Midterm Review

Two roads diverged in a wood

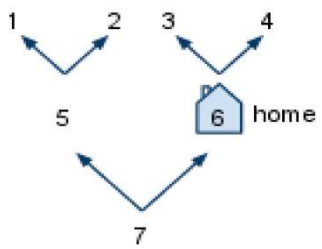
You're lost in the forest. Every [place] in the forest is either a *dead-end* or has exactly 2 *one-way paths*: *left* and *right*. Your goal is to find out if there is a way home. We introduce a new data type called a [place], but you don't know *(and you don't need to know)* how it is represented; it could be a string, a number, or a list. You are presented with four new blocks, two predicates and two reporter blocks (all take a place as an argument):

- [home? place] returns [true] if the [place] is your home, [false] otherwise.
- [dead-end? place] returns [true] is the place is a dead-end (i.e., no paths from it).
- [go-left place] follows the *left* path, returning a new [place].
- [go-right place] follows the *right* path, returning a new [place].

It is an error to [go-left place] or [go-right place] if [place] is a *dead-end* (because it has no paths!). There is no way in this forest to follow a sequence of left paths and/or right paths and end up where you started. I.e., there's no way to walk in circles. Your *home* (if one exists) might be at a dead-end or it might not. You might actually start your search at home.

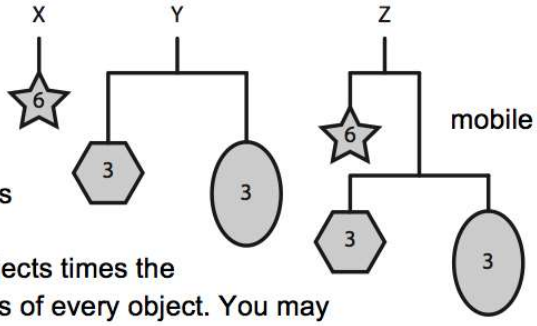
Write [path-home? place], which uses the four functions above and returns [true] if you can get home following a (possibly zero) number of lefts and rights starting from [place], and [false] otherwise. Use the technique we described for authoring BYOB code on paper. We've provided an example forest for you, but **your solution needs to be able to work with ANY forest.** Below, we present a table that shows the responses of various blocks when you are at different places in the sample forest on the lower right.

place	home? place	dead-end? place	go-left place	go-right place	path-home? place
1	false	true	ERROR	ERROR	false
2	false	true	ERROR	ERROR	false
3	false	true	ERROR	ERROR	false
4	false	true	ERROR	ERROR	false
5	false	false	1	2	false
6	true	false	3	4	true
7	false	false	5	6	true




```
if <dead-end? (place)>
    report False
if <home? (place>
    report True
else
    report < <path-home? (go-left (place))> or <path-home? (go-right (place))> >
```

A little town in Alabama…



You fondly remember the *mobiles* hanging above your crib, but you always wondered what force it took to hold them up. You wish to write **`Force(mobile)`** to answer that question. A mobile is either *simple* (has only a single object hanging from it), or *complex* (has a horizontal "inverted-T" rod that balances two mobiles on its left and right). Each object has a mass (the numbers in the examples on the right), and the *total* force is the *total* mass of all objects times the GRAVITY constant, also computed as the sum of the individual forces of every object. You may assume the vertical strings and horizontal "inverted-T" rods are weightless. From our example, **`Force(X)`** = **`Force(Y)`** = 6 * GRAVITY, and **`Force(Z)`** is double that.

Here are 4 helper blocks you'll need to use:

Block	Description
Simple? Mobile	Report if **`Mobile`** is *simple*, **`true`** for X above, **`false`** for Y and Z.
Left Complex Mobile	Reports the mobile on the *left* of the topmost "inverted-T" junction. Calling this function is an error if the mobile is simple. Example: **`Left(Z)`** would report a mobile identical to X.
Right Complex Mobile	Reports the mobile on the *right* of the topmost "inverted-T" junction. Calling this function is an error if the mobile is simple. Example: **`Right(Z)`** would report a mobile identical to Y.
Mass Simple Mobile	Reports the mass of the simple mobile. Calling this function is an error if the mobile is complex. Examples: **`Mass(X)`** would report 6, and **`Mass(Left(Y))`** would report 3.

a) Complete **`Force(mobile)`**, that reports the force required to hold up the mobile. Use GRAVITY as needed.

```
Force(mobile)
    if ( <simple? (mobile)> )
        report ( ( (GRAVITY) x (Mass (mobile))) )
    else
        report ( ( ( Force (Left (mobile))) + ( Force (Right (mobile))) ) )
```

b) As a function of the number of objects, what is the order of growth of **`Force(mobile)`**? Circle one:

Constant Logarithmic **Linear** Quadratic Cubic Exponential

c) Your solution above was either Iterative or Recursive. If, instead of the constraining **`if-else`** structure we provided, we had no structure, could it have been written *the other way*? Circle one and explain.

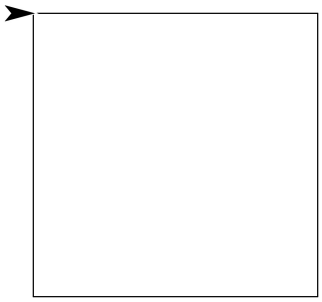
TRUE FALSE Reason: Anything that can be written iteratively can be written recursively and vice versa. Iterative and Recursive present different ways of writing code but you can still write it either way and have the same outcome.

Diamonds In the Rough

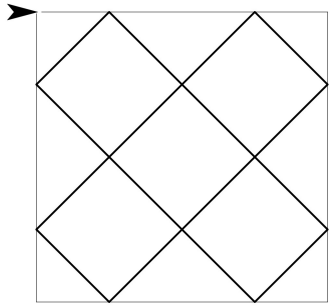
In the following exercise, you will be coding the square fractal below. We have provided the first four levels of the fractal. The recursive cases for each level are represented by bold lines.

Level 1 is 300 pixels on each side. Each of the sides of the four smaller diamonds are the length of the side of the larger square divided by the square root of 8. You may assume that the sprite starts off at the top-left corner of each level, facing right and ends in the same position. (Hint: The repeat block will be very useful for both the base case and the recursive case)

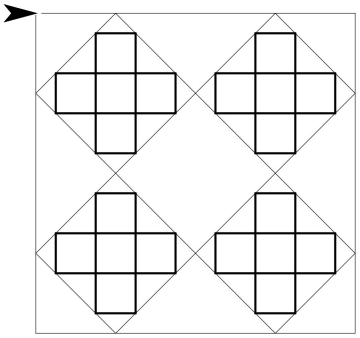
Level 1:



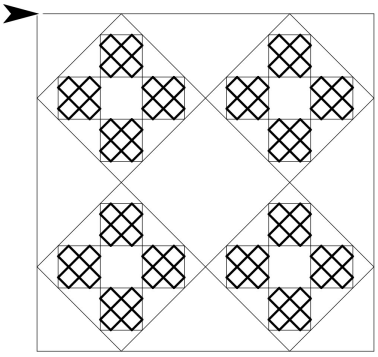
Level 2:



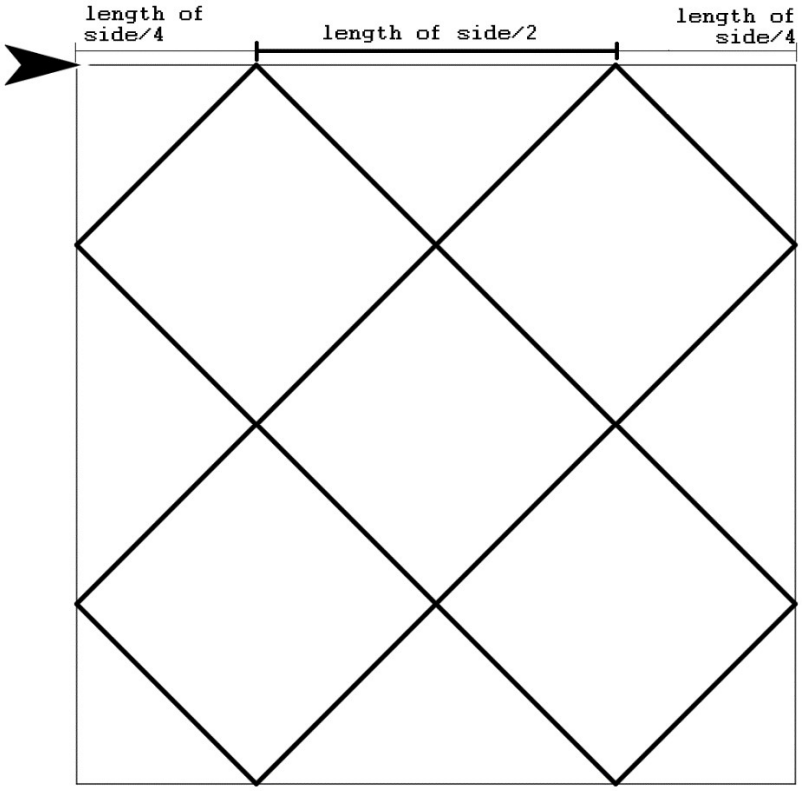
Level 3:



Level 4:



The measurements of Level 2 are as follows:



base case

```
if level == 1
    repeat 4
        move size steps
        turn right 90 degrees
```

This draw the level 1 case.

The code inside of the repeat draws one side of the square. We repeat 4 times in order to draw all four sides of the square.

recursive case

```
repeat 2
    move (size / 4) steps
    turn right 45 degrees
    draw fractal (levels - 1) and (size / (sqrt of 8))
    turn left 45 degrees
    move (size / 2) steps
    turn right 45 degrees
    draw fractal (levels - 1) and (size / (sqrt of 8))
    turn left 45 degrees
    move (size / 4) steps
    turn right 90 degrees
    move size steps
    turn right 90 degrees
```

This section draws two sides of the square and two of the previous level.

We move (size / 4) steps in order to get to the correct starting point for the first recursive case. We then move (size / 2) to move on to the starting point of the next recursive case.

Then we move (size / 4) steps to the corner, turns right 90 degrees, and moves size steps to draw a new line.

We then repeat this code to draw the rest of the fractal.