Data Structures and Algorithms in Java

Assignment 3 (Percolation) Discussion

Implement a data type called `Die` that represents a six-sided die and supports the following API

Die	
`Die()`	constructs a die with the face value -1
`void roll()`	rolls this die
`int value()`	returns the face value of this die
`boolean equals(Die other)`	returns `true` if this die is the same as `other`, and `false` otherwise
`String toString()`	returns a string representation of this die

>_ ~/workspace/percolation

```
$ javac -d out src/Die.java
$ java Die
*   *
  *
*   *
$ java Die
*


    *
```

Instance variables

- Face value of the die, *value* (int)

`Die()`

- Set *this.value* to -1

`void roll()`

- Set *this.value* to a random integer from $[0, 7)$

`int value()`

- Return *this.value*

`boolean equals(Object other)`

- Return *true* if *this* die and *other* have the same value, and *false* otherwise

`String toString()`

- Return a string representation of the die (use the format `".....\n.....\n....."`, where each `.` is either a space or a `*`)

Implement a data type called `Location` that represents a location on Earth and supports the following API

Location

`Location(String name, double lat, double lon)`	constructs a new location given its name, latitude, and longitude
`double distanceTo(Location other)`	returns the great-circle distance between this location and `other`
`boolean equals(Object other)`	returns `true` if the latitude and longitude of this location are the same as those of `other`, and `false` otherwise
`String toString()`	returns a string representation of this location

`>_ ~/workspace/percolation`

```
$ javac -d out src/Location.java
$ java Location
x                = Paris (48.51, -2.17)
y                = Boston (42.36, -71.05)
x.distanceTo(y) = 5224.780334245809
x.equals(y)      = false
```

Instance variables

- Name of the location, *name* (String)
- Latitude of the location, *lat* (double)
- Longitude of the location, *lon* (double)

`Location(String name, double lat, double lon)`

- Set *this.name* to *name*, *this.lat* to *lat*, and *this.lon* to *lon*

`double distanceTo(Location other)`

- Return the great-circle distance between *this* location and *other* computed using the formula from Problem 3 of Assignment 1

`boolean equals(Object other)`

- Return *true* if *this* location and *other* have the same latitude and longitude values, and *false* otherwise

`String toString()`

- Return a string representation of the location (use the format `"<lat>, <lon>"`)

Implement an immutable data type called `Rational` that represents a rational number and supports the following API

Rational	
`Rational(long x)`	constructs a rational number whose numerator is `x` and denominator is 1
`Rational(long x, long y)`	constructs a rational number given its numerator `x` and denominator `y`
`Rational add(Rational other)`	returns the sum of this rational number and `other`
`Rational multiply(Rational other)`	returns the product of this rational number and `other`
`boolean equals(Object other)`	returns `true` if this rational number is equal to `other`, and false otherwise
`String toString()`	returns a string representation of this rational number

```
>_ ~/workspace/percolation

$ javac -d out src/Rational.java
$ java Rational 10
1 + 1/2 + 1/4 + ... + 1/2^10 = 1023/512
```

Instance variables

- Numerator, x (long)
- Denominator, y (long)

`Rational(long x)`

- Set *this*.x to x and *this*.y to 1

`Rational(long x, long y)`

- Set *gcd* (long) to `gcd(x, y)`
- Set *this*.x to x/gcd and *this*.y to y/gcd

`Rational add(Rational other)`

- Return the sum of *this* and *other* rational numbers (recall, $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$)

`Rational multiply(Rational other)`

- Return the product of *this* and *other* rational numbers (recall, $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$)

`boolean equals(Object other)`

- Return *true* if *this* rational number and *other* have the same numerator and denominator values, and *false* otherwise

Harmonic.java

Command-line input	n (int)
Standard output	the nth harmonic number, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$, in rational form

~/workspace/percolation

```
$ javac -d out src/Harmonic.java
$ java Harmonic 10
7381/2520
```

Accept n (int) as command-line argument

Set *total* (Rational) to the rational number 0

For each int i in $[1, n]$
- Set *term* (Rational) to the rational number $1/i$
- Increment *total* by *term*

Write *total*

The percolation threshold of a system is a measure of how porous the system needs to be so that it percolates

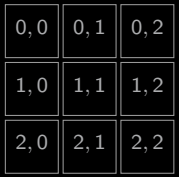
Goal: write programs to estimate the percolation threshold of a system

We model percolation system as an $n \times n$ array of booleans (`true` $\implies$ open site and `false` $\implies$ blocked site)

We use an `UF` object with $n^2 + 2$ sites and the `encode()` method to translate sites $(0, 0), (0, 1), \ldots, (n-1, n-1)$ of the array to sites $1, 2, \ldots, n^2$ of the `UF` object

Sites 0 (source) and $n^2 + 1$ (sink) are virtual, ie, not part of the percolation system

A 3×3 percolation system and its UF representation



Create a data type called `Percolation` that supports the following API

Percolation	
`Percolation(int n)`	constructs an `n x n` percolation system, with all sites blocked
`void open(int i, int j)`	opens site `(i, j)` if it is not already open
`boolean isOpen(int i, int j)`	returns `true` if site `(i, j)` is open, and `false` otherwise
`boolean isFull(int i, int j)`	returns `true` if site `(i, j)` is full, and `false` otherwise
`int numberOfOpenSites()`	returns the number of open sites
`boolean percolates()`	returns `true` if this system percolates, and `false` otherwise

```
>_ ~/workspace/percolation

$ javac -d out src/Percolation.java
$ java Percolation data/input10.txt
10 x 10 system:
  Open sites = 56
  Percolates = true
$ java Percolation data/input10-no.txt
10 x 10 system:
  Open sites = 55
  Percolates = false
```

Instance variables

- Percolation system size, `int n`
- Percolation system, `boolean[][] open`
- Number of open sites, `int openSites`
- Union-find representation of the percolation system, `WeightedQuickUnionUF uf`

`private int encode(int i, int j)`

- Return the `uf` site $(1, 2, \ldots, n^2)$ corresponding to the percolation system site `(i, j)`

`public Percolation(int n)`

- Initialize instance variables

`void open(int i, int j)`

- If site `(i, j)` is not open
  - Open the site
  - Increment `openSites` by one
  - If the site is in the first (or last) row, connect the corresponding `uf` site with the source (or sink)
  - If any of the neighbors to the north, east, west, and south of site `(i, j)` is open, connect the `uf` site corresponding to site `(i, j)` with the `uf` site corresponding to that neighbor

`boolean isOpen(int i, int j)`

- Return whether site `(i, j)` is open or not

`boolean isFull(int i, int j)`

- Return whether site `(i, j)` is full or not — a site is full if it is open and its corresponding `uf` site is connected to the source

`int numberOfOpenSites()`

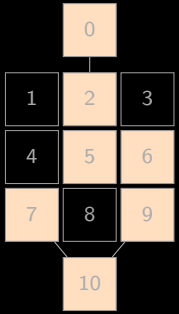
- Return the number of open sites

`boolean percolates()`

- Return whether the system percolates or not — a system percolates if the sink is connected to the source

Using virtual source and sink sites introduces what is called the *back wash* problem

In the 3×3 system, consider opening the sites $(0,1)$, $(1,2)$,$(1,1)$, $(2,0)$, and $(2,2)$, and in that order; the system percolates once $(2,2)$ is opened



The site $(2,0)$ is not full, but the corresponding `uf` site 7 is connected to the source, so is incorrectly reported as being full — this is the back wash problem

To solve the back wash problem, create another `WeightedQuickUnionUF` object

Create an immutable data type called `PercolationStats` that supports the following API

PercolationStats	
`PercolationStats(int n, int m)`	performs `m` independent experiments on an `n x n` percolation system
`double mean()`	returns sample mean of percolation threshold
`double stddev()`	returns sample standard deviation of percolation threshold
`double confidenceLow()`	returns low endpoint of 95% confidence interval
`double confidenceHigh()`	returns high endpoint of 95% confidence interval

>_ ~/workspace/percolation

```
$ javac -d out src/PercolationStats.java
$ java PercolationStats 100 1000
Percolation threshold for a 100 x 100 system:
  Mean                = 0.592
  Standard deviation  = 0.016
  Confidence interval = [0.591, 0.593]
```

Instance variables

- Number of independent experiments, `int m`
- Percolation thresholds for the `m` experiments, `double[] x`

`PercolationStats(int n, int m)`

- Initialize instance variables
- Repeat the following experiment `m` times
  - Create an $n \times n$ percolation system
  - Until the system percolates, choose a site (i, j) at random and open it if it is not already open
  - Calculate percolation threshold as the fraction of sites opened, and store the value in `x[]`

`double mean()`

- Return the mean μ of the values in `x[]`

`double stddev()`

- Return the standard deviation σ of the values in `x[]`

`double confidenceLow()`

- Return $\mu - \frac{1.96\sigma}{\sqrt{m}}$

`double confidenceHigh()`

- Return $\mu + \frac{1.96\sigma}{\sqrt{m}}$