CSCC24 2021 Summer – Assignment 2
Due: Wednesday, July 7, midnight
This assignment is worth 10% of the course grade.

In this assignment, you will deepen (pun intended, just you wait) your experience with lazy evaluation and lazy infinite data in Haskell.

As usual, you should also aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

In questions 2 and 3, code quality is worth 10%.

Question 1: Solitaire Knight (2 marks)

Here is a game: The board is an $n \times n$ chess board, $n \geq 1$. A knight is at some initial position on the board, and it is the only game piece. The legal moves are exactly the legal knight moves. Perhaps the goal is to get the knight to a certain destination, perhaps in as few moves as possible.

Implement

```
knightNext :: Int -> (Int, Int) -> [(Int, Int)]
```

If the board size is $n \times n$, and the knight position is currently (x, y), then $\mathit{knightNext}\, n\, (x, y)$ should be the list of positions the knight can be after one move. We use integers in $1, \cdots, n$ for coordinates, e.g., $1 \leq x, y \leq n$.

Example:
$\mathit{knightNext}\, 60\, (2, 5) = [(4, 6), (3, 7), (1, 7), (1, 3), (3, 3), (4, 4)]$
or some other order.

Games, Game Trees, And Game State Transition Functions

A game like Chess, Go, Tic-Tac-Toe, or Solitaire Knight above can be modelled as an initial game state and a state transition function. The state transition maps a state to possible next states after one move.

In Solitaire Knight, the only information we need for a game state is the knight position. Given board size $n \times n$, the state transition function is $\mathit{knightNext}\, n$.

If we want to know how to end in the fewest number of moves, but we can't (or too lazy to) think up a clever algorithm, we are happy enough to brute-force: For i from 0 to ∞, look at the states reachable from the initial state after i moves, is any of them an end state?

People think of it as searching through a tree. The root is the initial state; from a parent state, use the state transition function to generate the children. The search is a breadth-first seach over the tree. However, to save memory, we don't build the whole humongous tree up front (we don't even remember which states have been seen before); we generate states as we go and later throw them away.

Question 2: Breadth-First Search (4 marks)

Implement breadth-first search given a state transition function and an initial state:

```
bfs :: (a -> [a]) -> a -> [a]
```

The 1st parameter is the state transition function; the 2nd parameter is the initial state. The outcome is the lazy list of states of the game tree in breadth-first order as explained above.

As an example and test, given this toy state transition function

```
quad i = [4*i + 1, 4*i + 2, 4*i + 3, 4*i + 4]
```

`bfs quad 0` should give the same list as `[0..]`. You can also try it on the Solitaire Knight game.

BFS makes you maintain a queue, so you will need to code up a helper function that takes an extra parameter for the queue, and this helper is also where the real recursion happens.

A downside of BFS is that queue size grows linearly in the number of nodes traversed, i.e., exponentially in depth. Moreover, in a functional language, if you implement enqueuing by simple list appending, your queue operations are no longer $O(1)$-time. It is OK to have these shortcomings in this question.

Question 3: Iterative Deepening (6 marks)

Iterative deepening is a seemingly magical way of outputting states in breadth-first order, but the traversal order is depth-first to save space (linear in traversed depth, logarithmic in number of traversed nodes). How can this be? Here is the sacrifice: We are happy to re-generate and re-traverse the same nodes over and over again. Here:

0. DFS from root to depth 0, output only nodes at depth 0—the root
1. DFS from root to depth 1, output only nodes at depth 1
2. DFS from root to depth 2, output only nodes at depth 2
3. ...

Implement iterative deepening given a state transition function and an initial state:

```
iterDeep :: (a -> [a]) -> a -> [a]
```

You can assume that every depth has at least a node. (More precisely, my tests will stop before it matters.)

Good news: A straightforward recursive helper for "given root and k, output the list of nodes at depth k", under lazy evaluation, will happen to do depth-first traversal.

Bad news: This part will be tested under limited memory and time to prevent you from coding up a less efficient algorithm. On mathlab, my solution used for

```
main = print (iterDeep quad 0)
```

is compiled as an executable at
/courses/courses/cscc24s21/laialber/quads
You can run it to watch how fast it can be, and use top or htop to monitor how little space (the "RES" field) it uses.

Silver lining: Your code (and my sample executable) will be compiled with "`ghc -O`" for testing, so you will not be missing out on well-known code optimizations.

End of questions.