COMP1406 – Winter 2019

Submit a single file called `assignment5.zip` to the submission server
http://134.117.31.149:9091/

Your zip file must contain a directory called `comp1406a5` and all of your `.java` files must be in this directory. Do not include your `.class` files.

This assignment has 50 marks.

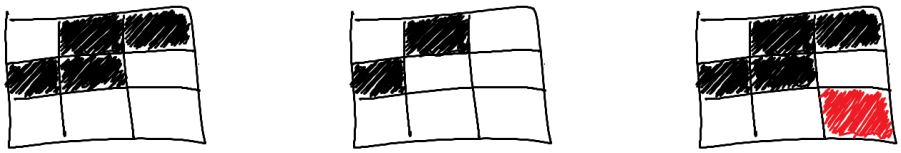
1 Blobs [25 marks]

An `Image` is a 2-dimensional grid of `Pixel`s. A pixel either has ink (i.e., you can see it) or does not. In this problem, you will use recursion to compute the border length of a blob of ink in an image. A blob is one or more pixels that are touching each other (neighbours in the grid). Two pixels at positions (r_1, c_1) and (r_2, c_2) are touching each other if one of the following is satisfied:

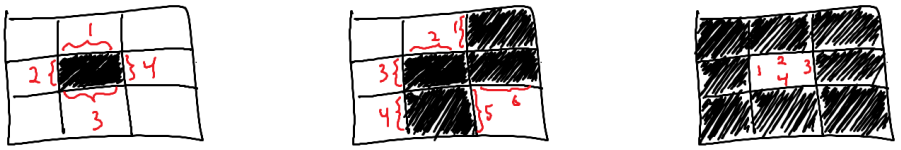
- $r_1 = r_2$ and $c_1 = c_2 \pm 1$
- $r_1 = r_2 \pm 1$ and $c_1 = c_2$

Here r_i is the row of pixel i and c_i is the column of pixel i.

For example, the image on the left (below) has one blob with four pixels in it, the middle image two blobs (each consisting of a single pixel) and the image on the right has two blobs (one of size four pixels in black and one of size one pixel in red). The black (and red) cells in the grid are pixels that have ink. The empty (white) cells do not have ink. All images have 3 rows and 3 columns. The red pixel (right image) is located at row 2 and column 2. (Rows and columns start with index 0 in image.)



In this problem, you will complete the `computePixelPerimeters()` and `perimeter()` methods in the `Blob` class. The perimeter of a blob (which your `perimeter()` method will output) is defined as the sum of all edges of each pixel in the blob where the pixel on the other side of the edge is not in the blob. Each pixel has four edges (the boundaries around the pixel). In the three examples below, the perimeters are given by 4 (left), 6 (middle) and 4 (right). Note that a blob might have "holes" in them (like the right image) and that the edges in the hole count toward the overall perimeter. Note also that we do NOT count edges where the pixel is at the edge of the image.



Example: Using the middle image above (let the image be called img)

```
img.perimeter(0,0) ---outputs--> 0
img.perimeter(1,1) ---outputs--> 6
img.perimeter(2,0) ---outputs--> 0
img.perimeter(1,2) ---outputs--> 6
```

The `computePixelPerimeters()` method will modify every pixel in the specified blob to update their 'sides' attribute. This attribute will store the number of pixels touching it that DO NOT have ink. For example, using the three images above, the 'sides' attributes of the pixels in the blob would be assigned as follows (where - denotes a pixel not in the blob):

-	-	-
-	4	-
-	-	-

-	-	1
-	2	1
-	2	-

0	1	0
1	-	1
0	1	0

It is suggested that the first line of your `perimeter()` method is a call to `computePixelPerimeters()` with the exact same input values (row and column). If you do this then your `perimeter()` method will be able to use the 'sides' attributes of all the pixel in the blob to help you solve the problem.

Note that the `Pixel` class has several attributes that can be changed (setters are provided).The `extra` attribute (Object) can be used to store anything you want (if needed). You do not have to use this attribute but it is available if you chose to use it. If you use this you will need to cast whatever you store back to whatever it is to actually retrieve the information.

```
public void computePixelPerimeters(int row, int column)
  // purpose: set the `sides' attribute of each pixel in the blob
  //          containing the pixel at position (row, column)
  // preconditions: (i) 0 <= row < this.rows
  //                (ii) 0 <= column < this.cols
  // postconditions: each pixel in the specified blob has its
  //                 `sides' attribute set to the number of neighbouring
  //                 pixels that DO NOT have ink.

public int perimeter(int row, int column)
  // purpose: compute the perimeter of the entire blob that has pixel at
  //          the position (row, column)
  // preconditions: (i) 0 <= row < this.rows
  //                (ii) 0 <= column < this.cols
  // postconditions: returns the length of the perimeter of blob containing
  //                 the pixel at position (row, column) in the image
  //
```

Your MUST use recursion for this problem. If your solution does not use recursion you may receive zero marks for this problem. If your `Blob` class has the word `for` or `while` in it (even if it is in a comment) the file will be rejected by the server. You may use the `clear` method in the `Image` class which uses a loop.

Do **NOT** import anything for this problem. Do **NOT** use any other packages/libraries/classes for this problem (except `Pixel` and `Image`).

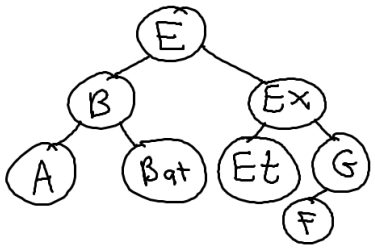
2 BST (Binary Search Tree) [25 marks]

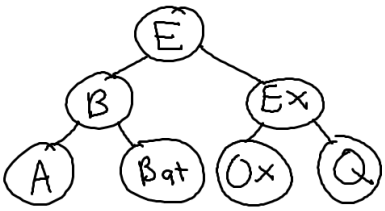
A binary search tree (BST) is a binary tree with the additional binary search tree property. The binary search tree property states that for every node `n` in the tree

1. `n.data` is greater than the `data` value of all nodes in the sub-tree rooted with `n.left`.
2. `n.data` is less than the `data` value of all nodes in the sub-tree rooted with `n.right`.

For this problem, all data (strings) in the tree will be unique. Every node in the tree will have its own distinct string. Strings will be compared using the canonical way (as defined by the `compareTo()` method in the String class).

For example, the binary tree on the left (below) IS a binary search tree and the one on the right is NOT.





You will implement several methods for a binary search tree in the `BST` class. The `BST` class must extend the `BinaryTree` class.

You will override the following methods from the BinaryTree class

```
public boolean contains(String s)
  // precondition: s is string and this is a valid binary search tree
  // postcondition: returns true if s in the current tree, false otherwise
  // efficiency: (i) this method must be efficient
  //             (ii) do NOT use recursion for this method

public void add(String s)
  // precondition: (i) s is a string that is not already in the tree
  //               (ii) this is a valid binary search tree
  // postcondition: (i) s is added to the tree so that the resulting tree
  //                    is a valid binary search tree
  //                (ii) the original tree structure must not be altered.
```

You will define two new methods in the BST class

```
public boolean isValidBST()
  // precondition: this is a binary tree
  // postcondition: returns true if this tree is a valid binary search tree, false otherwise
  // note: the empty tree is a valid binary search tree

public BST makeBalanced()
  // precondition: this is a valid binary search tree
  // postcondition: returns a new binary search tree that
  //                (i) has the same data (strings) as this tree
  //                (ii) has minimal height
  //                (iii) is a valid binary search tree
```

The `makeBalanced()` method must create a new balanced binary search tree with minimal height. For a given tree there may be many different valid binary search trees with the same minimal height. It does not matter which optimal tree you return.

Do NOT change the `BinaryTree` class. You do NOT have to use recursion to solve these problems. I would suggest that you use an iterative approach for `contains` and `add`.

Submission Recap

A complete assignment will consist of a single file (`assignment5.zip`) that has a single folder/directory called `comp1406a5`. The `comp1406a5` folder will have the following two files included:

```
Blob.java
BST.java
```

All classes must be in the `comp1406a5` package. That is, all files must have the `package comp1406a5;` directive as the first line. Your code will NOT compile if it does not have this and you will receive zero correctness marks if your code does not compile.