Project 1 (Percolation) Checklist

Prologue

Project goal: write a program to estimate the percolation threshold of a system

Relevant lecture material

⇝ Programming Model

⇝ Data Abstraction

Files

⇝ `project1.pdf` (project description)

⇝ `project1.zip` (starter files for the exercises/problems, `report.txt` file for the project report, `run_tests.py` file to test your solutions, and test data files)

Exercise 1. (*Great Circle Distance*) Write a program `GreatCircle.java` that takes four doubles x_1, y_1, x_2, and y_2 representing the latitude and longitude in degrees of two points on earth as command-line arguments and writes the great-circle distance (in km) between them, given by the equation

$$d = 111 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2)).$$

```
$ java GreatCircle 48.87 -2.33 37.8 -122.4
8701.389543238289
```

Exercises

GreatCircle.java

```java
public class GreatCircle {
    public static void main(String[] args) {
        // Get angles x1, y1, x2, and y2 from command line as
        // doubles.
        ...

        // Convert the angles to radians.
        ...

        // Calculate great-circle distance d.
        ...

        // Write d.
        ...
    }
}
```

Exercise 2. (*Counting Primes*) Implement the static method `isPrime()` in `PrimeCounter.java` that takes an integer argument x and returns `true` if it is prime and `false` otherwise. Also implement the static method `primes()` that takes an integer argument N and returns the number of primes less than or equal to N. Recall that a number x is prime if it is not divisible by any number $i \in [2, \sqrt{x}]$.

```
$ java PrimeCounter 100
25
$ java PrimeCounter 1000000
78498
```

Exercises

PrimeCounter.java

```java
public class PrimeCounter {
    // Checks if x is prime
    private static boolean isPrime(int x) {
        // For each 2 <= i <= sqrt(x), if x is divisible by
        // i, then x is not a prime. If no such i exists,
        // x is a prime.
        ...
    }

    // Returns the number of primes <= N.
    private static int primes(int N) {
        // For each 2 <= i <= N, use isPrime() to test if
        // i is prime, and if so increment a count. At the
        // end return count.
        ...
    }

    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        StdOut.println(primes(N));
    }
}
```

Exercise 3. (*Euclidean Distance*) Implement the static method `distance()` in `Distance.java` that takes position vectors x and y — each represented as a 1D array of doubles — as arguments and returns the Euclidean distance between them, calculated as the square root of the sums of the squares of the differences between the corresponding entries.

```
$ java Distance
5
-9 1 10 -1 1
5
-5 9 6 7 4
13.0
```

Exercises

Distance.java

```java
public class Distance {
    // Returns the Euclidean distance between the position vectors x and y.
    private static double distance(double[] x, double[] y) {
        // For each 0 <= i < x.length, add the square of
        // (x[i] - y[i]) to distance. At the end return
        // sqrt(distance).
        ...
    }

    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        double[] x = StdArrayIO.readDouble1D();
        double[] y = StdArrayIO.readDouble1D();
        StdOut.println(distance(x, y));
    }
}
```

Exercises

Exercise 4. (*Matrix Transpose*) Implement the static method `tranpose()` in `Transpose.java` that takes a matrix x — represented as a 2D array of doubles — as argument and returns a new matrix that is its transpose.

```
$ java Transpose
3 3
1 2 3
4 5 6
7 8 9
3 3
  1.00000   4.00000   7.00000
  2.00000   5.00000   8.00000
  3.00000   6.00000   9.00000
```

Exercises

Transpose.java

```java
public class Transpose {
    // Returns a new matrix that is the transpose of x.
    private static double[][] transpose(double[][] x) {
        // Create a new 2D matrix t (for transpose) with
        // dimensions n x m, where m x n are the dimensions
        // of x.
        ...

        // For each 0 <= i < m and 0 <= j < n, set t[j][i]
        // to x[i][j].
        ...

        // Return t.
        ...
    }

    // Entry point. [DO NOT EDIT]
    public static void main(String[] args) {
        double[][] x = StdArrayIO.readDouble2D();
        StdArrayIO.print(transpose(x));
    }
}
```

Exercise 5. (*Rational Number*) Implement a data type `Rational` in `Rational.java` that represents a rational number, ie, a number of the form a/b where a and $b \neq 0$ are integers. The data type must support the following API:

method	**description**
`Rational(long x)`	constructs a rational number whose numerator is the given number and denominator is 1
`Rational(long x, long y)`	constructs a rational number given its numerator and denominator†
`Rational add(Rational that)`	returns the sum of this and *that* rational number
`Rational multiply(Rational that)`	returns the product of this and *that* rational number
`boolean equals(Rational that)`	checks if this rational number is the same as that
`String toString()`	returns a string representation of the rational number

† Use the private method `gcd()` to ensure that the numerator and denominator never have any common factors. For example, the rational number 2/4 must be represented as 1/2.

```
$ java Rational 10
true
```

Exercises

Rational.java

```java
// A data type representing a rational number.
public class Rational {
    private long x; // numerator
    private long y; // denominator

    // Constructs a rational number whose numerator is x and
    // denominator is 1.
    public Rational(long x) {
        // Set this.x to x and this.y to 1.
        ...
    }

    // Constructs a rational number given its numerator and
    // denominator.
    public Rational(long x, long y) {
        // Set this.x to x / gcd(x, y) and this.y to
        // y / gcd(x, y).
        ...
    }

    // Returns the sum of this and that rational number.
    public Rational add(Rational that) {
        // Sum of rationals a/b and c/d is the rational
        // (ad + bc) / bd.
        ...
    }

    // Returns the product of this and that rational number.
    public Rational multiply(Rational that) {
        // Product of rationals a/b and c/d is the rational
        // ac / bd.
        ...
    }

    // Checks if this rational number is the same as that.
    public boolean equals(Rational that) {
        // Rationals a/b and c/d are equal iff a == c
        // and b == d.
```

Exercises

Rational.java

```java
        ...
    }

    // Returns a string representation of the rational number.
    public String toString() {
        long a = x, b = y;
        if (a == 0 || b == 1) {
            return a + "";
        }
        if (b < 0) {
            a *= -1;
            b *= -1;
        }
        return a + "/" + b;
    }

    // Returns gcd(p, q), computed using Euclid's algorithm.
    private static long gcd(long p, long q) {
        return q == 0 ? p : gcd(q, p % q);
    }

    // Test client. [DO NOT EDIT]
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Rational total = new Rational(0);
        Rational term = new Rational(1);
        for (int i = 1; i <= n; i++) {
            total = total.add(term);
            term = term.multiply(new Rational(1, 2));
        }
        Rational expected = new Rational((long) Math.pow(2, n) - 1,
                                         (long) Math.pow(2, n - 1));
        StdOut.println(total.equals(expected));
    }
}
```

Exercise 6. (*Harmonic Number*) Write a program `Harmonic.java` that takes an integer n as command-line argument, and uses the `Rational` data type from the previous exercise to compute and write the nth harmonic number H_n as a rational number. H_n is calculated as

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n}.$$

```
$ java Harmonic 5
137/60
```

Exercises

Harmonic.java

```
public class Harmonic {
    public static void main(String[] args) {
        // Get n from command line as integer.
        ...

        // Set total to the rational number 0.
        Rational total = ...;

        // For each 1 <= i <= n, add the rational term
        // 1 / i to total.
        for (...) {
            Rational term = ...;
            total = ...;
        }

        // Write total.
        ...
    }
}
```

Student

The guidelines for the project problems that follow will be of help only if you have read the description of the project and have a general understanding of the problems involved. It is assumed that you have done the reading.

Instructor

Please summarize the project description for the students before you walk them through the rest of this checklist document.

Problems

Problem 1. (*Model a Percolation System*) To model a percolation system, create a data type `Percolation` with the following API:

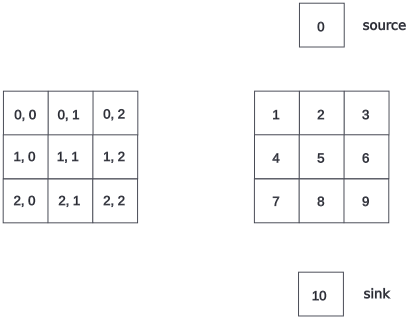
method	description
`Percolation(int N)`	creates an N-by-N grid, with all sites blocked
`void open(int i, int j)`	opens site (i, j)
`boolean isOpen(int i, int j)`	checks if site (i, j) is open
`boolean isFull(int i, int j)`	checks if site (i, j) is full
`int numberOfOpenSites()`	returns the number of open sites
`boolean percolates()`	checks if the system percolates

Hints

- Model percolation system as an $N \times N$ array of booleans (`true` $\Longrightarrow$ open site and `false` $\Longrightarrow$ blocked site)
- Can implement the API by scanning the array directly, but that does not meet all the performance requirements; use Union-find (UF) data structure instead
- Create an UF object with $N^2 + 2$ sites and use the private `encode()` method to map 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

Problems

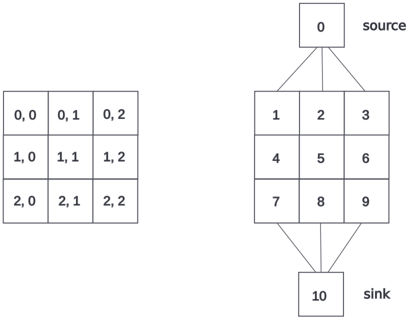
- A 3×3 percolation system and its UF representation



- 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`

Problems

- `private int encode(int i, int j)`
  - Return the UF site $(1 \ldots N)$ corresponding to the percolation system site (i, j)
- `public Percolation(int N)`
  - Initialize instance variables
  - Connect the sites corresponding to first and last rows of the percolation system with the source and sink sites respectively
  - The 3×3 system with its top and bottom row sites connected to the source and sink sites respectively



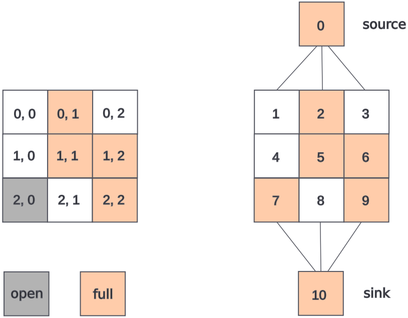
Problems

- `void open(int i, int j)`
  - Open the site `(i, j)` if it is not already open
  - Increment `openSites` by one
  - Check if any of the neighbors to the north, east, west, and south of `(i, j)` is open, and if so, connect the site corresponding to `(i, j)` with the 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 site is connected to the source site
- `int numberOfOpenSites()`
  - Return the number of open sites
- `boolean percolates()`
  - Return whether the system percolates or not; the system percolates if the sink site is connected to the source site

Problems

⇝ 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 technically not full since it is not connected to an open site in the top row via a path of neighboring (north, east, west, and south) open sites, but the corresponding site (7) is connected to the source, so is incorrectly reported as being full — this is the back wash problem

⇝ To receive full credit for the problem, you must address the back wash problem

Problem 2. (*Estimate Percolation Threshold*) To estimate the percolation threshold, create a data type `PercolationStats` with the following API:

method	description
`PercolationStats(int N, int T)`	performs T independent experiments on an N-by-N grid
`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

Hints

- ⇝ Instance variables
  - ⇝ Number of independent experiments, `int T`
  - ⇝ Percolation thresholds for the `T` experiments, `double[] p`

Problems

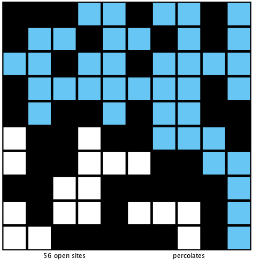
- `PercolationStats(int N, int T)`
  - Initialize instance variables
  - Perform the following experiment `T` times
    - Create an $N \times N$ percolation system
    - Until the system percolates, choose a site (i, j) at random and open it if it is not already open
    - Calculate percolation threshold as the fraction of sites opened, and store the value in `p[]`
- `double mean()`
  - Return the mean μ of the values in `p[]`
- `double stddev()`
  - Return the standard deviation σ of the values in `p[]`
- `double confidenceLow()`
  - Return $\mu - \frac{1.96\sigma}{\sqrt{T}}$
- `double confidenceHigh()`
  - Return $\mu + \frac{1.96\sigma}{\sqrt{T}}$

Problems

The data directory contains some input files for use with the percolation clients, and associated with most input .txt files are output .png files that show the desired output; for example

```
$ more data/input10.txt
10
9 1
1 9
...
3 4
7 9
```

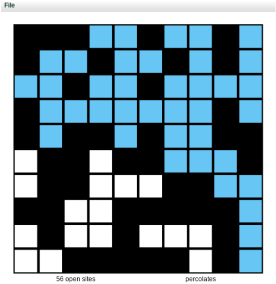
```
$ display data/input10.png
```



Problems

The visualization client `PercolationVisualizer` takes as command-line argument the name of a file specifying the size and open sites of a percolation system, and visually reports if the system percolates or not

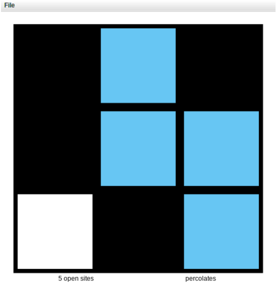
```
$ java PercolationVisualizer data/input10.txt
```



Problems

The visualization client `InteractivePercolationVisualizer` constructs an N-by-N percolation system, where N is specified as command-line argument, and allows you to interactively open sites in the system by clicking on them and visually inspect if the system percolates or not

```
$ java InteractivePercolationVisualizer 3
3
0 1
1 2
1 1
2 0
2 2
```



Epilogue

Use the template file `report.txt` to write your report for the project

Your report must include

- Time (in hours) spent on the project
- Difficulty level (1: very easy; 5: very difficult) of the project
- A short description of how you approached each problem, issues you encountered, and how you resolved those issues
- Acknowledgement of any help you received
- Other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Epilogue

Before you submit your files

- Make sure your programs meet the style requirements by running the following command on the terminal

```
$ check_style <program>
```

  where `<program>` is the `.java` file whose style you want to check

- Make sure your programs meet the input and output specifications by running the following command on the terminal

```
$ python3 run_tests.py -v [<items>]
```

  where the optional argument `<items>` lists the exercises/problems (`Exercise1`, `Problem2`, etc.) you want to test, separated by spaces; all the exercises/problems are tested if no argument is given

- Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and running time
- Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Epilogue

Files to submit

1. `GreatCircle.java`
2. `PrimeCounter.java`
3. `Distance.java`
4. `Transpose.java`
5. `Rational.java`
6. `Harmonic.java`
7. `Percolation.java`
8. `PercolationStats.java`
9. `report.txt`