preceding exercise. Write a Haskell function `drawStrategy :: Bool -> String -> IO ()`, which takes a boolean that says which strategy to draw (true for Red/X and false for Green/O), and a file name for the output file. The effect of `drawStrategy b fn` is that PostScript code that draws the appropriate strategy tree is generated and written to the file `fn`.

How to turn in

Use the UNM Learn facility as follows: Navigate to `https://learn.unm.edu/` and log in. Then click on CS-357L-000 (Spring 2018). Now click on Assignments in the left side navigation menu. After that, click on the appropriate homework assignment link. Now attach your `.hs` file(s) and click submit. You are allowed to submit as many times as you like but only the latest submission will be graded.