Computer Science E214

Tutorial 3

For this tutorial, you may do either Question 2 or Question 3 (the latter is a little more challenging), as well as your choice of Question 4 or Question 5. In addition, the last Question is optional.

1 Maze

Create a program which employs and modifies portions of the percolation case study in Section 2.4 of the textbook to address the following problem. Your program must, given a starting and final position, find a path through a given square maze between these two points.

You must write a class called `Maze`, which implements the path-finding logic in a method called `path`. The method is passed an array of booleans (similar to the array `open` within Percolation), as well as a starting row and column, and an ending row and column. This method must then find a path through the maze from the starting position to the ending position. If a path is found, it must return the path represented as a two-dimensional array of `boolean`s, where parts of the path are marked `true`, and other cells are marked false. **The path you return must not cross itself, or contain any branches.** If no path is found, your function should return `null`.

Note that `Maze` itself does not need to have a `main` method.

You must also write a class `Visualize` which is similar to that in the percolation case study. It must have a `main` method that takes two command-line arguments, for example:

```
java Visualize 19 10
```

This command should generate 10 mazes of size 19x19, choose random distinct open starting and end points in each case, and show the solution for each briefly (similar to the `Visualize` class in the percolation case study). The output of your `Visualize` program for a single maze should look something like Figure 1.

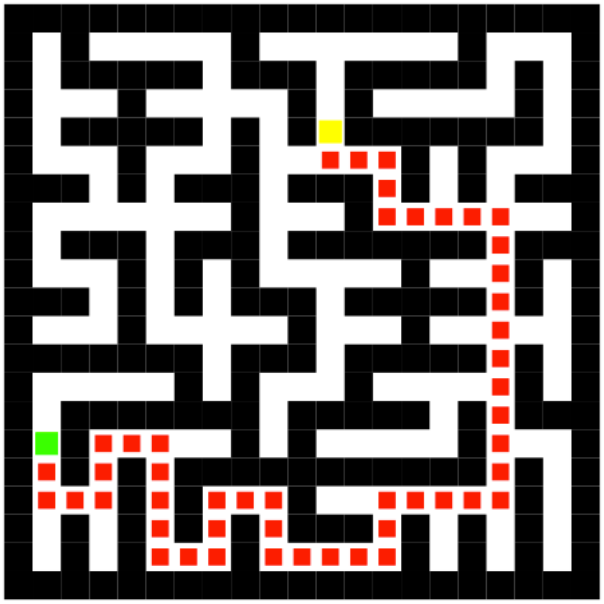


Figure 1: Sample output for the Visualize class. The start position is marked green, the end in yellow. The path returned by `Maze.path()` is shown in red — note that the yellow and green blocks each occlude a further red square. Also note that the red, yellow and green blocks are slightly smaller than each grid block, so that it is easy to spot if your path accidentally passes through a wall.

To help you debug your program, you are provided with a class `RandomMaze`, which generates an $N \times N$ maze each time `RandomMaze.getMaze(N)` is called. The mazes returned will be guaranteed to have a (unique) path between any pair of open points in the maze. Your `Visualize` class must also call this each time a new maze is needed. In addition, some JUnit test cases have been provided in `MazeTest.java` to help you with the initial development of `Maze`. Your `Visualize` class need not work for $N < 5$, since in such cases, `RandomMaze.getMaze` returns mazes with less than two open cells!

2 Rotation filter (Book 3.1.38)

Write a program `Rotation` that takes two command-line arguments (the name of an image file and a real number, θ) and rotates the image θ degrees clockwise about its centre. For an example, see Figure 2. For each target pixel, you have to copy the colour of a pixel in the source image. The following equations give the relationship between the co-ordinates (s_i, s_j) in the source image and the co-ordinates (t_i, t_j) in the target image:

$$\begin{aligned} t_i &= (s_i - c_i)cos(\theta) - (s_j - c_j)sin(\theta) + c_i \\ t_j &= (s_i - c_i)sin(\theta) + (s_j - c_j)cos(\theta) + c_j \end{aligned}$$

Here (c_i, c_j) are the co-ordinates of the point in the image being rotated about, and θ is the (clockwise) angle of rotation.

(a) Original (b) Rotate 30 degrees

Figure 2: Baboon on a merry-go-round

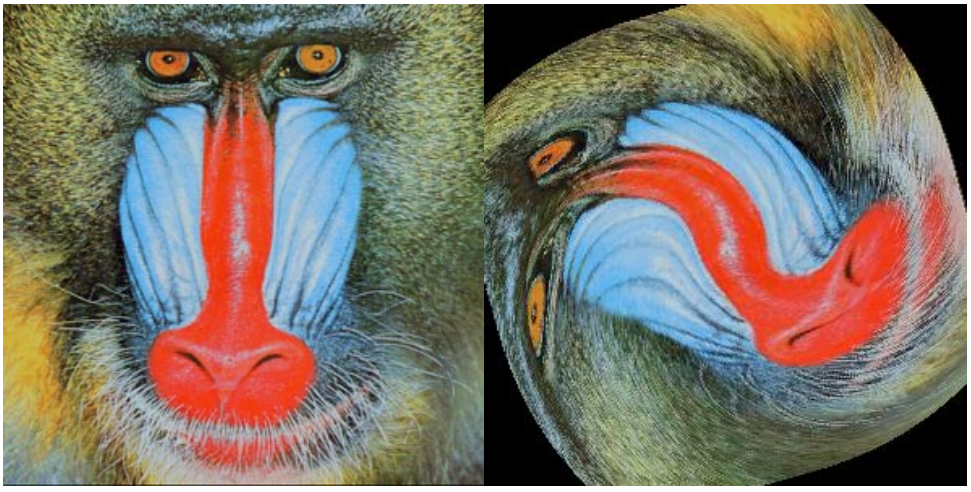
You may make the output image the same size as the original image.

Hint: It is easier to run through the taget picture's pixels and get their values from the source, so it may be convenient to think how to invert the rotation above (and what effect that has on the fomulae) in order to get formulae for s_i and s_j from t_i and t_j instead.

Additional challenge: Modify the output image so that corners of the original image are not cut off by the rotation, but so that the rotated image fits snugly in the new image.

3 Swirl filter (Book 3.1.39)

Write a program `Swirl` that implements a swirl effect similar to the rotation, except that the angle changes as a function of the distance from the center. For an example, see Figure 3. Use the same formulae as in the rotation filter, but compute θ as a function of (s_i, s_j): specifically, let θ be $\pi/256$ times the distance to the center.



(a) Original (b) Swirl filter

Figure 3: Baboon sucked into a black hole

4 Safe password verification (Book 3.1.32)

Write a static method `checkConditions`, in a class `Authenticate`, that takes a string as parameter and returns `true` if it meets the following conditions (and `false` otherwise):

- it is at least eight characters long
- it contains at least one digit `[0-9]`
- it contains at least one upper case letter `[A-Z]`
- it contains at least one lower case letter `[a-z]`
- it contains at least one character that is neither a letter nor a number.

Include a number of test cases checking your code in the `main` method of your class.

5 Weather (Book 3.1.27)

Write a screen-scraper program `Weather` so that typing `java Weather` will give you the weather forecast provided on `weather.sun.ac.za`.

6 Rational numbers (Book 3.2.7)

Implement a data type `Rational` for rational numbers that supports addition, subtraction, multiplication and division, as per the following API. Your implementation need only support positive rational numbers.

`public class Rational`		
	`Rational(int numerator, int denominator)`	
`Rational`	`plus (Rational b)`	sum of this number and b
`Rational`	`minus(Rational b)`	difference of this number and b
`Rational`	`times(Rational b)`	product of this number and b
`Rational`	`over (Rational b)`	quotient of this number and b
`String`	`toString()`	string representation

The string representation should print the rational number in its simplest form. (Copy the method `gcd` from PROGRAM 2.3.1 into your class (as a `private` method), and use it to ensure that the numerator and the denominator do not have any common factors.) Include a `main` method with test cases exercising all of your methods.

In your implementation, declare the instance variables used to represent the numerator and denominator as `private` and `final`.

7 Quaternions (Book 3.2.25)

In 1843, Sir William Hamilton discovered an extension to complex numbers called *quaternions*. Quaternions extend the concept of rotation in three dimensions to four dimensions. Applications of quaternions include computer graphics, control theory, signal processing, and orbital mechanics.

Quaternions are generated by adding new operations to the set of 4-dimensional real vectors $a = (a_0, a_1, a_2, a_3)$. All vectors support scalar multiplication and addition: for a real number c, we have

$$ca = (ca_0, ca_1, ca_2, ca_3) \ ,$$

and for two vectors a and b, we have

$$a + b = (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3) \ .$$

In addition, we can define the magnitude (or norm) of a vector by

$$|a| = \sqrt{a_0^2 + a_1^2 + a_2^2 + a_3^2} \ .$$

Quaternions add the following operations:

- Conjugate: $a^* = (a_0, -a_1, -a_2, -a_3)$; and
- Product[1]:

$$\begin{aligned} a * b = (&a_0 b_0 - a_1 b_1 - a_2 b_2 - a_3 b_3, \\ &a_0 b_1 - a_1 b_0 + a_2 b_3 - a_3 b_2, \\ &a_0 b_2 - a_1 b_3 + a_2 b_0 + a_3 b_1, \\ &a_0 b_3 + a_1 b_2 - a_2 b_1 + a_3 b_0) \ . \end{aligned}$$

Using these operations, we can define the inverse[2] of a quaternion by $a^{-1} = a^*/|a|$, and use this to divide quaternions: $a/b = a * b^{-1}$.

Design an API for a data type `Quaternion` for representing quaternions that supports all of the operations above in some way. Implement your API in a class called `Quaternion`. Place the API in the comments at the top of the class, and include testing code in your `main` method that checks all of your methods work correctly.

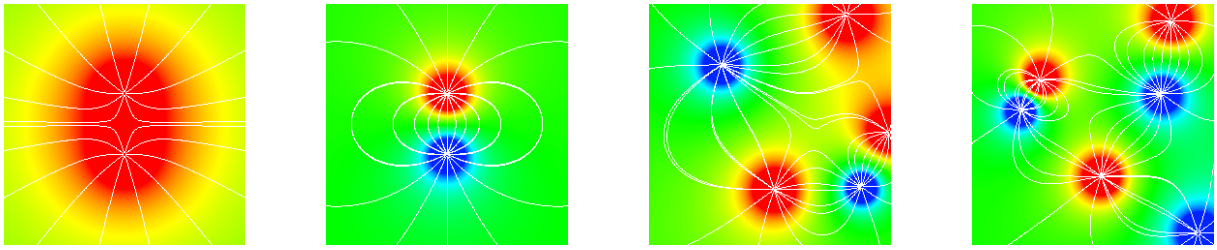
For your implementation, declare the instance variables for the components of each quaternion as `private` and `final`. This requirement may impact the design of your API!

8 Electric field lines (Web 3.2.34)

Michael Faraday introduced an abstraction called *electric field lines* to visualize an electric field. By Coulomb's law, the electric field at a point (x, y) induced by a point charge q_i is given by $E_i = kq_i/r^2$ and the direction points towards q_i if E_i is negative and away from q_i if E_i is positive. Here q_i is the magnitude of the charge and r is the distance from the charge to (x, y). If there are a group of n point charges, the electric field at a point is the vector sum of the electric fields induced by these n individual point charges. Fig. 4a below illustrates the field lines for two equal positive point charges and Fig. 4b illustrates two point charges of opposite signs. The second configuration is called an *electric dipole*: the charges cancel each other out, and the electric field weakens very quickly as you move away from the charges. Examples of dipoles can be found in molecules where charge is not evenly distributed. Oscillating dipoles can be used to produce electromagnetic waves to transmit radio and television signals. Figures 4c and 4d illustrate the electric potential and field lines for several random charges of equal magnitude.

[1] Note that $a * b \neq b * a$ in general!

[2] Again, note that we can not be sure that $a * b^{-1} = b^{-1} * a$.



(a) (b) (c) (d)

Figure 4: Electric potential and field lines of various charge configurations

Write a program `FieldLines.java` which reads in a number of charges from standard input, and then draws 10 electric field lines coming out of each charge. (We take some liberties since the number of electric field lines per unit area should actually be proportional to the magnitude of the field strength.) Each line starts on a small circle around the charge, at 10 equally spaced angles.

After identifying a starting position, we can generate a field line from that starting point by repeatedly calculating the electric field strength at the current position, and then moving a short distance using the direction of the electric field, drawing a line between the old and the new position. In the case of a positive charge, we move in the direction of the field, following the field line; in the case of a negative charge, we move in the opposite direction, tracing our way back along the field line. We repeat this process until we reach the boundary of the region we are plotting or we arrive at another point charge — do not move too far each step, or you might move right through another point charge!

For your convenience, two example files containing input for your program are provided — please see them for the format of the input. `dipole.txt` sets up an electric dipole, while `charges.txt` has a number of differing charges at various locations. Finally, a modified version of the book's `Charge.java` adding accessor methods for the co-ordinates and the strength of the charge is provided for you to use to model each charge.